Efficient HB Suffix

Apr 2008
David Barrett-Lennard

1 Introduction

This document concerns the representation of a History Buffer (HB) suffix used within a session for an interactive collaboration to transform remote operations.

2 Notation

- For vector times v_1,v_2, $v_1 \uparrow v_2$ is a vector time that denotes the *union* of vector times v_1,v_2.
- For vector times v_1,v_2, $v_1 \downarrow v_2$ is a vector time that denotes the *intersection* of vector times v_1,v_2.
- The empty vector time is denoted by $v_\varnothing$.
- op(s,t) denotes the t^{th} *atomic operation* performed on site s, starting at t = 0.
- $\chi(v) = \{ op(s,t) \mid t < v(s) \}$ denotes the *extent* of vector time v
- $\chi[v_1,v_2) = \chi(v_2) \backslash \chi(v_1)$
- $v_{in}(O)$ denotes the vector time associated with the *execution context* (or *input context*) on which O is defined.
- $v_{out}(O)$ denotes the vector time associated with the output state produced by execution of O.
- Given composite operation O, the extent of O is the set of atomic operations represented by O and is denoted by $\aleph(O) = \chi(v_{out}(O)) \setminus \chi(v_{in}(O))$.
- Composite operation O is *empty* if $v_{in}(O) = v_{out}(O)$.
- O_1,O_2 are *context equivalent* if $v_{in}(O_1) = v_{in}(O_2)$.
- O_1,O_2 are *contextually serialised* if $v_{out}(O_1) = v_{in}(O_2)$.
- The composite operation $O_1 \oplus O_2$ denotes the (non lossy) merge of O_1 and O_2 where O_1,O_2 are contextually serialised operations.
- $\sum L$ denotes the merge of all operations in L into a single composite operation, where L is a list of contextually serialized operations.

- $\sum[v_1,v_2)$ denotes the composite operation obtained by merging a list of contextually serialised operations corresponding to $\chi(v_2)\backslash\chi(v_1)$ and having execution context v_1.
- Lf(O,v) is shorthand for $\sum[v_{in}(O)\ ,\ v)$, called the *Lfactor of O with respect to v*.
- Rf(O,v) is shorthand for $\sum[v\ ,\ v_{out}(O))$, called the *Rfactor of O with respect to v*.

Definition: Let the conditional (b ? x : y) evaluate to x if b is true, or otherwise it evaluates to y.

Note that

- $v_{in}\ (\sum[v_1,v_2)) = v_1$
- $v_{out}(\sum[v_1,v_2)) = v_2$
- $v_{in}(O) \leq v_{out}(O)$
- Let O = op(s,t). Then $v_{in}(O)(s) = t$ and $v_{out}(O)(s) = t+1$.
- $op(s,t) \in \aleph(O) \iff v_{in}(O)(s) \leq t < v_{out}(O)(s)$

2.1 Notation for context equivalent and context serialized

Definition: We write $O_1 >> O_2$ if O_1,O_2 are *contextually serialised*. ie $v_{out}(O_1) = v_{in}(O_2)$.

Definition: We write $O_1 <> O_2$ if O_1,O_2 are *context equivalent*. ie $v_{in}(O_1) = v_{in}(O_2)$.

Definition: We write $O_1 >< O_2$ if O_1,O_2 are *context convergent*. ie $v_{out}(O_1) = v_{out}(O_2)$.

3 Quadratic complexity

In [1] the HB suffix is a per session transient linear sequence of atomic operations, and dual IT is used to transform incoming operations against the HB suffix. Unfortunately this becomes inefficient when the suffix becomes very large - eg when there has been a temporary network outage. Basically the problem is the quadratic order complexity of transforming a list against a list. For example if two sites have performed 10000 operations then the number of transformations required for their reconciliation is 100 million which is prohibitive.

3.1 Calculating a transient HB suffix

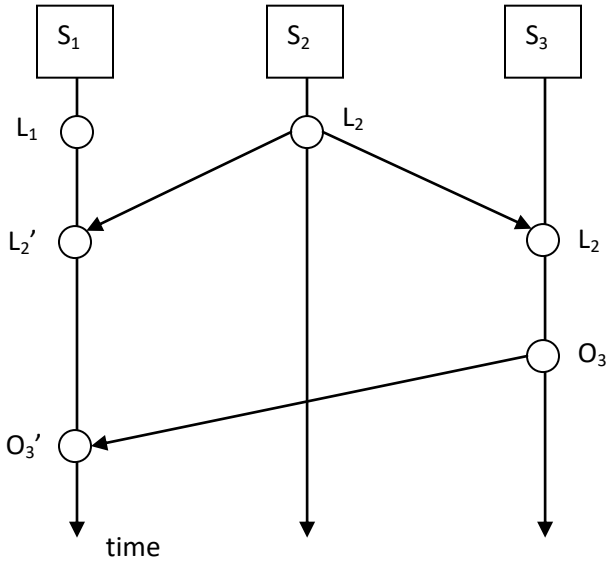
Given a local site HB with vector time v_{hb}, and a remote operation with execution context described by vector time v, there is a need to calculate the initial HB suffix – which consists of all the operations in $\chi(v_{hb})\backslash\chi(v)$. In [1] this involves:

1. Initialising the HB suffix by copying the tail of the persistent HB, beginning with the left most operation that is outside $\chi(v)$.
2. Taking the transpose of adjacent operations in the HB suffix in order to move operations outside $\chi(v)$ to the right of operations that are inside $\chi(v)$.
3. Popping (ie removing) a prefix consisting of the operations inside $\chi(v)$ from the HB-suffix.

The combination of steps 2 and 3 is called a *transpose and pop*.

Step 2 can be computationally expensive. The most expensive case occurs when there are a large number n_1 of operations outside $\chi(v)$ to the left of a large number n_2 of operations that are inside $\chi(v)$. The cost of transposing is $O(n_1\, n_2)$.

This can readily occur in practice. For example, let there be three sites S_1,S_2,S_3. In the diagram below, assume that S_1 connects to S_3 only after S_1 has already appended lists L_1 then L_2' to its local HB. When S_1 connects to S_3 and receives O_3 it will need to transpose L_1 and L_2' in order to calculate an appropriate HB suffix to transform O_3. This could be expensive if L_1 and L_2 have many atomic operations.



Steps 2,3 are also repeatedly applied *during* the session in order to bring the existing transient HB-suffix up to date as the execution context of received remote operations grows over time. Again, step 2 can be computationally expensive. Consider the same diagram again, now assuming that S_1 and S_3 had established a connection from the very beginning. At the time S_1 receives O_3 from S_3, S_1 will have already appended L_1 then L_2' to its transient HB suffix for its session with S_3. The execution context of O_3 shows that transpose of L_1 and L_2' will be required on its transient HB suffix.

4 Factorisation of an operation

Let L be a contextually serialised list of operations, associated with operations in $\chi(v_2)\backslash\chi(v_1)$ where v_1 and v_2 are causally valid vector times satisfying $v_1 \leq v_2$ and v_1 represents the execution context of L. Let $\sum$L denote the merge of all operations in L into a single composite operation.

Let v be a causally valid vector time satisfying $v_1 \leq v \leq v_2$. It should therefore be possible to transpose operations in L as required so that L is L_2 appended to L_1 – ie $L = L_1\, L_2$ where L_1

corresponds to operations in $\chi(v)\backslash\chi(v_1)$, and L_2 are the operations in $\chi(v_2)\backslash\chi(v)$. Note that $L_1 >> L_2$, $v_{in}(L_1) = v_1$, $v_{in}(L_2) = v$.

The *factorization* of operation $\sum L$ for given v returns $\sum L_1$ and $\sum L_2$. Factorisation can be regarded as the opposite of merging because $\sum L = \sum L_1 \oplus_{>>} \sum L_2$.

We therefore define the following functions that return the left and right factors of a given composite operation $\sum L$ for a given vector time v.

Definition: $Lf(\sum L,v) = \sum L_1$ denotes the *Lfactor with respect to v*, and $Rf(\sum L,v) = \sum L_2$ denotes the *Rfactor with respect to v*.

Note that transpose and pop on a transient HB-suffix is analogous to replacing it by its Rfactor.

4.1 More formal definition

Definition: Let O be a composite operation and v be a causally valid vector time satisfying

$$v_{in}(O) \leq v \leq v_{out}(O).$$

Then $Lf(O,v) = \sum[v_{in}(O) , v)$ and

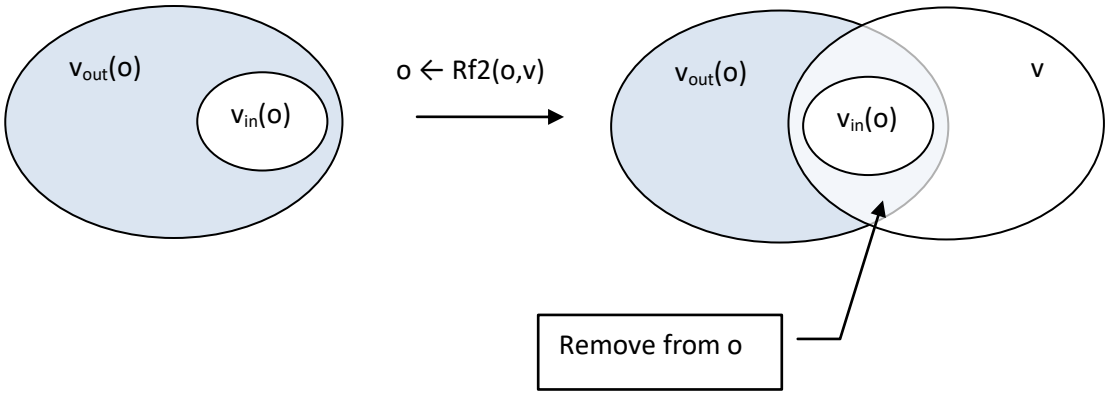
$$Rf(O,v) = \sum[v , v_{out}(O))$$

4.2 Rf2 function

It is convenient to define the following function:

```
Operation Rf2(Operation o, VectorTime v)
{
    assert(v_in(o) ≤ v);
    return Rf(o,v_out(o) ↓ v);
}
```

This relates to the following picture where from composite operation o associated with $v_{out}(o)\backslash v_{in}(o)$ we want to exclude that part that overlaps with some vector time v that is known to contain $v_{in}(o)$:



4.3 List based implementation of composite operations

There is a canonical implementation of composite operations based on the representation as an ordered list of atomic operations. Note the following:

- An empty list represents an empty composite operation
- The merge $O_1 \oplus O_2$ of context serialised composite operations O_1, O_2 is implemented using list concatenation.
- IT and ET on composite operations involves the list based extensions of IT and ET. See LIT1, LIT2, LIT3 defined in [2].
- Factorisation of a composite operation with respect to vector time v involves transpose of adjacent operations to separate the list into prefix and suffix, where the prefix is in $\chi(v)$ and the suffix is outside $\chi(v)$. The Lfactor is the prefix and the RFactor is the suffix.

5 Precedes relation on composite operations

Definition: Let M_1,M_2 be composite operations. We write $M_1 \rightarrow M_2$ if

$$O_1 \in \aleph(M_1), O_2 \in \aleph(M_2) \quad \Rightarrow \quad O_1 \rightarrow O_2$$

Definition: Let M_1,M_2 be composite operations. We write $M_1 \,||\, M_2$ if

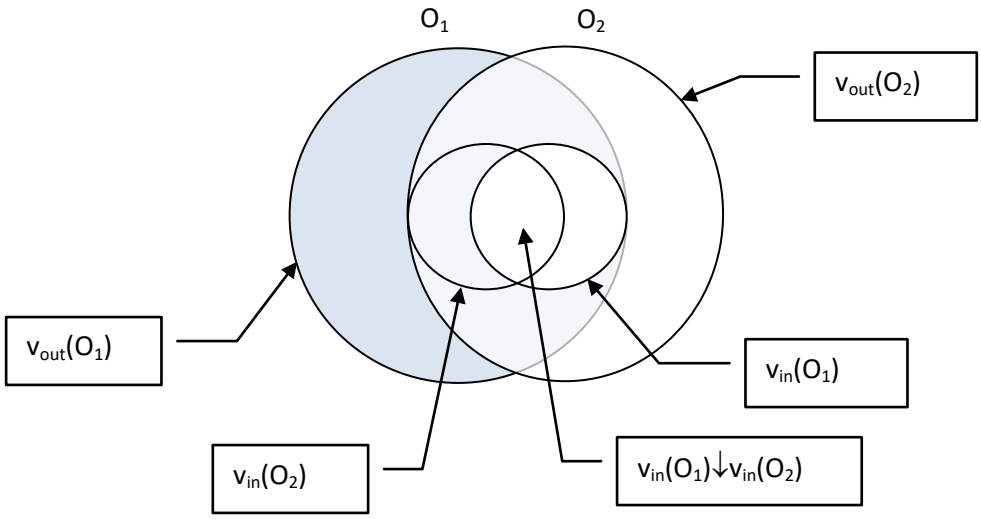
$$O_1 \in \aleph(M_1), O_2 \in \aleph(M_2) \quad \Rightarrow \quad O_1 \,||\, O_2$$

Note that it is possible to have $\neg(M_1 \rightarrow M_2) \wedge \neg(M_2 \rightarrow M_1) \wedge \neg(M_1 \,||\, M_2)$

6 Generalised symmetric merge

In this section we generalise the merge operator $\oplus$ so it no longer assumes that the operations being merged are contextually serialised.

Definition: Let operations O_1, O_2 satisfy $v_{in}(O_1) \leq v$ and $v_{in}(O_2) \leq v$
where $v = v_{out}(O_1) \downarrow v_{out}(O_2)$.
Then $O_1 \oplus O_2 = \sum[\, v_{in}(O_1) \downarrow v_{in}(O_2),\ v_{out}(O_1) \uparrow v_{out}(O_2)\,)$



Claim: $\aleph(O_1 \oplus O_2) = \aleph(O_1) \cup \aleph(O_2)$

Proof:

Note that : $\aleph(O) = \chi(v_{out}(O)) \setminus \chi(v_{in}(O))$

Note the set theoretic laws

$(A \cup B) \setminus C = (A \setminus C) \cup (B \setminus C)$

$A \setminus (B \cap C) = (A \setminus B) \cup (A \setminus C)$

Let $o_1 = \chi(v_{out}(O_1))$, $o_2 = \chi(v_{out}(O_1))$, $i_1 = \chi(v_{in}(O_1))$, $i_2 = \chi(v_{in}(O_1))$

Then the constraints are $i_1 \subseteq (o_1 \cap o_2)$ and $i_2 \subseteq (o_1 \cap o_2)$.

$\aleph(O_1 \oplus O_2)$

$= \chi(v_{out}(O_1 \oplus O_2)) \setminus \chi(v_{in}(O_1 \oplus\oplus O_2))$

$= \chi(v_{out}(O_1) \uparrow v_{out}(O_2)) \setminus \chi(v_{in}(O_1) \downarrow v_{in}(O_2))$

$= (\chi(v_{out}(O_1)) \cup \chi(v_{out}(O_2))) \setminus (\chi(v_{in}(O_1)) \cap \chi(v_{in}(O_2)))$

$= (o_1 \cup o_2) \setminus (i_1 \cap i_2)$

$= (o_1 \setminus (i_1 \cap i_2)) \cup (o_2 \setminus (i_1 \cap i_2))$

$= o_1 \setminus i_1 \cup o_1 \setminus i_2 \cup o_2 \setminus i_1 \cup o_2 \setminus i_2$

$= o_1 \setminus i_1 \cup o_2 \setminus i_2$ (todo : why?)

$= \aleph(O_1) \cup \aleph(O_2)$

Claim: $O_1 \oplus O_2 = O_2 \oplus O_1$

Proof: Follows from symmetry of the definition.

The condition $v_{in}(O_1) \leq v$ and $v_{in}(O_2) \leq v$ implies that it is possible to factorise both O_1 and O_2 with respect to v. Informally we say that O_1,O_2 have *sufficient history.*

Claim: $Rf(O_1,v) \,||\, Rf(O_2,v)$

Proof: Suppose $\exists\, O_x \in \aleph(Rf(O_1,v))$ and $\exists\, O_y \in \aleph(Rf(O_2,v))$ such that $O_x \rightarrow O_y$. Now $O_y \in \aleph(Rf(O_2,v)) \subseteq \chi(v_{out}(O_2))$. But $v_{out}(O_2)$ is a causally valid vector time so therefore $O_x \in \chi(v_{out}(O_2))$ as well. Now $O_x \in \aleph(Rf(O_1,v)) \subseteq \chi(v_{out}(O_1))$. Therefore $O_x \in \chi(v_{out}(O_1)) \cap \chi(v_{out}(O_2)) = \chi(v_{out}(O_1) \downarrow v_{out}(O_2)) = \chi(v)$ which contradicts $O_x \in \aleph(Rf(O_1,v))$. In a similar manner we obtain a contradiction with $O_y \rightarrow O_x$.

Ignoring differences between $v_{in}(O_1)$ and $v_{in}(O_2)$, we could say that the merge of O_1,O_2 entails a dualIT between $Rf(O_1,v)$ and $Rf(O_2,v)$.

To simplify the control algorithm we regard the merge and factorisation operators as more fundamental than IT or ET!

7 Asymmetric merge

In this section we define a generalised asymmetric merge operator $\oplus_R$.

Definition: Let operations O_1, O_2 satisfy $v_{in}(O_1) \leq v$ and $v_{in}(O_2) \leq v$
where $v = v_{out}(O_1) \downarrow v_{out}(O_2)$.
Then $O_1 \oplus_R O_2 = \sum[\, v_{in}(O_1), v_{out}(O_1) \uparrow v_{out}(O_2)\,)$

Conceptually $O_1 \oplus_R O_2$ is obtained by starting with O_1 and merging in all the atomic operations that are in $v_{out}(O_2) \setminus v_{out}(O_1)$.

It is not generally the case that $\aleph(O_1 \oplus_R O_2) = \aleph(O_1) \cup \aleph(O_2)$, because $v_{in}(O_1 \oplus_R O_2) = v_{in}(O_1)$. This lack of symmetry in the definition means we don't generally expect $O_1 \oplus_R O_2 = O_2 \oplus_R O_1$ unless $v_{in}(O_1) = v_{in}(O_2)$.

The condition $v_{in}(O_1) \leq v$ and $v_{in}(O_2) \leq v$ implies that it is possible to factorise both O_1 and O_2 with respect to v. Informally it means that O_1,O_2 each have *sufficient history.*

As for the symmetric merge, $Rf(O_1,v) \,||\, Rf(O_2,v)$.

Claim: $O_1 \oplus_R O_2 = O_1 \oplus_C IT(Rf(O_2,v), Rf(O_1,v)$
where $v = v_{out}(O_1) \downarrow v_{out}(O_1)$.

However in practice it doesn't make sense for the implementation to treat $\oplus_C$ and IT as independent operators, because merging implicitly involves dual IT, so it is better for the implementation to implement IT and merge as a single operator $\oplus_R$. Furthermore the

implementation should return the transformed O_2, or else allow for O_2 to be executed as part of the merge.

8 Using a single primitive merge operator

Consider that we avoid needing to treat database states within our mathematical model by thinking of a state as nothing more than the result of all the operations that created that state. Then if operations converge under merging it necessarily follows that database states do as well.

Furthermore let's pretend that operations happen to carry all the information required to be applied from the initial state. In other words an operation is identified with an entire history buffer and therefore is uniquely specified for a given output vector time. At this level of abstraction the execution context of the operation is irrelevant. This means that states and operations are really one and the same! We have in effect dropped the whole idea that operations represent deltas.

The point of this is to reduce the system to a single primitive operator (denoted by $\oplus$) that merges any two given operations (or states or history buffers if you prefer).

Let O_{hb} represent a site's entire history buffer. O_{hb} can also be regarded as an operation or a state. The control algorithm is very simple:

Generate and apply local operation:

generate O;
$O_{hb} = O$;

Send operation:

send O_{hb};

Receive operation, merge with local HB and apply required changes

receive O;
$O_{hb} = O_{hb} \oplus O$;

Note that there is no need to separately apply an operation because the database state is identified directly with O_{hb}.

The properties required of this operator are simply:

- $O_1 \oplus O_2 = O_2 \oplus O_1$
- $(O_1 \oplus O_2) \oplus O_3 = O_1 \oplus (O_2 \oplus O_3)$

Associativity and commutativity are sufficient to ensure all sites converge at quiescence. This can be compared to TP1 and TP2 properties for IT which is comparatively complex, particularly in light of the rather complicated proof of convergence by Ressel et al.

This "solution" appears hideously inefficient. For example, every time an operation is sent, the entire history buffer is sent. When an operation is generated, we actually create what amounts to

an entire history buffer rather than just a small delta. When we receive an operation we somehow merge it with the entire history buffer.

However, looks can be deceiving! We are thinking of this solution at a very abstract level and we expect the implementation will indeed only generate and send efficient deltas. Nevertheless if such an implementation can be shown to be mathematically equivalent to the above solution then we have a very simple way to explain how all sites converge at quiescence.

A good reason to promote this underlying system rather than a system more directly based on the inclusion transform is to avoid a hasty use of contextually serialised lists of deltas which quickly lead to inefficient quadratic order reconciliation algorithms.

As an illustrative example, consider the case of a database state that is a 32 bit integer and the operations, thought of as deltas to the database state are positive or negative offsets. A solution directly based on lists and the inclusion transform is straightforward but very inefficient. By contrast a solution based on merging operations into composite operations can be much more efficient.

9 A new set of primitive operators

There are 4 primitive operators:

1. $Lf(O,v)$ defined wherever $v_{in}(O) \leq v \leq v_{out}(O)$
2. $Rf(O,v)$ defined wherever $v_{in}(O) \leq v \leq v_{out}(O)$
3. $O_1 \oplus_{<>} O_2$ defined wherever $O_1 <> O_2$ and $O_1 \,||\, O_2$.
4. $O_1 \oplus_{>>} O_2$ defined wherever $O_1 >> O_2$

Required properties:

- $v_{in}(Lf(O,v)) = v_{in}(O)$
- $v_{out}(Lf(O,v)) = v$
- $v_{in}(Rf(O,v)) = v$
- $v_{out}(Rf(O,v)) = v_{out}(O)$
- $v_{in}(O_1 \oplus_{<>} O_2) = v_{in}(O_1) = v_{in}(O_2)$
- $v_{out}(O_1 \oplus_{<>} O_2) = v_{out}(O_1) \uparrow v_{out}(O_2)$
- $v_{in}(O_1 \oplus_{>>} O_2) = v_{in}(O_1)$
- $v_{out}(O_1 \oplus_{>>} O_2) = v_{out}(O_2)$
- $Lf(O,v) \oplus_{>>} Rf(O,v) = O$

- $O_1 \oplus_{<>} O_2 = O_2 \oplus_{<>} O_1$
- $(O_1 \oplus_{<>} O_2) \oplus_{<>} O_3 = O_1 \oplus_{<>} (O_2 \oplus_{<>} O_3)$
- $(O_1 \oplus_{>>} O_2) \oplus_{>>} O_3 = O_1 \oplus_{>>} (O_2 \oplus_{>>} O_3)$
- $v_{in}(O_1) = v_{in}(O_2) \Rightarrow Lf(O_1,v) = Lf(O_2,v)$
- $v_{out}(O_1) \leq v \Rightarrow Lf(O_1 \oplus_{>>} O_2,v) = O_1 \oplus_{>>} Lf(O_2,v)$
- $v_{out}(O_1) \leq v \Rightarrow Rf(O_1 \oplus_{>>} O_2,v) = Rf(O_2,v)$

Associativity and commutativity of $\oplus_{||}$ is very powerful because it readily demonstrates that all sites will converge at quiescence. This can be compared to TP1 and TP2 properties for IT which is comparatively complex. Note as well that the conventional approach requires the mathematical model to introduce the concept of the database state, applying operations on the database state, and ensuring that the output database states are equivalent where convergence is required. We instead avoid the notion of state in out model since it can always be regarded as a function of a composite operation, so to prove convergence it is sufficient to prove that composite operations converge.

TODO: Relate associativity and commutativity of $\oplus_{<>}$ to an equivalence to the previous system where all operations O have $v_{in}(O) = v_{\varnothing}$, and there is only one associative and commutative merge operation $\oplus$.

TODO: Relate this more directly to how $\sum[v_1,v_2)$ can be defined.

TODO: Define the control algorithm that all this leads to (still avoiding the need to execute an operation, because each site stores O_{hb} satisfying $v_{in}(O_{hb}) = v_{\varnothing}$)

Claim: $v_1 \leq v_2 \Rightarrow Lf(Lf(O,v_2),v_1) = Lf(O,v_1)$

We can define a more general merge operator as follows:

Definition: Let O_1,O_2 be context equivalent operations (ie satisfying $v_{in}(O_1) = v_{in}(O_2)$) and let $v = v_{out}(O_1) \downarrow v_{out}(O_2)$. Then we define

$$O_1 \oplus_C O_2 = Lf(O_1,v) \oplus_{>>} (Rf(O_1,v) \oplus_{<>} Rf(O_2,v))$$

Claim $v_{in}(O_1 \oplus_C O_2) = v_{in}(O_1) = v_{in}(O_2)$
Proof: $v_{in}(O_1 \oplus_C O_2) = v_{in}(Lf(O_1,v) \oplus_{>>} (Rf(O_1,v) \oplus_{||} Rf(O_2,v))) = v_{in}(Lf(O_1,v)) = v_{in}(O_1)$

Claim $v_{out}(O_1 \oplus_C O_2) = v_{out}(O_1) \uparrow v_{out}(O_2)$
Proof:

$v_{out}(O_1 \oplus_C O_2)$
$= v_{out}(Lf(O_1,v) \oplus_{>>} (Rf(O_1,v) \oplus_{<>} Rf(O_2,v)))$
$= v_{out}(Rf(O_1,v) \oplus_{<>} Rf(O_2,v))$
$= v_{out}(Rf(O_1,v)) \uparrow v_{out}(Rf(O_2,v))$
$= v_{out}(O_1) \uparrow v_{out}(O_2)$

Claim: $O_1 \oplus_C O_2 = O_2 \oplus_C O_1$
Proof: $O_1 \oplus_C O_2$
$= Lf(O_1,v) \oplus_{>>} (Rf(O_1,v) \oplus_{<>} Rf(O_2,v))$
$= Lf(O_2,v) \oplus_{>>} (Rf(O_2,v) \oplus_{<>} Rf(O_1,v))$
$= O_2 \oplus_C O_1$

Claim: $(O_1 \oplus_{>>} O_2) \oplus_C (O_1 \oplus_{>>} O_3) = O_1 \oplus_{>>} (O_2 \oplus_C O_3)$
Proof: let $v = v_{out}(O_2) \downarrow v_{out}(O_3)$.
Note that $v_{out}(O_1 \oplus_{>>} O_2) = v_{out}(O_2)$, and $v_{out}(O_1 \oplus_{>>} O_3) = v_{out}(O_3)$
So $v = v_{out}(O_1 \oplus_{>>} O_2) \downarrow v_{out}(O_1 \oplus_{>>} O_3)$
$v_{out}(O_1) = v_{in}(O_2) = v_{in}(O_3)$.
Now $v_{in}(O_2) \le v_{out}(O_2)$ and $v_{in}(O_3) \le v_{out}(O_3)$
Therefore $v_{out}(O_1) \le v_{out}(O_2) \downarrow v_{out}(O_3) = v$
$(O_1 \oplus_{>>} O_2) \oplus_C (O_1 \oplus_{>>} O_3)$
$= Lf(O_1 \oplus_{>>} O_2,v) \oplus_{>>} (Rf(O_1 \oplus_{>>} O_2,v) \oplus_{<>} Rf(O_1 \oplus_{>>} O_3,v))$
$= O_1 \oplus_{>>} Lf(O_2,v) \oplus_{>>} (Rf(O_2,v) \oplus_{<>} Rf(O_3,v))$
$= O_1 \oplus_{>>} (O_2 \oplus_C O_3)$

Claim: $(O_1 \oplus_C O_2) \oplus_C O_3 = (O_2 \oplus_C O_1) \oplus_C O_3$
Proof: Note that $v_{in}(O_1) = v_{in}(O_2) = v_{in}(O_3)$.
let $v_i = v_{out}(O_i)$
$v_{out}(O_i \oplus_C O_j) = v_i \uparrow v_j$
Let $v_{123} = v_1 \downarrow v_2 \downarrow v_3$
So $Lf(O_1,v_{123}) = Lf(O_2,v_{123}) = Lf(O_3,v_{123})$

$(O_1 \oplus_C O_2) \oplus_C O_3$
$= Lf((O_1 \oplus_C O_2),v') \oplus_{>>} (Rf((O_1 \oplus_C O_2),v') \oplus_{<>} Rf(O_3,v'))$ where $v' = (v_1 \uparrow v_2) \downarrow v_3$
$= \ldots$ hmmm!
$= Lf(O_1,v_{123}) \oplus_{>>} (Rf(O_1,v_{123}) \oplus_{||} Rf(O_2,v_{123}) \oplus_{<>} Rf(O_3,v_{123}))$
$= \ldots$
$= Lf(O_1,v'') \oplus_{>>} (Rf(O_1,v'') \oplus_{<>} Rf(O_2 \oplus_C O_3,v''))$ where $v'' = v_1 \downarrow (v_2 \uparrow v_3)$
$= O_1 \oplus_C (O_2 \oplus_C O_3)$

Claim: $\sum[v_1,v_2)$ can be defined

10 Notes on causality with regard to the primitive operators

Consider two sites using the conventional approach of history buffers that are linear lists of operations that only grow as operations are appended at the ends of the lists.

Site S_1 HB = $[a_1\ a_2\ a_3\ a_4]$

Site S_2 HB = $[a_1\ a_2\ b_1\ a_3\ b_2]$

S_2 is about to send b_1 and b_2. The operations that causally precede b_1 are $\{ a_1,a_2 \}$, whereas the operations that causally precede b_2 are $\{ a_1,a_2,a_3,b_1 \}$

In the conventional approach S_2 will send b_1 with a different execution context than b_2. The result is that the dual IT performed on S_1 must always involve concurrent operations.

However in the new proposed approach, S_2 will send an RFactor of its HB that contains both b_1 and b_2, and it is impossible for S_1 to know that a_3 precedes b_2 but not b_1. Our approach actually assumes that it is nevertheless possible for S_1 to merge this RFactor directly into its HB and ignore the fact that it can't actually tell for sure which of its operations are concurrent with the remote operation.

It is important to note that when we factorise the S_2 HB into Lf = { a_1 a_2 a_3 } and Rf = { b_1 b_2 } it is not the case that all atomic operations in Lf causally precede the atomic operations in Rf. Nevertheless we propose that such a factorisation is meaningful - because there is no operation in Rf that precedes an operation in Lf - which is exactly what is meant by a causally valid vector time.

The conventional approach can be seen as necessarily recording much more information about causality. The proposal suggests that this is not necessary - even to the extent of avoiding the need to implement a solution in terms of an inclusion transform that assumes that the operations are concurrent.

TODO: The subsequent treatment of assignment contains errors because of exactly this issue. It turns out that the proposal doesn't actually lead to uniqueness of a merge because of this very problem.

11 Operations for deltas on a numerical field

Let the document state consist of a single field of a type supporting a binary operator (which we henceforth call '+') that is associative and commutative. Examples are

- Sets under union
- Sets under intersection
- Bags under union
- Arbitrary sized integers under addition.
- Arbitrary sized integers under multiplication.
- Integers under addition using modular arithmetic for some fixed modulus n. For example, with $n = 2^{32}$ we have the 32 bit integers.
- Matrices under addition
- Rational numbers represented with arbitrary sized numerator and denominator under addition.
- Complex numbers under addition

We assume that all sites agree on the initial value of the field. Let an operation be associated with applying an additive offset to the field. Clearly such offsets can be applied to the field in any order. Therefore achieving convergence at all sites is very easy.

Let a composite operation be represented as a set of triples (s,t,o) where a triple is associated with an atomic operation generated on site s with sequence number t on that site. The offset o represents an additive delta to be applied to the field.

A composite operation stores these triples in a set rather than a list because the order in which the offsets are applied is immaterial to the result.

Definition: Let v be a causally valid vector time satisfying $v_{in}(O) \le v \le v_{out}(v)$. Then we define:

- $LF(O,v) = \{ (s,t,o) \in O \mid t < v(s) \}$
- $RF(O,v) = \{ (s,t,o) \in O \mid t \ge v(s) \}$

Definition: Given O_1, O_2 let $O_1 \oplus O_2 = O_1 \cup O_2$

Claim: $\oplus$ is commutative and associative.

Claim: $LF(O,v) \oplus RF(O,v) = O$

Definition. Let $O_1 >> O_2$. Then we define $O_1 \oplus_{>>} O_2 = O_1 \oplus O_2 = O_1 \cup O_2$

Definition. Let $O_1 <> O_2$. Then we define $O_1 \oplus_{<>} O_2 = O_1 \oplus O_2 = O_1 \cup O_2$

todo: Write out proofs of all the properties required by Lf,Rf,$\oplus$

11.1 Implementation

Let a composite operation record a map keyed by siteid to an ordered list of the following struct

```
struct Offset
{
    int t;
    T offset;
};
```

Factorisation with respect to a given vector time involves, for each map entry keyed by siteid s, the split of the list of offsets into prefix satisfying $t < v(s)$ and suffix satisfying $t \ge v(s)$. The position about which to split the list can be found in O(log n) time using binary search. If the prefix is empty then the corresponding map entry in the L-factor may be removed. Similarly it may be possible to remove map entries in the R-factor.

Merging of operations simply involves concatenation of the lists of offsets as appropriate.

12 Composite operations on bags based on deltas

Let the document state consist of a single field of type bag<T>. Consider that for each possible $k \in T$, c_k denotes the number of elements in the bag with value k. In other words c_k is the multiplicity of k.

Consider furthermore that an operations on the bag to insert an element with value k is regarded as applying an offset of +1 to c_k. Similarly an operation that removes an element with value k is regarded as an offset of -1 to c_k.

The nice thing about this approach is that the solution for deltas is directly appropriate and we know that this solution is very efficient (and of course ensures that all sites converge).

We would like to enforce the constraint that c_k cannot become negative. However concurrent removals from a bag can certainly lead to that. Even worse, consider the case where initially the bag has one element and concurrently site S_1 removes the element, site S_2 also removes the element and site S_3 inserts an element. Then the solution based on deltas results in $c_k = 0$ - ie the bag is empty which is probably not very sensible because S_1,S_2 were able to remove an element that they never knew about!

TODO: Investigate a solution which is compatible with conservation of matter and without negative amounts appearing.

12.1 Deletes that identify the element they are deleting

Consider that we pretend the elements of the bag have identity according to the (s,t) of the operation that inserted that element. By accounting for identity in this way we can ensure that a delete operation is only able to delete elements that causally preceded it.

So for given k an operation records a set I_k of (s,t) insertions and a set D_k of (s,t) deletions. These two sets are independently managed using delta semantics based on set union. Both of these sets can only increase over time. Note that when we generate a delete operation it is important to distinguish the (s,t) assigned to it as an atomic operation versus the (s,t) used to identify the element being removed.

It is straightforward to calculate the set difference $P_k = I_k \backslash D_k$. We define the multiplicity of k in the bag to be $c_k = |P_k|$. Note by this approach we never get $c_k < 0$. Also deletes can never dominate concurrent insertions.

This approach can also be used for sets, by defining presence in the set by the condition $c_k > 0$.

12.1.1 Implementation

We store a map keyed by k. This takes us to a map keyed by s. This takes us to

1. a set of enabled insertions recorded as an ordered list of t values.
2. a set of disabled insertions recorded as an ordered list of t values.

3. a set of deletions recorded as a set of (t,si,ti) triples, where t is the sequence number assigned to the deletion as an atomic operation and (si,ti) identified the insertion being deleted.

The distinction between enabled and disabled insertions is a caching concern that is only relevant to an operation with v_{in} = {}. We want to wangle it do that when we merge a remote op into the local HB we don't actually need to re-enable an insertion when we calculate the LFactor. Arguably factorisation shouldn't be separated out from concurrent merge.

When an operation is to be merged, we basically want to apply the new deletions. This should be quite fast if we use binary search amongst the remaining enabled insertions. That allows these insertions to be moved over to the deleted pile. This is where we decrement c_k.

When calculating an RFactor to be sent over the wire we don't distinguish between enabled/disabled insertions. As far as we're concerned we're sending the whole lot.

12.1.2 Problem

A problem with this approach: What happens when the bag is initially non-empty? How do we record the deletes? Note that when an interactive collaboration begins relative to some define base vector time we will certainly have nonempty bags.

13 Composite operations for insertions into a string

Let the document state be a text field (ie single string of characters), and the only operations are stringwise insertions. Let an operation store the following state

- The vector times v_{in} and v_{out}
- A vector<Entry> ordered by insertion position, that are specified in post insertion coords.

```
struct Entry
{
    SiteId s;
    int t;
    int q;       // insertion position
    string str; // string to be inserted
};

Operation LFactor(O,v)
{
    int offset = 0;
    Lf.vin = O.vin;
    Lf.vout = v;
    Lf.entrys = [];
    for each e in O
    {
        if (e.t < v(e.s)) Lf.entrys += Entry(e.s, e.t, e.q - offset, e.str);
        else offset += e.str.size();
    }
    return Lf;
}

Operation RFactor(O,v)
{
    Rf.vin = v;
```

```
    Rf.vout = O.vout;
    Rf.entrys = [];
    for each e in O
    {
        if (e.t >= v(e.s)) Rf.entrys += e;
    }
    return Rf;
}

Operation MergeContextEquivalent(O1,O2)
{
    int offset1 = 0;
    int offset2 = 0;
    // linear scan through both lists, shifting to right according to the other
    // offset.   When offset insertions are at the same place then resolve
    // the tie using siteid comparisons.  Accumulate offsets as we go.
}

Operation MergeContextSerialised(O1,O2)
{
}
```

13.1 Ensuring MP3

Let O_1,O_2 represent insertions on a single text field. Each operations record insertion intervals ordered by q-position.

Let O_1 <> O_2. The dual IT of O_1,O_2 involves simultaneous left to right scans through both O_1 and O_2 in a manner reminiscent of merge sort. As the algorithm proceeds we accumulate a counter of the total number of characters inserted so far by O_1. Similarly we have a counter for O_2. In order to compare the positions of the next interval from O_1 and the next from O_2, we must offset the interval from O_1 to the right by the number of insertions by O_2. Similarly we must offset the interval from O_2 by the number of insertions by O_1.

If the positions compare equal then it would appear we must compare siteids to break the tie. This suggests the following algorithm:

```
struct Interval
{
    SiteId s;
    int t;
    int q;
    string str;
    Interval* next;  // ptr to next interval, or NULL
};

void DualIT(Interval* i1, Interval* i2)
{
    int s1 = 0;     // Accumulated characters inserted by i1
    int s2 = 0;     // Accumulated characters inserted by i2
    while(i1 && i2)
    {
        int d = (s1 + i2->q) - (s2 + i1->q);
        if (d < 0 || d == 0 && i2->s < i1->s)
        {
            // Process i2
            i2->q += s1;
            s2 += i2->str.size();
            i2 = i2->next;
        }
```

```
            else
            {
                // Process i1
                i1->q += s2;
                s1 += i1->str.size();
                i1 = i1->next;
            }
        }
        if (s1)
        {
            while (i2)
            {
                i2->q += s1;
                i2 = i2->next;
            }
        }
        if (s2)
        {
            while (i1)
            {
                i1->q += s2;
                i1 = i1->next;
            }
        }
    }
```

However this algorithm is incompatible with the MP3 property [2]. The following example illustrates the problem:

Let there be three sites with siteids satisfying $S_0 < S_1 < S_2$. Initially all sites have an empty text field. The following steps are performed.

1. S_0 generates local operation $O_0 = S_0$: insert "00"
2. S_1 generates local operation $O_1 = S_1$: insert "1"
3. S_2 receives O_0 from S_0, and inserts "00"
4. S_2 generates local operation $O_2 = S_2$: insert "2", to get "020"
5. S_2 receives O_1 from S_1, and calculates $v = v_{out}(S_1.hb) \downarrow v_{out}(S_2.hb) = (0,1,0) \downarrow (1,0,1) = (0,0,0)$. S_2 factorises its HB according to v = (0,0,0), so Lf = "", Rf = "020". It then merges "1" to yield "0120" which is wrong – it should have got "0201". The reason is that "1" must come after "00".

Note that $O_1 \,||\, O_0$ and $O_1 \,||\, O_2$, so we have merged two concurrent R-factors. Evidently rule MP3 is broken:

(MP3) $\quad IT(O_1, O_0 \oplus O_2) = IT(O_1, [O_0, O_2])$

The problem is associated with how to correctly IT past the merge of O_0,O_2 where $O_0 \rightarrow O_2$.

13.2 Dual IT of O_1,O_2 again

Let $O_1 <> O_2$. Consider that $\aleph(O_1)$ contains O_x,O_y with $O_x \rightarrow O_y$. Let each insertion interval in an operation record the vector time of the context in which it was originally generated. Let $gc(O_y)$ denote the generational vector time of atomic operation O_y. Then by [4]

$$O_x \rightarrow O_y \text{ iff } O_x \in \chi(gc(O_y)).$$

Given non-empty operation O,

$$\exists\, O_y \in \aleph(O) \text{ such that } (\neg\, \exists\, O_x \in \aleph(O) \text{ such that } O_x \rightarrow O_y).$$

Hence O can be factorised into $O = O_x \oplus_{>>} O'$ for some O'. It follows that given O_1, O_2 we can factorise out atomic LFactors like this:

$O_1 = O_x \oplus_{>>} O_1'$
$O_2 = O_y \oplus_{>>} O_2'$

This provides a conceptual basis for recursively decomposing the problem into a simpler one (using properties MP3 and MP4 from [2]).

13.3 Siteid inheritance

Consider that there is a concept of defining an *effective siteid* for a given interval that may be different to its actual siteid. In the example above, the insertion of "2" in the middle of "00" would be deemed to have an effective siteid of S_0 instead of S_2 because $O_0 \rightarrow O_2$. This would fix the problem in that particular example.

However the following suggests that inheritance cannot work: Let there be four sites with siteids $s_1 < s_2 < s_3 < s_4$

1. s_1 generates $O_1 = s_1$: insert "1"
2. s_3 generates $O_3 = s_3$: insert "3"
3. s_4 generates $O_4 = s_4$:insert "4"
4. s_2 receives O_1,O_4, to give "14"
5. s_2 generates $O_2 = s_2$:insert "2" to give "124"
6. s_2 receives O_3 to give "1234"

In this case $O_1 \rightarrow O_2$ and $O_4 \rightarrow O_2$ so it would seem appropriate for O_2 to inherit some siteid different from its actual. However, its actual site id s_2 was needed to determine the insertion position of the "3" relative to the "2".

It is instructive to consider the above example, where instead $s_2 < s_1 < s_3 < s_4$. In that case the "3" must still be between the "1" and the "4". So we still get "1234". If instead we have $s_1 < s_3 < s_4 < s_2$ then again s_3 must still be inserted between the "1" and the "4", but this time we get "1324".

Consider that we think of causality as creating a constraint applied *early* to narrow down the possible insertion location. In the above, since $O_1 \rightarrow O_2$ and $O_4 \rightarrow O_2$ we must initially ignore O_2! ie we pretend we only see "14" instead of "124". This can be compared to taking an LFactor – and it must be a subsequence without changing the relative order. This narrows down the insertion position to between the "1" and the "4". We repeat this process as often as necessary to eat away at the causal dependencies, until none remain. When none remain we should find that the remaining insertions are ordered by siteid, so it is sufficient to insert according to this ordering.

It is worth noting that any approach that helps to define an ordering compatible with the precedes relation may help allow for efficient factorization.

13.4 The largest LFactor we can take

For given operation O, it would seem appropriate to take the largest LFactor such that there are no causal dependencies between any two atomic operations within the LFactor.

$$Lf = \{ O_y \in \aleph(O) \mid \neg \exists\, O_x \in \aleph(O) \text{ such that } O_x \rightarrow O_y \}$$

It is obvious that such a factorization doesn't violate causality (ie there is no $O_x \in \aleph(Rf)$ and $O_y \in \aleph(Lf)$ such that $O_x \rightarrow O_y$).

Note that all operations within Lf are concurrent. Therefore we should find that intervals in Lf are ordered by siteid, and therefore there is a unique insertion position.

13.5 3-partitions of q-contiguous operations

Definition: Let operation O represent insertions on a single given text field. Let the insertion intervals be recorded in an ordered list, expressed in post insertion coordinates (ie q coordinates). We say that O is *q-contiguous* if there are no gaps in q-space between adjacent intervals. In other words, execution of O is equivalent to the insertion of a single string into the text field.

O = [….)[……)[..)[……)[….)

Note that if O contains no intervals, or only a single interval then we assume O is q-contiguous.

Definition: If O is q-contiguous, let O.q denote the q position of its left most character (if any) and O.size denote the total size of its intervals.

Definition: Given lists L_1, L_2, let $L_1 + L_2$ denote the concatenation of L_1 and L_2 (ie L_2 appended to the end of L_1). Note that concatenation is associative but doesn't commute.

Claim: L_1 and L_2 are q-contiguous and $L_1.q + L_1.size = L_2.q \Leftrightarrow (L_1+L_2)$ is q-contiguous.

Definition $[x_1,\ldots,x_n]$ denotes the list containing elements $x_1,x_2,\ldots.,x_n$ in that order.

Definition: Let O be nonempty and q-contiguous. A *3-partition* of O is some partition of O into three parts as follows:

O = A+[b]+C

where b is an interval in O satisfying $\neg$ ($\exists$ $O_x \in \aleph(O)$ such that $O_x \rightarrow b$).

It follows that

- A and C are q-contiguous (and possibly empty)
- A.q = O.q
- b.q = A.q + A.size
- C.q = b.q + b.size.

The condition $\neg$ ($\exists$ $O_x \in \aleph(O)$ such that $O_x \rightarrow b$) shows that a valid factorisation exists that extracts L-factor [b] from O (because it doesn't break causality). Therefore we can write the factorisation into contextually serialised left and right parts as follows:

$$O = [b'] \oplus_{>>} (A+C)$$

The Lfactor algorithm involves decrementing the q-position of [b] as it ETs backward past A, resulting in b'.q = O.q. The RFactor algorithm doesn't alter q positions (ie A.q and C.q are unchanged by factorisation), so we know a gap in q coordinates of b.size has appeared between A and C. Therefore the R-factor (A+C) is not q-contiguous if both A and C are non-empty.

Definition: A 3-partition is *non-trivial* if either A or C in the above definition is non empty.

13.6 The merge of concurrent q-contiguous operations at different positions

Let shift(O,n) represent the result of shifting all insertion intervals in O to the right by n characters.

Let O_1,O_2 be q-contiguous, satisfy O_1 || O_2 and O_1 <> O_2. Suppose $O_1.q < O_2.q$. The merge of O_1,O_2 is:

$$O_1 \oplus_{<>} O_2 = O_1 + \text{shift}(O_2, O_1.\text{size})$$

where + denotes list concatenation.

Proof:

First we show that it is true if O_2 is a single interval [b_2] :

> Let O_1 be q-contiguous and let O_2 = [b_2] be a single interval. Let O_1 || O_2 and O_1 <> O_2. Suppose $O_1.q < O_2.q$. The merge of O_1,O_2 is:
>
> $$O_1 \oplus_{<>} O_2 = O_1 + \text{shift}(O_2, O_1.\text{size})$$
>
> Proof: We show this by induction on the number of intervals in O_1.
>
> (n = 1) : This follows readily

($n \rightarrow n+1$) : Let there be a 3-partition $O_1=A_1+[b_1]+C_1$. As an L-factor b_1 satisfies $b_1.q = O_1.q < O_2.q = b_2.q$, so b_2 will be shifted to the right by $b_1.size$ at it ITs past b_1. Since $A_1.size < O_1.size$, by the induction premise b_1 will be shifted to the right by $A_1.size$ as it ITs past A_1. This results in b_2 shifted by $b_1.size + A_1.size$. Therefore at this point the shifted $b_2.q$ satisfies $C_1.q = O_1.q + A_1.size + b_1.size < b_2.q$. Since $C_1.size < O_1.size$, by the induction premise b_2 will be shifted to the right as it IT's past C_1. This results in $b_2.q$ being shifted by a further $C_1.size$. So overall b_2 is shifted by $O_1.size = A_1.size + b_1.size + C_1.size$.

Let there be a 3-partition $O_2=A_2+[b_2]+C_2$. As an L-factor b_2 satisfies $b_2.q = O_2.q > O_1.q$, so it follows that the first b_2 will be shifted to the right by $O_1.size$ at it ITs past O_1.

This leaves the R-factor (A_2+C_2) remaining in O_2. If the next interval in temporal order is in A_2 then as an LFactor it has position $O_2.q$. Otherwise if it comes from C_2 then as an LFactor it will have position $O_2.q + b_2.size$. It follows that it will be shifted to the right by $O_1.size$ as it ITs past all of O_1.

Over time more gaps will appear in the remaining R-factor of O_2, so extracted intervals (as LFactors) tend to have q positions shifted to the right relative to the original $O_2.q$. Therefore they are always shifted by $O_1.size$ as they IT past O_1.

Note that at no time was it necessary to compare siteids.

13.7 The merge of concurrent q-contiguous operations at the same position

We are interested in defining the merge of O_1,O_2 where O_1,O_2 are each q-contiguous and satisfy $O_1 || O_2$, $O_1 <> O_2$ and $O_1.q = O_2.q$.

Evidently the merge $O_{sum} = O_1 \oplus_{<>} O_2$ will also be q-contiguous and satisfy $O_{sum}.q = O_1.q = O_2.q$, $O_{sum}.size = O_1.size + O_2.size$.

Since O_1 is q-contiguous, the q-position of each interval is determined by $O_1.q$ and the sizes of all the intervals that come before it. Similarly for O_2 and O_{sum}. This suggests that the recorded q-positions are redundant.

The goal is to define a merge algorithm that completely ignores q-position values, and instead uses them only implicitly by virtue of the order in which the insertion intervals are stored.

13.7.1 Merging an interval into a context equivalent, concurrent, coincident, contiguous operation

Let O_1 be q-contiguous and have a 3-partition $O_1 = A_1+[b_1]+C_1$ and let $O_2 = [b_2]$ where $O_1 <> O_2$ and $O_1 || O_2$. Let O_1,O_2 be coincident - meaning that $O_1.q = O_2.q$. To simplify the following exposition assume WLOG $O_1.q = O_2.q = 0$. We are interested in the merge of O_2 into O_1 using an algorithm that ignores q-position values.

Let O_1 be factorised as follows

$$O_1 = [b_1] \oplus_{>>} (A_1+C_1)$$

Then we know as an LFactor, b_1 satisfies $b_1.q = 0$.

The dual IT between O_1,O_2 first involves comparison of siteids of b_1,b_2 because they are coincident at q = 0.

Firstly assume $b_1.s < b_2.s$ so b_2 will appear on the right of b_1 in the effects document. Under dual IT of b_1,b_2 we see that after transformation $b_1.q = 0$ whereas $b_2.q = b_1.size$. b_2 then dualITs against (A_1+C_1) which is not q-contiguous. Since $b_2.q = b_1.size > 0$ we see that the first interval of A_1, having q = 0 is necessarily to the left of b_2. This pushes b_2 further to the right. In fact when b_2 is compared to any subsequent intervals of A_1, b_2 will always be pushed further to the right. So eventually b_2 IT's past all of A_1 resulting in $b_2.q = b_1.size + A_1.size$. Now b_2 will dualIT with C_1 which also begins at $q = A_1.size + b_1.size$, so b_2 and C_1 are coincident. Therefore following our idea of ignoring q-positions in the merge algorithm, we see that b_2 needs to be merged into C_1 in a recursive fashion. The conclusion is that $O_1 \oplus_{<>} O_2 = A_1 + [b_1] + (C_1 \oplus_{<>} [b_2])$.

[todo: this is sloppy – we need to more formally bring in the idea that we have developed an algorithm that ignores q positions]

Otherwise assume $b_2.s < b_1.s$ so b_2 will appear on the left of b_1 in the effects document. When b_2 IT's past b_1 we end up with $b_2.q = 0$. Therefore b_2 will dualIT with (A_1+C_1) at the same q position. In fact we can do better than that because we know that b_2 will appear to the left of b_1 so it is sufficient to dual IT b_2 with A_1 only.

This suggests the following result:

Given 3-partition of $O_1 = A_1+[b_1]+C_1$ and $O_2 = [b_2]$ where $O_1 <> O_2$ and $O_1 \,||\, O_2$ then

$$O_1 \oplus_{<>} O_2 = (b_1.s < b_2.s) \; ? \; (A_1+[b_1]+ (C_1 \oplus_{<>} [b_2])) \; : \; ((A_1 \oplus_{<>} [b_2]) +[b_1]+C_1)$$

As the algorithm proceeds there is a range of intervals that narrows over time until we have found the proper insertion position.

The approach is similar to binary search in the way at each step we have a range of potential insertion positions, and using a comparison to some point within the range we narrow the search down to either the left or the right "half". However, binary search normally allows us to compare to a point midway in the range, but we have no such luck. Instead we must make comparisons to intervals in an order compatible with the partial ordering defined by the precedes relation. Therefore there is the potential to only eat away at the range one interval at a time.

Presumably on average this algorithm is O(n log n), and in the worst case is $O(n^2)$ where n is the number of intervals in O_1. Maybe it doesn't matter anyway because it's not common to have many concurrent insertions in the same field at exactly the same position. [Actually what it we merge the work of two independent groups of users that have typed in thousands of characters into an initially empty document?]

13.7.1.1 Using a total temporal ordering

Consider that we record a total ordering on the intervals that is compatible with the partial ordering of the precedes relation. This total ordering can be recorded using a 32 bit *temporal*

index position which we denote by u in each interval. This is only a minor increase in storage space, and doesn't complicate serialisation or splitting of intervals.

Note therefore that $O_x \rightarrow O_y \Rightarrow O_x.u < O_y.u$

The converse is false.

An alternative could be for insertion intervals to simultaneously take part in two double linked lists - ie for both a spatial ordering by q and a temporal ordering by u. However we reject this approach for the following reasons:

- this doesn't appear to be a very good fit for the merge algorithm because global u position order within a large O_1 may take us for a long detour and we need to skip past many intervals until we reappear back inside the range of interest relevant to the merge algorithm. This could easily be much more expensive than a simple linear scan for minimum u if the range of interest is small.
- Serialisation of an operation is complicated
- It is less space efficient than simply storing a temporal index in each interval
- Splitting intervals is complicated

13.7.1.2 Merge algorithm

Consider that (i_1,i_2) represents an open range of intervals in O_1. In other words the range is exclusive of positions i_1,i_2 and i_1 and i_2 are respectively *one before* and *one after* sentinels. If the range is empty (i.e. $next(i_1) = i_2$) then we stop and insert at the position between i_1 and i_2. Otherwise we scan the intervals in (i_1,i_2) for the interval with the minimum temporal index u. This yields an interval j satisfying $i_1 < j < i_2$. Depending on the siteid comparison we either assign $i_1=j$ or else $i_2 = j$. Note therefore that the range must get smaller at each iteration which guarantees that the loop will eventually terminate.

```
// Merge coincident interval x into contiguous range of intervals
// in (i1,i2)
void MergeCoincidentIntervalIntoContiguousRange(i1,i2,x)
{
    while(next(i1) != i2)
    {
        scan j in (i1,i2) for smallest u
        if (x.s < j.s) i2 = j; else i1 = j;
    }
    insert(i1,x);  // insert x immediately after i1.
}
```

Example:

Let (s,u) denote an insertion interval with siteid s and temporal index u.

In the following we depict the insertion intervals ordered left to right by q position. Let $s_1 < s_2 < s_3 < s_4$ and $u_1 < u_2 < u_3$

O_1 intervals: (s_1,u_2) (s_2,u_3) (s_4,u_1)

O_2 intervals: (s_3,u_1)

Consider that we pick the first (and only) interval from O_2 which is (s_3,u_1). Our job is to find its insertion position amongst the intervals of O_1. Initially there are 3 intervals and hence 4 possible insertion positions.

First we compare (s_3,u_1) with (s_4,u_1). Comparing siteids we deduce that (s_3,u_1) is to the left of (s_4,u_1). This narrows the potential insertion positions according to the intervals to the left of (s_4,u_1). ie (s_1,u_2) and (s_2,u_3). So now there are only 3 potential insertion positions.

Next we compare (s_3,u_1) with (s_1,u_2) and find that (s_3,u_1) must be to the right of (s_1,u_2). This leave us with only (s_2,u_3) in the range, and we deduce that (s_3,u_1) must be inserted between (s_2,u_3) and (s_4,u_1).

13.7.1.3 Assigning the temporal index as we merge operations

As we merge operations into a history buffer there should be no need to change the temporal index values of the existing intervals. In fact the basic idea is for the HB to record the next u, and as intervals are added to the HB we assign u values to these new intervals as required. This can be regarded as analogous to thinking of the HB in its conventional form of a growing linear list of operations ordered by u.

13.7.2 Merging of two 3-partitions

Let O_1, O_2 be q-contiguous and satisfy $O_1 <> O_2$ and $O_1 \,||\, O_2$. Let $O_1.q = O_2.q = 0$.

Our approach to merge O_1,O_2 is to merge one interval at a time from O_2 into O_1, until all intervals from O_2 have been merged into O_1. We process the intervals of O_2 in temporal order. We expect that each time we merge an interval from O_2 into O_1, O_1 will remain q-contiguous.

Let there be a 3-partition of $O_2 = A_2+[b_2]+C_2$. If we L-factorise b_2 we get:

$$O_2 = [b_2] + (A_2+C_2)$$

In this form $b_2.q = A_2.q = 0$. A gap in q coordinates of b_2.size has appeared between A_2 and C_2.

The merge of $[b_2]$ into O_1 is straightforward (see section ?) and yields O_1' containing b_2 somewhere. However for the purposes of merging in the remaining intervals from O_2, it is actually important to L-factorise $[b_2]$ from O_1', which means a gap may well appear in the remaining RFactor of O_1'.

After removal of b_2 from O_2 we are left with the following R-factor:

$$O_2 := (A_2 + C_2)$$

Now we want to take the next 'b_2' interval (in temporal order) from O_2 as an LFactor. There are two cases. If it lies in A_2 then in similar fashion the interval b_2 to be extracted will satisfy $b_2.q = 0$. However, if b_2 is inside C_2 then as an LFactor it will have $b_2.q > 0$. In either case this LFactor of O_2 is suitable for merging into the RFactor of O_1' that doesn't contain the intervals that have already been added from O_2.

Claim: When a b_2 is L-factored out of O_2, as it ETs backward through earlier intervals of O_2, the gaps from missing intervals on its left are in effect summed in order to determine its resultant q position as an LFactor of O_2. Then when it is merged with the relevant RFactor of O_1, the gaps caused by the very same O_2 intervals that have previously been added cause a neat book keeping effect such that the merge will in fact be applied to a q-contiguous section bounded below and above by the relevant b_2's appearing now on O_1 according to their original q-position order in O_2.

Upshot: We can implement an algorithm that is based around the idea that an interval is merged into a coincident, concurrent, contiguous range of intervals. To merge O_2 into O_1 we simply iterate through intervals of O_2 in temporal order, and merge them into a range within O_1 that is bounded by the intervals that have already been inserted so as to ensure we maintain the existing q-position order in O_2.

Note that we should merge O_2 into O_1 if O_2 has fewer intervals. Otherwise it is better to instead merge intervals from O_1 into O_2.

13.7.2.1 Merge algorithm

The interesting aspect here is how to efficiently keep track of the locations of the intervals from O_2 that have been inserted so far into O_1, so that when we insert the next interval from O_2 we can quickly find the q-contiguous range of intervals from O_1 into which it should be merged.

Assuming double linked lists in order of q-position, for a given interval from O_2, we can quickly find the previous and next intervals in O_2. However they may not have been inserted into O_1 yet. So that doesn't really help us.

Consider that we separately record an array of pointers to the intervals from O_2 that have already been inserted into O_1. Then when we want to insert the next interval from O_2 we can perform a binary search to find the insertion position with respect to the intervals from O_2 that have already been inserted into O_1. More to the point this gives us the intervals below and above. Now since these intervals have already been inserted into O_1 they form part of the double linked list ordered by q-position. The upshot is that we have found a pair of iterators that bound the range in which the next insertion must be made.

In the following let X and Y be arrays of pointers to intervals.

```
// Merge O2 = (j1,j2) into O1 = (i1,i2) where O1,O2 are q-contiguous,
// concurrent and coincident.
void MergeCoincidentRanges(
    Interval* i1,Interval* i2,
    Interval* j1,Interval* j2)
{
    Copy pointers to intervals in (j1,j2) to an array X;
    Sort pointers to intervals in X by their u values;
    Y = [];
    for each j in X  (in order of increasing u)
    {
        // Y is ordered by q-position.   Insert j into Y using binary
        // search according to q-position
        // Returns pointers to intervals y1,y2 that straddle interval j.
        // May return y1=NULL or y2=NULL (or both)
        (y1,y2) = Y.insert(j);   // insert j in Y
        if (!y1) y1 = i1;
```

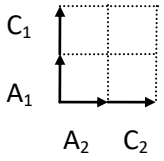
```
            if (!y2) y2 = i2;

            // Merge j into open range (y1,y2)
            MergeCoincidentIntervalIntoContiguousRange(y1,y2,j);
        }
    }
```

13.7.2.2 Recursive formulation of the algorithm

Lemma: Let $O_1 = A_1+C_1$, $O_2 = A_2+C_2$, where A_1,A_2,C_1,C_2 are each q-contiguous. Let A_1,A_2 be q-coincident. Let there be a gap of x>0 characters between the end of A_1 and the start of C_1. Let there be the same sized gap x between A_2 and C_2. Then

$$(A_1+C_1) \oplus (A_2+C_2) = (A_1 \oplus A_2) + (C_1 \oplus C_2)$$



Proof: (very informal)

> Given the gap between A_1,C_1 and the gap between A_2,C_2, we see that A_1,A_2 are the next maximal q-contiguous ranges to be merged. When A_2 IT's past A_1 we know that the merged result consist of character positioned in the range [0,A_1+size+A_2.size). Inevitably all characters in $A_1 \oplus A_2$ are judged as being on the left of the characters in C_1. Therefore after C_1 IT's past A_2, C_1 will be shifted further to the right by A_2.size and be positioned at A_1.size+A_2.size+x. Similarly after C_2 ITs past A_1, C_2 will be shifted to the right by A_1.size and so will also be positioned at A_1.size+A_2.size+x. Therefore C_1,C_2 will be q-coincident when they are merged.

Let there be a 3-partition of O_2 as $A_2+[b_2]+C_2$. As an L-Factor we know that b_2.q = 0. Therefore b_2 is coincident with O_1 and by section ? we have an algorithm to find the insertion position of b_2 within O_1. Let is write this as

$$O_1 = A_1 + [b_2] + C_1.$$

i.e. the interval b_2 we inserted into O_1 can be written as a 3-partition of O_1. This leaves us with $O_2 = A_2+C_2$.

Claim: $O_1 \oplus O_2 = (A_1 \oplus A_2) + [b_2] + (C_1 \oplus C_2)$

Proof: (very informal)

This follows readily by taking the RFactor of $O_1 = A_1 + [b_2] + C_1$ without $[b_2]$, and the RFactor of $O_2 = A_2+[b_2]+C_2$ without $[b_2]$. This leaves us with the merge of A_1+C_1 with A_2+C_2. By the previous lemma $(A_1+C_1) \oplus (A_2+C_2) = (A_1 \oplus A_2) + (C_1 \oplus C_2)$. It only remains to undo the factorisation that took out [b2] as an LFactor.

13.7.2.3 Effective siteids

Consider that the first interval in temporal order in O_1 is an interval with siteid s_1, and the second is an interval with siteid s_2 and appearing on the left of the first.

O_1: s_2 s_1

If $s_2 < s_1$ then this breaks up O_1 neatly into three regions. Otherwise if $s_1 < s_2$ then we actually have a *no man's land* between them.

O_1: s_2 s_1
$<s_1$ NONE $>s_1$

We can emulate the selection process by setting the *effective siteid* of all intervals in this range to s_1.

Alternatively consider that s_2 is to the right of s_1. Then we have

O_1: s_1 s_2

If $s_1 < s_2$ then this breaks up O_1 neatly into three regions. Otherwise if $s_2 < s_1$ then we again have a *no man's land* between them. Again, we can emulate the selection process by setting the *effective siteid* of all intervals in this range to s_1.

Now consider s_1,s_2,s_3. There are lots of cases to consider! The intervals can appear in 6 different orders by q position and 6 different orders by siteid, giving 36 combinations! It is not clear whether all these are possible.

13.7.2.4 Calculation of effective siteids

Let O be q-contiguous. We want to calculate the *effective siteid* and *effective temporal index* of its intervals. It is claimed we can do this with essentially a simple left to right linear scan. [TODO: That's not quite true - there are possibly some right to left scans involved as well - maybe enough to make it $O(n^2)$ in pathological cases]. Let the intervals be ordered by q-position and each interval records its generating siteid s and temporal index u.

The following algorithm in C++ assumes there are n given q-contiguous intervals ordered by q-position and for each i in [0,n), s[i] is the siteid and u[i] is the temporal index of the i^{th} interval. The function calculates the effective siteid es[i], and the effective temporal index eu[i] for each i. We assume n > 0.

```
void CalculateEffectiveSiteids(const int s[],const int u[],int n, int es[], int eu[])
{
     es[0] = s[0];
     eu[0] = u[0];
     for (int i=1 ; i < n ; ++i)
     {
         if (es[i-1] <= s[i])
         {
             es[i] = s[i];
             eu[i] = u[i];
         }
         else
         {
             assert(eu[i-1] != u[i]);
             if (eu[i-1] < u[i])
```

```
                {
                    // i-1 dominates i, so i inherits es,eu from i-1
                    es[i] = es[i-1];
                    eu[i] = eu[i-1];
                }
                else
                {
                    // i dominates i-1, so we must scan right to left as far as
                    // needed to assign the s,u of i to the es,eu of itself and the
                    // preceding intervals
                    int j = i;
                    do
                    {
                        es[j] = s[i];
                        eu[j] = u[i];
                        --j;
                        if (es[j] > s[i])
                        {
                            if (eu[j] < u[i])
                            {
                                // j dominates i, so we need to assign the es,eu
                                // of j to all the intervals in [j+1,i].
                                for (int k=j+1 ; k <= i ; ++k)
                                {
                                    es[k] = es[j];
                                    eu[k] = eu[j];
                                }
                                break;
                            }
                        }
                        else
                        {
                            break;
                        }
                    } while (j > 0);
                }
            }
        }
    }
```

Example:

s	2	4	6	8	2	5	5	3
u	6	0	7	4	6	5	3	2

Consider that we iterate through the intervals from O in order of q-position. While the siteid monotone increases we simply assign es and eu from s,u

es	2	4	6	8
eu	6	0	7	4

Now we want to add the interval with s=2, u=6. We can no longer simply assign es,eu from s,u because it will break the requirement that siteids monotone increase from left to right. There are two possibilities, depending on the comparison of u coord to the eu of the last interval in the list we are building. In this case the new interval has a larger (ie losing) u coord so it inherits the es=8,eu=4 from the tail of the list:

es	2	4	6	8	8
eu	6	0	7	4	4

Now we want to add s=5, u=5. Again we have a lower siteid than at the tail of the list we are building, so we must compare u values. Again we inherit es=8, eu=4:

es	2	4	6	8	8	8
eu	6	0	7	4	4	4

Now we want to add the interval with s=5,u=3. This time the entry to be added dominates the tail of the list because it has a lower u coord. This dominating s,u is propagated, starting from the tail of the list using a right to left scan until the requirement that siteids monotone increase is established once again:

es	2	4	5	5	5	5	5
eu	6	0	3	3	3	3	3

Finally we want to add the interval with s=3, u=2. Again this dominates intervals in the tail so we propagate es=3,eu=2 from right to left. Eventually we each the interval with es=4,eu=0. This dominates all the intervals that come after it, so we have to scan left to right again applying es=4, eu=0. The end result is:

es	2	4	4	4	4	4	4	4
eu	6	0	0	0	0	0	0	0

It is conceivable that any risk of $O(n^2)$ behavior can be eliminated by using a representation that partitions this list into sub-lists and we avoid the need to explicitly iterate through a sub-list when applying a different (es,eu).

13.7.2.5 Notes

For any given operation we can formalise the concept of breaking it up into as few q-contiguous pieces as possible. ie each q-contiguous piece is maximal because there are no adjacent pieces to which it can be merged to form a larger q-contiguous range.

For each maximal q-contiguous piece we can calculate the effective siteids as shown above. As a result we end up with a very simple dual IT algorithm!

Note that for an operation O with $v_{in}(O) = v_{\varnothing}$, it will always be the case that (for a given text field) O is q-contiguous. In particular the entire HB is always q-contiguous. Gaps only appear when we take an RFactor of O with respect to some causally valid vector time. The calculated effective siteids of the intervals depend on the locations of the gaps. This shows that there is no point calculating effective siteids for the entire HB because they won't generally be applicable to an RFactor relevant to dual IT.

13.7.2.6 Maintaining effective siteids in a transient suffix

The only remaining ingredient is to determine how to update the effective siteids under merging. More specifically given two q-coincident, q-contiguous operations, what are the effective siteids of the merged result? Well of course there is nothing to do because the merge results in a single q-contiguous result that is ordered by effective siteid, and working in the manner of merge sort.

It would seem that local operations are the culprit as far as upsetting the ordering by siteid in q-contiguous sections. So we need to investigate what happens to effective siteids when we insert an interval. Note that this will always have the largest u coord.

The rule is very simple:

When an insertion operation is generated, we consider the characters to the left and to the right of the insertion position. We refer to these as the neighbors. If the local site's siteid falls between the siteids of the neighbors there is nothing to do. However if it is smaller than both neighbors then it inherits s,u from the neighbor with smallest s. If it is larger than both then it inherits the s,u from the neighbor with largest s.

PROBLEM: This doesn't account for the fact that factorisation is often required on a transient suffix, and factorisation affects the effective siteids.

14 Control algorithm using merge and factorisation

In the following we describe a control algorithm that avoids the assumption that composite operations are implemented as contextually serialised lists of atomic operations - and in particular a linear list of atomic operations that comprise a local history buffer, or a list of atomic operations for a transient HB-suffix.

Instead the proposed control algorithm makes use of abstractly defined composite operations that represent arbitrarily large changes to the database, and provide the IT, merge and factorization operators. The potential benefit is the avoidance of the quadratic complexity implicit with using linear lists of atomic operations to represent composite operations.

Persistent working set state:

s	The site identifier for the working set
t	The next t coordinate for a locally generated operation, initialised to zero when the working set is first created.
hb	A single composite operation equal to the merge of all operations that have been executed on the working set. In general $v_{in}(hb) = v_{\varnothing}$.
	The database objects on which operations are performed

```
struct WorkingSet
{
    SiteId s;
    int t;
    Operation hb;
    Mutex m;
};

void DoLocalOperation(WorkingSet& w,Operation o)
{
    o.Init(w.s, w.t++);       // Set (s,t) in all intervals
    {
        Lock lock(w.m);
        w.hb ⊕= o;
        o.Execute();
```

```
    }
}

void RunSession(WorkingSet& w, OutStream& os, InStream& is)
{
    {
        Lock lock(w.m);
        os << v_out(w.hb);  // Send vector time describing content of local HB
    }

    VectorTime vr;
    is >> vr;           // Receive vector time describing content of remote HB

    async repeat        // Thread to send operations
    {
        Lock lock(w.m);
        os << Rf(w.hb, v_out(w.hb) ↓ vr);
        vr ↑= v_out(w.hb);
    }
    async repeat        // Thread to receive operations
    {
        Operation o;
        is >> o;
        {
            Lock lock(w.m);
            VectorTime v = v_out(w.hb) ↓ v_out(o);
            o = IT(Rf(o,v),Rf(w.hb,v);
            w.hb ⊕= o;
            o.Execute();
        }
    }
}
```

Notes:

- A session is always between a *pair* of sites. A site can hold any number of sessions with its sites at the same time.
- A given *communication channel* is associated with a sender site that uses an output stream ‘os’ to write vector times or operations, and another receiver site that uses an input stream ‘is’ to read vector times or operations. It is assumed that the channel behaves in the manner of a reliable transient FIFO queue. For simplicity we have ignored issues of send and receive buffers, flushing etc.
- A session involves full duplex communication using two independent channels as described above. The protocol involves the sending of a single vector time following by a stream of composite operations.
- When a session begins, each site sends its local v_{out}(hb) which summarises the set of operations that have been applied at that site. By the time this is received by the other site it may only represent an *underestimate* of what operations are present.
- Each session hosts two worker threads – one to repeatedly send operations and another to repeatedly receive operations.

- The working set contains a mutex to serialise access to all of its state. Both worker threads need to lock the mutex before they are granted access.

- Since we assume that a given site only has an underestimate of what's already present on the remote site, it is necessary to allow for the possibility that an operation o is received that (perhaps partially) contains atomic operations that are already present. This is achieved by taking the RFactor of o with respect to $v_{out}(hb)$ in order to remove from o, all those atomic operations that are already contained within the HB. This could potentially render o empty! Note that the assertion in the call to Rf2 won't fail – ie $v_{in}(o) \leq v_{out}(hb)$ because the sender can never send an operation performed in the context of atomic operations that are not already present on the receiver.

- An atomic operation is never sent more than once. This follows from the fact that we send an RFactor of the HB with respect to a vector time vr that is subsequently unioned with $v_{out}(hb)$. Therefore any existing atomic operations in the HB will not be sent again.

- The execution context of sent operations can only increase over the life of the session. ie if O_1 is sent before O_2 then $v_{in}(O_1) \leq v_{in}(O_2)$. This is obvious from the fact that we send an operation O calculated as the RFactor of the HB with respect to vr so therefore $v_{in}(O) = vr$. The result follows from the fact that vr is only changed by assignment from a union with itself so the new vr contains the previous vr.

15 Control algorithm with transient HB-suffix

Transient per session state:

vr	A vector time describing what the sender thread has already sent to the remote site.
suffix	A composite operation equal to an Rfactor of the working set hb with respect to some vector time.

```
struct WorkingSet
{
    // Persistent state
    SiteId s;
    int t;
    Operation hb;

    // Transient state
    set<Session*> sessions; // The current set of active sessions
};

struct Session
{
    VectorTime vr;
    Operation suffix;
};

void DoOperation(WorkingSet* w, Session* y, Operation o)
```

```
{
    w->hb ⊕= o;
    o.Execute();
    for each x ∈ w->sessions such that x ≠ y
    {
        x->suffix ⊕= o;
    }
}

void DoLocalOperation(WorkingSet* w,Operation o)
{
    o.Init(w->s, w->t++);       // Set (s,t) in all intervals
    DoOperation(w,NULL,o);
}

void RunSession(WorkingSet* w,Session* y)
{
    os << v_out(w->hb); // Send vector time describing content of local HB
    is >> y->vr;        // Receive vector time describing content of remote HB
    y->suffix = Rf2(w->hb,y->vr);
    async repeat        // Separate thread to send operations
    {
        os << Rf2(y->suffix, y->vr);
        y->vr ↑= v_out(y->suffix);
    }
    async repeat        // Separate thread to receive operations
    {
        Operation o;
        is >> o;
        o = Rf2(o,v_out(w->hb));
        y->suffix = Rf2(y->suffix, v_in(o));
        DualIT(o,y->suffix);
        DoOperation(w,y,o);
    }
}
```

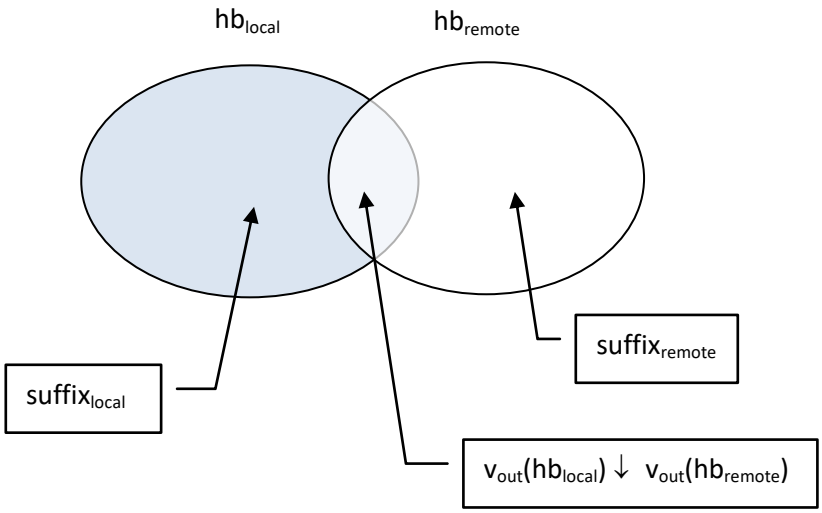
Notes:

- When the vector time describing the context of the remote HB is received, a session is able to calculate the inial value of its suffix which is a composite operation that is an RFactor of the site's HB.
- For simplicity we have ignored showing where mutexes are required. Each session needs a mutex to protect access to its suffix by its sender and receiver threads. A working set needs a mutex to protect the database objects and its hb.
- A received operation o must be transformed against all the local atomic operations that are outside its execution context. This is basically the session's transient suffix. However it is first necessary to take the RFactor of the suffix to exclude any atomic operations that are already in $v_{in}(o)$ which is the execution context of o. Note that the assertion in the call to Rf2 won't fail – ie $v_{in}(suffix) \leq v_{in}(o)$. The proof follows.

Proof (by induction)

(initial case)

When two sites first connect they exchange values of $v_{out}(hb)$ and compute initial suffixes as follows:



Initially $v_{in}(suffix_{local}) = v_{in}(suffix_{remote}) = v_{out}(hb_{local}) \downarrow v_{out}(hb_{remote})$. When a remote site subsequently sends the first operation o, it is calculated as an RFactor from $suffix_{remote}$ and therefore $v_{in}(suffix_{remote}) \leq v_{in}(o)$. It follows therefore that $v_{in}(suffix_{local}) \leq v_{in}(o)$.

(inductive step)

We have shown that the execution context of sent operations can only increase. ie if O_1 is sent before O_2 then $v_{in}(O_1) \leq v_{in}(O_2)$. Therefore if $v_{in}(suffix_{local}) \leq v_{in}(O_1)$ then for the same suffix it must follow that $v_{in}(suffix_{local}) \leq v_{in}(O_2)$.

The only way that $v_{in}(suffix_{local})$ may change is be assigning it to $v_{in}(O)$ for some received operation O. Of course this also preserves $v_{in}(suffix_{local}) \leq v_{in}(O)$.

16 Assignment operations

The problem is partly addressed by treating assignment operations specially. According to [2] we can avoid the whole need for a HB suffix to transform remote operations by using a map from FieldId to the right most enabled assignment in the HB, and by using backward chaining of enabled assignment operations for a given FieldId.

In the following treatment we go even further because we don't even want to store a history buffer as a linear list of operations.

Consider that a composite assignment operation O stores the following state:

- $v_{in}(O)$ and $v_{out}(O)$, or what amounts to the same thing, for each siteid record a half open interval $[t_1,t_2)$ for the operations that are present in the operation.

- A map<FieldId, list<AssignmentEntry> > that records entries for enabled assignments. Disabled assignments are not recorded.

where we have defined

```
struct AssignmentEntry
{
    SiteId s;
    int t;
    Value v;
};
```

Definition: Let L_i denote the i^{th} element of list L. We say that list L of assignment entries is *causally ordered* if

$$i < j \iff L_i \rightarrow L_j$$

Note that if L is causally ordered then there can be no entries with $L_i \,||\, L_j$.

Definition: A subsequence of a list is any list formed from a subset of the elements as long as they appear in the same relative order.

Obviously any sub-sequence of a causally ordered list is also causally ordered.

Definition: Given causally valid vector time v, let Q(v) be characterised as follows:

1. $v_{in}(Q(v)) = v_{\varnothing}$ and $v_{out}(Q(v)) = v$
2. Q(v) only contains a subset of the atomic assignments $op(s,t) \in \chi(v)$ on the field.
3. Q(v) is causally ordered.
4. Q(v) is maximal, meaning no additional operations can be inserted into the sequence without breaking the causal ordering.

Claim: Q(v) is uniquely defined.

The list<AssignmentEntry> is interpreted as an ordered sequence of enabled assignments, where each assignment dominates all previous assignments in the sequence. The list is assumed to be causally ordered.

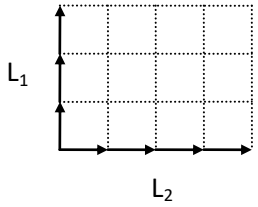
The justification for not storing disabled assignments (or indeed q positions) is that the proposed control algorithm never uses transpose (or ET).

In the following we focus attention on assignment operations on the same FieldId.

16.1 Dual IT between two lists

Let L_1, L_2 be concurrent, context equivalent, causally ordered lists of atomic assignment operations on the same field.

The dualIT transforms every operation in L_1 with every operation in L_2 according to the following figure.



Definition: Let s_∞ be a special siteid value which compares larger to all other siteids. A site cannot use s_∞ for its siteid.

Definition: Let $s_{min}(L)$ be the minimum siteid that appears in L, or else s_∞ if L = [].

Definition: $L_1 \oplus L_2 = (s_{min}(L_1) < s_{min}(L_2)) \ ? \ L_1 : L_2$

L_1, L_2 are causally ordered so therefore $L_1 \oplus L_2$ must be causally ordered.

Claim: $L_1 \oplus L_2 = L_2 \oplus L_1$

Proof: $L_1 \oplus L_2 = (s_{min}(L_1) < s_{min}(L_2)) \ ? \ L_1 : L_2 = (s_{min}(L_2) < s_{min}(L_1)) \ ? \ L_2 : L_1 = L_2 \oplus L_1$

Claim: $(L_1 \oplus L_2) \oplus L_3 = L_1 \oplus (L_2 \oplus L_3)$

Proof: Let $s_i = s_{min}(L_i)$. Then $L_i \oplus L_j = (s_i < s_j \ ? \ L_i : L_j)$ and $s_{min}(L_i \oplus L_j) = \min \{s_i, s_j\}$

$(L_1 \oplus L_2) \oplus L_3$

$= \min \{s_1, s_2\} < s_3 \ ? \ (s_1 < s_2 \ ? \ L_1 : L_2) : L_3$

$= (s_1 < \min \{s_2, s_3\} \ ? \ L_1 : (s_2 < s_3 \ ? \ L_2 : L_3))$

$= L_1 \oplus (L_2 \oplus L_3)$

Associativity and commutativity of $\oplus$ shows that at quiescence all sites will converge to the same representation of the operation which represents the merge over all assignments performed across all sites. ie we have a uniqueness of representation.

Claim: $(L_1 \oplus L_2) \sim (L_1 + IT(L_2, L_1))$.

Proof:

To show equivalence we show that both sides act on the same initial state and produce the same output state.

$v_{in}(L_1) = v_{in}(L_2) = v_{in}(L_1 \oplus L_2) = v_{in}(L_1 + IT(L_2, L_1))$

Let S_1 denote the set of siteids appearing in L_1, and S_2 the set of siteids appearing in L_2. Since $L_1 \ || \ L_2$, $S_1 \cap S_2 = \varnothing$.

Assuming either L_1 and L_2 are non-empty, we can define $s_m = \min S_1 \cup S_2$ which is the minimum siteid that appears in either L_1 or L_2. Since $S_1 \cap S_2 = \varnothing$, s_m appears either in S_1 or S_2 but not both.

If $s_m \in S_1$ then we know that all operations in L_2 are disabled as they IT past L_1. Referring to the above figure we can consider the path from the bottom left corner to the top right along the left and top sides. This leads to the result that $(L_1 + IT(L_2,L_1)) \sim L_1 = L_1 \oplus L_2$.

Alternatively, if $s_m \in S_2$, then we deduce $L_1 + IT(L_2,L_1) \sim L_2 + IT(L_1,L_2) \sim L_2 = L_1 \oplus L_2$.

Claim: $L_1 \oplus L_2$ is maximal in its use of atomic operations compatible with the constraint that it be causally ordered and all operations in the sequence are enabled.

We now extend the definition of $\oplus$ to deal with context equivalent operations O_1,O_2 (ie satisfying $v_{in}(O_1) = v_{in}(O_2)$). Let $O_1 \oplus_C O_2 = Lf(O_1,v) + (Rf(O_1,v) \oplus_{||} Rf(O_2,v))$ where $v = v_{out}(O_1) \downarrow v_{out}(O_2)$.

Claim: $O_1 \oplus_C O_2 = O_2 \oplus_C O_1$

Proof: This is true if $Lf(O_1,v) = Lf(O_2,v)$, and we could only assume that with some kind of inductive argument!

16.2 Factorising an assignment operation

Note that

$$\aleph(Lf(O,v)) = \{ op(s,t) \in \aleph(O) \mid t < v(s) \}$$

$$\aleph(Rf(O,v)) = \{ op(s,t) \in \aleph(O) \mid t \geq v(s) \}$$

Factorisation of a list of assignment entries doesn't require transposing of entries because there is no pair O_1,O_2 that are concurrent ($O_1 \mid\mid O_2$).

Instead factorisation with respect to a causally valid vector time v simply involves splitting the list into prefix and suffix where assignment entries in the prefix satisfy $t < v(s)$, and in the suffix satisfy $t \geq v(s)$.

That this is always possible without need for transpose follows directly from the definition of a causally valid vector time.

v is causally valid so $O_1 \rightarrow O_2$ and $O_2 \in \chi(v) \Rightarrow O_1 \in \chi(v)$

so if O_1 appears before O_2 in the list, $O_2 \in \chi(v) \Rightarrow O_1 \in \chi(v)$

Note that since both an LFactor or RFactor of a list is a subsequence of the original, each preserves the causal ordering.

16.3 Merge of assignments

Let O_1,O_2 be two composite assignment operations on the same FieldId, and satisfying $v_{in}(O_1) \leq v$ and $v_{in}(O_2) \leq v$ where

$v = v_{out}(O_1) \downarrow v_{out}(O_2)$.

By definition $O_1 \oplus_{\mathbb{R}} O_2 = \sum[v_{in}(O_1), v_{out}(O_1) \uparrow v_{out}(O_2))$

Consider that we factorise O_1 with respect to v. From the above we have seen that this simply involves splitting its list of assignment entries into two parts without any need for transposition. Similarly we factorise O_2 with respect to v.

It was shown previously that $Rf(O_1,v) \,||\, Rf(O_2,v)$. Therefore given an assignment entry in $Rf(O_1,v)$ and an assignment entry in Rf(O2,v) we can assume they are concurrent. Therefore they were generated by different sites and the entry with smaller siteid is regarded as dominating the other.

We have previously treated the case of merging two concurrent lists.

Definition: $O_1 \oplus_{\mathbb{R}} O_2 = s_{min}(Rf(O_1,v)) < s_{min}(Rf(O_2,v)) \ ? \ O_1 \ : \ Lf(O_1,v) + Rf(O_2,v)$
where + denotes list concatenation.

Claim: O_1, O_2 causally ordered $\Rightarrow$ $O_1 \oplus_{\mathbb{R}} O_2$ is causally ordered.

Proof:

If $s_{min}(Rf(O_1,v)) < s_{min}(Rf(O_2,v))$ then $O_1 \oplus_{\mathbb{R}} O_2 = O_1$ which is causally ordered.

Otherwise $O_1 \oplus_{\mathbb{R}} O_2 = Lf(O_1,v) + Rf(O_2,v)$.

Hand waving: Since $\oplus$ is associative and commutative, there is a uniqueness of representation of O_1 and O_2. Therefore we expect some commonality in $Lf(O_1,v)$ and $Lf(O_2,v)$, and in fact $Lf(O_1,v) = Lf(O_2,v)$ if $v_{in}(O_1) = v_{in}(O_2)$. That suggests all operations in $Lf(O_1,v)$ causally precede operations in $Rf(O_2,v)$. TODO: formalise this!

Definition: Let $R(O_1,O_2) = Rf(O_1, v_{out}(O_1) \downarrow v_{out}(O_2))$

Definition: Let $S(O_1,O_2)$ be the set of siteids over all the assignment entries in $R(O_1,O_2)$.

We can assume $S(O_1,O_2) \cap S(O_2,O_1) = \varnothing$.

Definition: Let $F(O_1,O_2)$ be the subsequence of operations taken from $R(O_1,O_2)$ without those entries with siteid $s_1 \in S(O_1,O_2)$ where $\exists\, s_2 \in S(O_2,O_1)$ such that $s_2 < s_1$.

Since $F(O_1,O_2)$ is a subsequence of $R(O_1,O_2)$, we see that it preserves causal ordering.

Claim: $F(O_1,O_2) \neq \varnothing \Rightarrow F(O_2,O_1) = \varnothing$

Proof: Suppose $\exists$ op(s,t) in $F(O_1,O_2)$. Then $\forall\, s_2 \in S(O_2,O_1)$, $s < s_2$. But that implies that every entry in $R(O_2,O_1)$ is dominated by op(s,t), so $F(O_2,O_1) = \varnothing$.

Definition: $O_1 \oplus_{\mathbb{R}} O_2 = Lf(O_1,v) + F(O_1,O_2) + F(O_2,O_1)$
where + denotes list concatenation.

Claim: O_1, O_2 causally ordered $\Rightarrow$ $O_1 \oplus_R O_2$ is causally ordered.

Proof:

$Lf(O_1,v) + F(O_1,O_2)$ is a subsequence of O_1 so must be causally ordered. We also know that $F(O_2,O_1)$ is causally ordered.

It remains to show that whenever atomic operation O_x is in $Lf(O_1,v) + F(O_1,O_2)$, and atomic operation O_y is in $F(O_2,O_1)$ then $O_x \rightarrow O_y$.

If O_x in $Lf(O_1,v)$ then O_x also appears in $Lf(O_2,v)$ [TODO: Does it?]. Therefore $O_x \rightarrow O_y$ because $Lf(O_2,v) + F(O_2,O_1)$ is causally ordered.

If O_x in $F(O_1,O_2)$ then $F(O_2,O_1) = \varnothing$.

16.4 Further compression of assignment operations

Consider that a site never takes an LFactor of an operation, and it always sends an entire RFactor of the existing HB.

It would appear possible to achieve further compression of an assignment operation by only recording the value for the last assignment in the list, and by ignoring the relative order of the enabled assignments in the list.

It turns out that this is enough information to apply operations, take RFactors and to merge two operations. However, it doesn't allow for an LFactor to be calculated.

Definition: Let N be the set of non-negative integers. ie N = { 0,1,2,3,... }

Definition: Let $T \subseteq N$ and $n \in N$. Then we define

$$T_{left}(T,n) = \{ t \in T \mid t < n \} \subseteq T$$

$$T_{right}(T,n) = \{ t \in T \mid t \geq n \} \subseteq T$$

Note that $\forall n$, $T = T_{left}(T,n) \cup T_{right}(T,n)$.

Definition: for a given FieldId, O and s, let

$$t_{set}(O,s) = \{ t \mid op(s,t) \in \aleph(O) \} \subseteq N.$$

Claim: $t_{set}(Lf(O,v),s) = T_{left}(t_{set}(O,s), v(s))$

Proof:

$$\begin{aligned} & t_{set}(Lf(O,v),s) \\ = \; & \{ t \mid op(s,t) \in \aleph(Lf(O,v)) \} \\ = \; & \{ t \mid op(s,t) \in \aleph(O) \;\&\; t < v(s) \} \\ = \; & T_{left}(\{ t \mid op(s,t) \in \aleph(O) \}, v(s)) \\ = \; & T_{left}(t_{set}(O,s), v(s)) \end{aligned}$$

Similarly: $t_{set}(Rf(O,v),s) = T_{right}(t_{set}(O,s), v(s))$

For the given FieldId, let $v_{end}(O)$ be the vector time satisfying (for each siteid s),

$$v_{end}(O)(s) = ((t_{set}(O,s) = \varnothing) ? 0 : 1 + \max t_{set}(O,s))$$

This provides some information about what assignments have been performed on the field.

$s_{min}(Rf(O,v))$
$= \min \{ s \mid \exists\ op(s,t) \in \aleph(Rf(O,v)) \}$
$= \min \{ s \mid \exists\ op(s,t) \in \aleph(O) \text{ such that } t \geq v(s) \}$
$= \min \{ s \mid \exists\ t \in t_{set}(O,s) \text{ such that } t \geq v(s) \}$
$= \min \{ s \mid T_{right}(t_{set}(O,s),v(s)) \neq \varnothing \}$
$= \min \{ s \mid v_{end}(O)(s) > v(s) \}$

Definition: Let predicate $P_s(O_1,O_2) = (s_{min}(Rf(O_1,v)) < s_{min}(Rf(O_2,v)))$

Can we use $t_{set}(O_1,s)$ and $t_{set}(O_2,s)$, to calculate $t_{set}(O_1 \oplus_{\mathbb{R}} O_2)$?

$t_{set}(O_1 \oplus_{\mathbb{R}} O_2, s)$

$= t_{set}(P_s(O_1,O_2) ? O_1 : Lf(O_1,v) + Rf(O_2,v), s)$

$= P_s(O_1,O_2) ? t_{set}(O_1, s) : t_{set}(Lf(O_1,v) + Rf(O_2,v), s)$

$= P_s(O_1,O_2) ? t_{set}(O_1) : t_{set}(Lf(O_1,v),s) + t_{set}(Rf(O_2,v), s)$

$= P_s(O_1,O_2) ? t_{set}(O_1) : T_{left}(t_{set}(O_1,s), v(s)) + T_{right}(t_{set}(O_2,s), v(s))$

16.5 Implementation notes

A set of t values can be stored in increasing order in a vector. Binary search can be used to efficiently find a given t value, or to find the position to split the list into prefix and suffix according to v(s).

Assignable variables are often edited by dragging with the mouse, and therefore there can be many assignments per second (eg 50 Hz) by a given site. This makes some kind of run length encoding highly appropriate for representation of a set of t values, because it is quite likely there will be long runs of sequential t values in a range $[t_1,t_2)$.

We can store both individual t values as well as pairs $[t_1,t_2)$ in a single vector<int32> if we use the MSB to distinguish between an individual t values and the t_1 of a pair $[t_1,t_2)$.

With some effort, binary search is compatible with run length encoding!

16.6 Proposed test

It is proposed that a self contained unit test be written that simulates n sites that generate and exchange assignment operations on a single assignable field.

An operation records the assigned value, plus a tset indexed by siteid.

```
struct Op
{
    VectorTime m_v1,m_v2;
    int m_value;
    map<SiteId, set<int> > m_tsets;
};

void Merge(Op& O1, const Op& O2)
{
    VectorTime v = O1.v2 ↓  O2.v2;
     for each s
     {
          set<int>& ts = O1.m_tsets[s];
          ...
     }
}
```

The test uses a single thread and doesn't use any IPC. In fact we don't even model a connection or transient HB-suffixes. Instead we emulate a sending of an operation by picking a sender site s_1 and receiver site s_2 at random and merging hb_1 into hb_2. This involves calculating $v = v_{out}(hb_1) \downarrow v_{out}(hb_2)$.

16.7 Dom relation on vector times

Definition: Let s_0 denote a special site id value (used as a "marker" - and not the identifier for any real site) that compares less to all other site ids. For any given vector time v, we assume $v(s_o) = 0$.

Definition: Given vector times v_1,v_2 we define $s_m(v_1,v_2)$ as follows:

$$s_m(v_1,v_2) \quad = \quad (v_1 = v_2) \ ? \ s_0 : \ \min \{ s \mid v_1(s) \neq v_2(s) \ \}$$

Claim: $s_m(v_1,v_2) = s_m(v_2,v_1)$
Proof:

if $v_1 = v_2$

$$s_m(v_1,v_2) = s_0 = s_m(v_2,v_1)$$

else

$$\begin{aligned} s_m(v_1,v_2) &= \min \{ s \mid v_1(s) \neq v_2(s) \} \\ &= \min \{ s \mid v_2(s) \neq v_1(s) \} \\ &= s_m(v_2,v_1) \end{aligned}$$

Definition: We say v_1 dominates v_2, written as $dom(v_1,v_2)$ if

$$v_1(s_m(v_1,v_2)) \ > \ v_2(s_m(v_1,v_2))$$

Obviously $dom(v_1,v_2) \Rightarrow v_1 \neq v_2$
In other words dom is irreflexive. ie $\forall v, \neg dom(v,v)$

Claim: $dom(v_1,v_2) \Leftrightarrow \exists s'$ st $v_1(s') > v_2(s')$ and $\forall s < s', v_1(s) = v_2(s)$.
Proof:

($\Rightarrow$)
Suppose $dom(v_1,v_2)$

So $v_1 \neq v_2$ and $v_1(s_m(v_1,v_2)) > v_2(s_m(v_1,v_2))$
Let $s' = s_m(v_1,v_2) = \min \{ s \mid v_1(s) \neq v_2(s) \}$

$v_1(s_m(v_1,v_2)) > v_2(s_m(v_1,v_2))$
so $v_1(s') > v_2(s')$.

$s' = \min \{ s \mid v_1(s) \neq v_2(s) \}$
so $\forall s < s', v_1(s) = v_2(s)$

($\Leftarrow$)
Suppose $\exists s'$ st $v_1(s') > v_2(s')$ and $\forall s < s', v_1(s) = v_2(s)$
$v_1(s') > v_2(s')$ shows that $v_1 \neq v_2$
It also follows that $s_m(v_1,v_2) = s'$, therefore $v_1(s_m(v_1,v_2)) > v_2(s_m(v_1,v_2))$.
Hence $dom(v_1,v_2)$ is proven.

Claim: $dom(v_1,v_2) \Leftrightarrow \exists s'$ st $v_1(s') > v_2(s')$ and $\forall s < s', v_1(s) \geq v_2(s)$
Proof:
($\Rightarrow$)
$dom(v_1,v_2) \Rightarrow \exists s'$ st $v_1(s') > v_2(s')$ and $\forall s < s', v_1(s) = v_2(s)$
$\Rightarrow \exists s'$ st $v_1(s') > v_2(s')$ and $\forall s < s', v_1(s) \geq v_2(s)$
($\Leftarrow$)
Suppose $\exists s'$ st $v_1(s') > v_2(s')$ and $\forall s < s', v_1(s) \geq v_2(s)$
if $\forall s < s', v_1(s) = v_2(s)$
Then s' satisfies the criteria to deduce $dom(v_1,v_2)$
else
$\exists s'' < s'$ st $v_1(s'') > v_2(s'')$
In fact let s'' be the smallest siteid satisfying $v_1(s'') > v_2(s'')$
i.e. $\forall s < s'', v_1(s) = v_2(s)$
Then s'' satisfies the criteria to deduce $dom(v_1,v_2)$

Claim: $v_1 \leq v_2 \Rightarrow \neg dom(v_1, v_2)$
Proof: $v_1 \leq v_2 \Rightarrow \forall s, v_1(s) \leq v_2(s)$
$\Rightarrow \neg(\exists s$ st $v_1(s) > v_2(s))$
$\Rightarrow \neg(\exists s'$ st $v_1(s') > v_2(s')$ and $\forall s < s', v_1(s) = v_2(s))$
$\Rightarrow \neg dom(v_1, v_2)$

Claim: $dom(v_1,v_2) \Rightarrow \neg dom(v_2,v_1)$.
Proof:
$dom(v_1,v_2)$
$\Rightarrow$ $v_1 \neq v_2$ and $v_1(s_m(v_1,v_2)) > v_2(s_m(v_1,v_2))$
$\Rightarrow$ $\neg (v_2(s_m(v_1,v_2)) > v_1(s_m(v_1,v_2)))$
$\Rightarrow$ $\neg dom(v_2,v_1)$

The converse of this claim is false.

Claim: $(\neg dom(v_1,v_2) \wedge \neg dom(v_2,v_1)) \Leftrightarrow v_1 = v_2$
Proof:
($\Rightarrow$)
Suppose not,

i.e. $\neg dom(v_1,v_2) \wedge \neg dom(v_2,v_1) \wedge v_1 \neq v_2$
given $v_1 \neq v_2$ it follows that $\{ s \mid v_1(s) \neq v_2(s) \}$ is non empty.
Let $s' = s_m(v_1,v_2) = s_m(v_2,v_1) = \min \{ s \mid v_1(s) \neq v_2(s) \}$
It follows that $v_1(s') \neq v_2(s')$
$\neg dom(v_1,v_2) \wedge v_1 \neq v_2 \Rightarrow \neg(v_1(s') > v_2(s')) \Rightarrow v_1(s') \leq v_2(s')$
$\neg dom(v_2,v_1) \wedge v_1 \neq v_2 \Rightarrow \neg(v_2(s') > v_1(s')) \Rightarrow v_2(s') \leq v_1(s')$
$v_1(s') \leq v_2(s') \wedge v_2(s') \leq v_1(s') \Rightarrow v_1(s') = v_2(s')$ -- contradiction

($\Leftarrow$)
$dom(v_1,v_2) \Rightarrow v_1 \neq v_2$
By contrapositive, $v_1 = v_2 \Rightarrow \neg dom(v_1,v_2)$
By symmetry, $v_2 = v_1 \Rightarrow \neg dom(v_2,v_1)$
Therefore $v_1 = v_2 \Rightarrow (\neg dom(v_1,v_2) \wedge \neg dom(v_2,v_1))$

Claim: $(dom(v_1,v_2) \wedge dom(v_2,v_3)) \Rightarrow dom(v_1,v_3)$
Proof:
Let $s_{12} = s_m(v_1,v_2)$, $s_{23} = s_m(v_2,v_3)$.
$v_1(s_{12}) > v_2(s_{12})$ (because $dom(v_1,v_2)$)
$v_2(s_{23}) > v_3(s_{23})$ (because $dom(v_2,v_3)$)
$\forall s < s_{12}, v_1(s) = v_2(s)$ (because $dom(v_1,v_2)$)
$\forall s < s_{23}, v_2(s) = v_3(s)$ (because $dom(v_2,v_3)$)
Let $s' = \min(s_{12},s_{23})$
$\forall s < s', v_1(s) = v_2(s) = v_3(s)$
Claim: $v_1(s') > v_3(s')$
if $s_{12} = s_{23}$
$s' = s_{12} = s_{23}$
$v_1(s') > v_2(s') > v_3(s')$
else if $s_{12} < s_{23}$
$s' = s_{12}$
$v_1(s') > v_2(s') = v_3(s')$
else if $s_{23} < s_{12}$
$s' = s_{23}$
$v_1(s') = v_2(s') > v_3(s')$
$dom(v_1,v_3)$ (definition)

It follows that dom is a transitive irreflexive relation.

Claim: dom is trichotomous (i.e. exactly one of $dom(v_1,v_2)$, $dom(v_2,v_1)$, $v_1=v_2$ is true)
Proof: This follows from these claims which have already been proven:
$dom(v_1,v_2) \Rightarrow v_1 \neq v_2$
$dom(v_1,v_2) \Rightarrow \neg dom(v_2,v_1)$
$v_1=v_2 \Rightarrow \neg dom(v_1,v_2)$
$(\neg dom(v_1,v_2) \wedge \neg dom(v_2,v_1)) \Leftrightarrow v_1 = v_2$

It follows that the dom relation is a *strict total order* on the set of vector times.

Claim: $\neg dom(v_1, v_1 \uparrow v_2)$
Proof: $v_1 \leq v_1 \uparrow v_2$ (property of $\uparrow$)
so $\neg dom(v_1, v_1 \uparrow v_2)$ (by an earlier claim)

Claim: $dom(v_1,v_3) \Rightarrow dom(v_1 \uparrow v_2, v_3)$
Proof:

Suppose $dom(v_1,v_3)$
$\exists s'$ st $v_1(s') > v_3(s')$ and $\forall s < s'$, $v_1(s) = v_3(s)$

$v_1(s') > v_3(s')$
so $(v_1 \uparrow v_2)(s') = \max(v_1(s'), v_2(s')) \geq v_1(s) > v_3(s')$

$\forall s < s'$, $v_1(s) = v_3(s)$
so $\forall s < s'$, $(v_1 \uparrow v_2)(s) = \max(v_1(s), v_2(s)) \geq v_1(s) = v_3(s)$

$\exists s'$ st $(v_1 \uparrow v_2)(s') > v_3(s')$ and $\forall s < s'$, $(v_1 \uparrow v_2)(s) \geq v_3(s) \Rightarrow dom(v_1 \uparrow v_2, v_3)$

Claim: $dom(v_1, v_2 \uparrow v_3) \Rightarrow dom(v_1,v_2)$
Proof: Suppose $dom(v_1, v_2 \uparrow v_3)$
$\exists s'$ st $v_1(s') > (v_2 \uparrow v_3)(s')$ and $\forall s < s'$, $v_1(s) = (v_2 \uparrow v_3)(s)$
so $v_1(s') > (v_2 \uparrow v_3)(s') = \max(v_2(s'), v_3(s')) \geq v_2(s')$
also, $\forall s < s'$, $v_1(s) = (v_2 \uparrow v_3)(s) = \max(v_2(s), v_3(s)) \geq v_2(s)$.
Result follows from $\exists s'$ st $v_1(s') > v_2(s')$ and $\forall s < s'$, $v_1(s) \geq v_2(s) \Rightarrow dom(v_1,v_2)$

Claims:

a. $\neg(dom(v_3,v_1) \wedge dom(v_3,v_2) \Rightarrow dom(v_3, v_1 \uparrow v_2))$
b. $\neg(v_1 \leq v_3 \wedge dom(v_3,v_2) \Rightarrow dom(v_3, v_1 \uparrow v_2))$
c. $\neg(dom(v_1 \uparrow v_2, v_3) \Rightarrow (dom(v_1,v_3) \vee dom(v_2,v_3)))$

Proof:

For a counter example for all these erroneous implications,
let there be three sites and $v_3 = (1,1,1)$ dominates both $v_1 = (1,0,1)$ and $v_2 = (0,1,2)$
whereas $v_1 \uparrow v_2 = (1,1,2)$ dominates v_3.

16.8 Even better compression of assignment operations

Consider that we

1. Avoid any concept of factorisation of the assignment operations because assignment operations O are always assumed to have $v_{in}(O) = v_{\varnothing}$.

2. Don't store $t_{set}(O,s)$ for each site s, but rather only store $v_{end}(O)$.

An operation on a given assignable field (only) records the following

```
struct
{
   T value;
   VectorTime v_end;
};
```

where 'value' is the state of the field associated with the operation, and 'v_{end}' provides some information about what sites have performed an assignment on the field.

Definition: We say operations O_1,O_2 are equal (written $O_1 = O_2$) if both

1. O_1.value = O_2.value; and
2. $O_1.v_{end} = O_2.v_{end}$

Definition: We say O_1 dominates O_2, written as dom(O_1,O_2) if dom($O_1.v_{end}$,$O_2.v_{end}$)

Definition: Let X be a set of operations. We say that X *satisfies convergence* if

$$\forall O_1,O_2 \in X, \quad O_1.v_{end} = O_2.v_{end} \Rightarrow O_1.\text{value} = O_2.\text{value}$$

[In other words, $O_1.v_{end} = O_2.v_{end}$ is sufficient to deduce $O_1 = O_2$]

Claim: If X satisfies convergence then

$$\forall O_1,O_2 \in X, \quad (\neg\text{dom}(O_1,O_2) \wedge \neg\text{dom}(O_2,O_1)) \Leftrightarrow O_1 = O_2$$

Claim: If X satisfies convergence, the dom relation is a total ordering on X.

Definition: Given operations O_1,O_2, the merge $O_1 \oplus O_2$ is an operation defined as follows:

$$O_1 \oplus O_2 = \text{dom}(O_1,O_2) \ ? \ O_1 : O_2$$

Obviously $\forall O,\ O \oplus O = O$

Claim: $\neg\text{dom}(O_1, O_1 \oplus O_2)$
Proof: Suppose $\text{dom}(O_1, O_1 \oplus O_2)$
if dom (O_1,O_2)
$O_1 \oplus O_2 = \text{dom}(O_1,O_2) \ ? \ O_1 : O_2 = O_1$
$\text{dom}(O_1, O_1 \oplus O_2) \Rightarrow \text{dom}(O_1, O_1)$ contradiction
else
$O_1 \oplus O_2 = \text{dom}(O_1,O_2) \ ? \ O_1 : O_2 = O_2$
$\text{dom}(O_1, O_1 \oplus O_2) \Rightarrow \text{dom}(O_1, O_2)$ contradiction

Claim: If X satisfies convergence then $\forall O_1,O_2 \in X, \quad O_1 \oplus O_2 = O_2 \oplus O_1$
Proof: if $O_1 = O_2$
$\neg\text{dom}(O_1,O_2) \wedge \neg\text{dom}(O_2,O_1)$ (X satisfies convergence)

$$\begin{aligned} O_1 \oplus O_2 &= \text{dom}(O_1,O_2) \ ? \ O_1 : O_2 \\ &= O_2 \\ &= O_1 \\ &= \text{dom}(O_2,O_1) \ ? \ O_2 : O_1 \\ &= O_2 \oplus O_1 \end{aligned}$$

else
$O_1.v_{end} \neq O_2.v_{end}$ (X satisfies convergence)
$\text{dom}(O_1,O_2) \Leftrightarrow \neg\text{dom}(O_2,O_1)$

$$\begin{aligned} O_1 \oplus O_2 &= \text{dom}(O_1,O_2) \ ? \ O_1 : O_2 \\ &= \text{dom}(O_2,O_1) \ ? \ O_2 : O_1 \\ &= O_2 \oplus O_1 \end{aligned}$$

Claim: $O_1 \oplus (O_1 \oplus O_2) = O_1 \oplus O_2$ (absorption property)
Proof: $O_1 \oplus (O_1 \oplus O_2) = \text{dom}(O_1,O_1 \oplus O_2) \ ? \ O_1 : O_1 \oplus O_2$

$= \quad O_1 \oplus O_2 \quad (\neg dom(O_1, (O_1 \oplus O_2)))$

Claim: If X satisfies convergence then $\forall O_1,O_2,O_3 \in X, \quad (O_1 \oplus O_2) \oplus O_3 = O_1 \oplus (O_2 \oplus O_3)$

Proof:

if $O_1 = O_2$

$(O_1 \oplus O_2) \oplus O_3 = O_2 \oplus O_3$ $(O_1 \oplus O_2 = O_2)$

$= O_1 \oplus (O_2 \oplus O_3)$ (absorption)

else

if $O_2 = O_3$

$(O_1 \oplus O_2) \oplus O_3 = O_1 \oplus O_2$ (absorption)

$= O_1 \oplus (O_2 \oplus O_3)$ $(O_2 \oplus O_3 = O_2)$

else

$O_1.v_{end} \neq O_2.v_{end}$ (X satisfies convergence)

$dom(O_1,O_2) \Leftrightarrow \neg dom(O_2,O_1)$

$O_2.v_{end} \neq O_3.v_{end}$ (X satisfies convergence)

$dom(O_2,O_3) \Leftrightarrow \neg dom(O_3,O_2)$

if $dom(O_1,O_2)$

if $dom(O_2,O_3)$

$dom(O_1,O_3)$ (transitivity)

$(O_1 \oplus O_2) \oplus O_3 = O_1 \oplus O_3$ $(dom(O_1,O_2))$

$= O_1$ $(dom(O_1,O_3))$

$= O_1 \oplus O_2$ $(dom(O_1,O_2))$

$= O_1 \oplus (O_2 \oplus O_3)$ $(dom(O_2,O_3))$

else

$(O_1 \oplus O_2) \oplus O_3 = O_1 \oplus O_3$ $(dom(O_1,O_2))$

$= O_1 \oplus (O_2 \oplus O_3)$ $(\neg dom(O_2,O_3))$

else

if $dom(O_2,O_3)$

$(O_1 \oplus O_2) \oplus O_3 = O_2 \oplus O_3$ $(\neg dom(O_1,O_2))$

$= O_2$ $(dom(O_2,O_3))$

$= O_1 \oplus O_2$ $(\neg dom(O_1,O_2))$

$= O_1 \oplus (O_2 \oplus O_3)$ $(dom(O_2,O_3))$

else

$dom(O_2,O_1)$ $(O_1 \neq O_2, \neg dom(O_1,O_2))$

$dom(O_3,O_2)$ $(O_2 \neq O_3, \neg dom(O_2,O_3))$

$dom(O_3,O_1)$ (transitivity)

$(O_1 \oplus O_2) \oplus O_3 = O_2 \oplus O_3$ $(\neg dom(O_1,O_2))$

$= O_3$ $(\neg dom(O_2,O_3))$

$= O_1 \oplus O_3$ $(dom(O_3,O_1))$

$= O_1 \oplus (O_2 \oplus O_3)$ $(\neg dom(O_2,O_3))$

Proof of convergence at quiescence: This basically follows from the fact that the merge operator $\oplus$ is commutative and associative. In addition it is obvious from the definition that:

$O_1,O_2 \in X \quad \Rightarrow \quad (O_1 \oplus O_2) \in X$

This shows that the merge operator doesn't add any new operations to X that may upset its ability to satisfy convergence. So as long as new generated operations O that are added to X have a distinct v_{end}, then it follows that X will continue to satisfy convergence.

16.9 Only sending deltas

Most generally there will be many identifiable assignable fields, each with their own value and v_{end}.

For efficient bandwidth utilisation it is important to only send small deltas over time between a given pair of sites. Let rhv be an underestimate of what's present in the remote history buffer. Then the sender site can avoid sending the value and v_{end} for an assignable field if it is known that no assignment to the field has occurred outside of X(rhv).

If an assignment has occurred then we must send the entire v_{end}, or else introduce delta vector times for each v_{end} of each field. However that would force the receiver to record a large amount of transient session information, perhaps enough to exhaust memory resources. Therefore we won't try to minimise bandwidth consumption in this way at present.

16.9.1 Session records the set of assignable fields that have changed

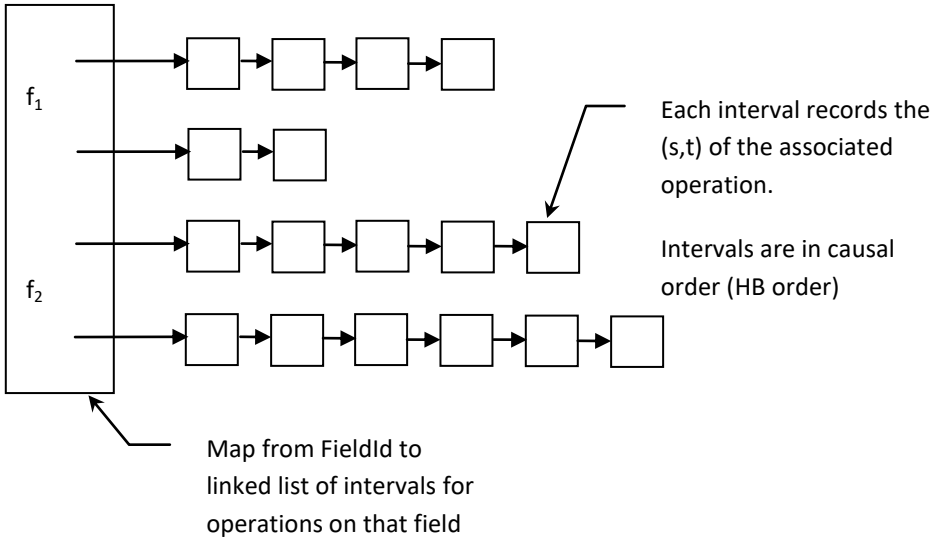
A transient session can record a set<FieldId> to keep track of the set of fields that have been assigned since the last operations was sent as part of the session. This set grows over time as local assignment operations are performed. Then, when a new operation needs to be sent it is a simple matter to retrieve the values and vend vector times for all these fields, and send them all as part of one universal operation.

However, this doesn't provide a solution to when a new session has been created, and therefore it is necessary to send all changes to all fields since the collaboration began. It would appear that we need to store a persistent set<FieldId> for the entire working set. This is only cleared after a check-in to the repository.

17 Vector<T> operations

17.1 Per field HB suffixes

We can achieve some benefit by recording the HB suffix on a per-field basis, as follows



When a remote operation is to be transformed against the HB suffix we deal with each field independently. For a given field we transpose and pop according to the execution context of the remote operation. Then we perform the dualIT against the contextually serialised sequence of operations for a given field in the normal way.

This approach can lead to a dramatic reduction in the CPU load when remote operations and/or local operations work are distributed over a number of different fields. Of course there is no benefit when there is substantial work on a single vector<T> field to be merged.

17.2 Merging a list of operations

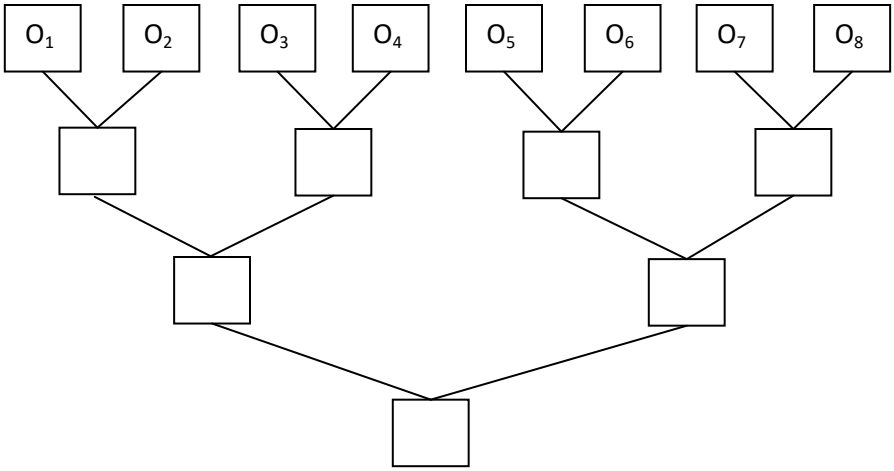
Let O_1,O_2 be contextually serialized. Let O_1 have n_1 insertion intervals and O_2 have n_2 insertion intervals on a given vector<T> field. The merge $O_1 \oplus O_2$ involves a linear scan through both sets of intervals. We estimate the cost as $O(n_1+n_2)$.

Consider a list L = [O_1, O_2, …, O_n] of n contextually serialized operations where each operation contains a single insertion interval into a given vector<T> field. Let this list be merged to become M = $\sum$L as follows

```
Operation M;  // M is initially empty
for each Oi in L
    M = M ⊕ Oi;
```

If none of the intervals can be coalesced then the merge results in n intervals, and yet the cost is (0+1)+(1+1)+(2+1)+(3+1)+…+(n-1+1) = n(n+1)/2 which is $O(n^2)$.

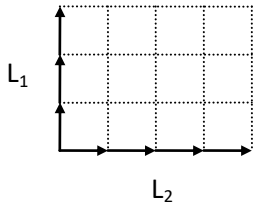
A better strategy is to tend towards merging roughly equal sized lists, in the manner of merge sort. For example:



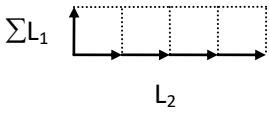
At each level of the tree the total number of insertion intervals to be processed is the same – ie O(n). The height of the tree is O(log n). Therefore the effort to merge the list is O(n log n).

17.3 Does merging improve performance of DualIT?

Let L_1, L_2 be lists of contextually serialised operations where each operation contains a single insertion interval. The dualIT transforms every operation in L_1 with every operation in L_2, so the algorithm is $O(|L_1||L_2|)$.

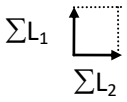


Consider that L_1 is merged into a single operation denoted by $\sum L_1$, and we are unlucky in that none of the intervals are adjacent (and managed to coalesce).



Then the dualIT of $\sum L_1$ and L_2 is still $O(|L_1||L_2|)$ - ie merging only one of the lists hasn't eliminated the quadratic complexity!

Now consider that we dualIT after merging both lists into single operations:



Now the cost is O((|L_1|+ |L_2|) – a great improvement.

17.4 Efficient R-factorization

Let M be a composite operation and v be a causally valid vector time satisfying

$$v_{in}(M) \leq v \leq v_{out}(M).$$

We want to efficiently calculate the R-factor of M with respect to v, which is denoted by

$$Rf(M,v) = \sum[v,v_{out}(M))$$

First let's limit ourselves to insertion operations on a single vector<T> field. Then M records a list of insertion elements (which we henceforth call "insertion intervals") ordered by q position. Note that the q positions are in *post insertion* coordinates. Each interval records

- the (s,t) associated with the original operation that generated the insertion of this string
- the q position
- the string being inserted at this q position

Claim: Rf(M,v) is obtained from M by simply removing every insertion interval with (s,t) satisfying t < v(s). There is no need to split or merge intervals, and no need to adjust q-positions!

Justification for not adjusting q positions: The intervals of Rf(M,v) are expressed in post creation intervals which corresponds to the state associated with $v_{out}(Rf(M,v))$. The intervals of M are expressed in the state associated with $v_{out}(M)$. But $v_{out}(Rf(M,v)) = v_{out}(M)$ so q positions don't require adjustment.

Now consider Create+Delete operations on a vector<T> field. The deletion intervals are ordered by q-position and also expressed in post-creation coordinates relative to $v_{out}(M)$. Therefore, again we find there is no need to adjust q positions when we remove deletion intervals with (s,t) satisfying t < v(s).

17.5 Efficient L-factorization

Let M be a composite operation and v be a causally valid vector time satisfying

$$v_{in}(M) \leq v \leq v_{out}(M).$$

We want to efficiently calculate the Lfactor of M with respect to v, which is denoted by

$$Lf(M,v) = \sum[v_{in}(M),v)$$

Claim: The insertion intervals of Lf(M,v) are found by

1. Removing intervals with (s,t) satisfying t ≥ v(s)

2. Shifting remaining q-positions to the left according to the total size of all intervals to the left that have been removed up to that point.

This suggests that the algorithm will use a left to right scan, calculating a negative offset to be applied to q-positions.

17.6 Merging the transient HB suffix

It is only worth merging the HB suffix into a single operation if the remote operations tend to have multiple insertion intervals, or else batches of remote operations are merged in order to avoid dualIT against lists of operations where each operation only contains about one insertion interval.

Compressing a batch of remote operations into a single operation seems like a good idea, although it is not clear how to recover the individual remote operations that were processed in a batch as a single merged UniOperation.

17.7 The equivalent of transpose and pop on a merged HB suffix

We need to be able to evolve the HB suffix over time (ie to remove operations from the merged suffix).

Consider that a remote operation Or is received with execution context v. In other words, the input state on which Or is meant to be executed is the same state that arises when applying exactly the operations in $\chi(v)$. Note therefore that v must be causally valid vector time v satisfying v ≤ hv, where hv denotes the vector time defining the current content of the entire HB.

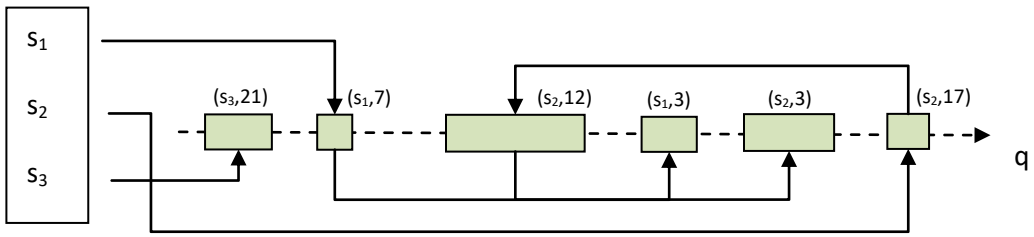
The equivalent of a transpose and pop simply involves the calculation of the R-factor of the HB-suffix with respect to the execution context of the next remote operation Or to be transformed against the HB-suffix.

The efficient calculation of an R-factor has been described above.

17.7.1 t-chains

Consider that for each site id, insertion intervals for a given vector<T> field are backward chained by decreasing t coordinate, independently of their ordering by q position (which is unrelated). For the given vector<T> field, a map keyed by siteid provides a pointer to the interval for that siteid with the largest t coord. Each interval stores a pointer to the interval with the next smallest t coord, or a NULL pointer to terminate the chain.

Backward chaining example



Map keyed by siteid

Intervals on a given vector<T> field depicted from left to right in order of q position

A given interval has uniquely defined (s,t), and therefore takes part in exactly one chain. The data structure for the interval can store a pointer to the next interval in the chain.

As additional intervals are merged into the operation, it is always the case that the new intervals have the largest t for given s, and therefore can be added to the chain without any need to scan through the intervals.

It is in fact possible to either forward chain or backward chain or both. To support forward chaining the map should additionally record a pointer to the interval with the *smallest* t coord for each siteid.

There can be many different intervals on the same vector<T> field with the same (s,t). For example an operation can insert at multiple places when it is first generated, or intervals can split under IT. We only require that the chain monotone increase with respect to t-coordinate. [Note that monotone increase means non-decreasing].

Forward chaining allows for an efficient means of iterating through all intervals inside $\chi(v)$ for a given v. ie to find all intervals satisfying i.t < v(i.s).

Backward chaining allows for an efficient means of iterating through all intervals outside $\chi(v)$ for a given v. ie to find all intervals satisfying i.t ≥ v(i.s).

17.7.2 t-chains across many fields

When there are many fields it can be very expensive to find all intervals inside $\chi(v)$ or outside $\chi(v)$, even with forward or backward chaining. The problem is that in practice a large proportion of the fields may have no intervals that are applicable to the result set.

Rather than store a map for each vector<T> field, consider instead that there is a single map for the entire UniOperation defined as `map<SiteId,set<Element>>` where `Element` is defined as follows

```
struct Element
{
    FieldId fid;
    Interval* first;
```

```
        Interval* last;
    };
```

A given element records the pointers to the first and last intervals in a t-chain for a given vector<T> field. Elements are only added to the set when the chain is non-empty. Therefore these pointers are never NULL.

An even more efficient approach is to chain across all vector<T> fields. However that requires the intervals to be self-describing (in the sense of being able to retrieve their associated FieldId). This can be achieved by making all intervals for a given FieldId hold a pointer to a variable that records the FieldId.

17.7.3 Potential approach: Recording entire HB as a single UniOperation

Can the whole HB be recorded as a single UniOperation, in such a way that it supports the requirements of sending local operations as well as transforming and accumulating remote operations?

Note that for the entire HB, the q positions are stored in post-insertion coordinates, and do not require adjustment as we consider session specific HB suffixes (that depend on the remote operation's execution context). Any given vector time v allows us to immediately see the entire HB as separated into prefix and suffix simply by testing whether each interval is inside $\chi(v)$. This leads to the idea of implementing a version of dualIT that is passed the vector time v so that it implicitly only transforms against the appropriate suffix of the HB.

The question then arises as to how to support efficient sending of operations, and how to avoid complete scans over all intervals in the HB all the time. Both of these seem related to the idea to filter the intervals based on a vector time.

Chaining of intervals by t coordinate would appear to be useful here, but it is not clear exactly how it would work. Chaining helps to define the subset of the intervals that are applicable. However to perform a dual IT they must be processed in order of q position.

Any clever indexing system would probably have a detrimental effect on the write performance of the system – because there is more persistent state that needs to be written to disk. Therefore this proposal is rejected.

NOTE: It is possible that this is the only approach that can avoid the quadratic complexity, in which case it trumps any I/O considerations!

17.7.4 Can an operation fit into memory?

Not if we follow this approach! An interactive session may involve operations that add huge amounts of data – including images and large amounts of text copied from the clipboard. After merging a single UniOperation may become too large to fit into memory. We therefore expect it to be broken up effectively using prefs somehow. This means that only parts of it need be marked as dirty as it changes.

We assume that LRU memory caching solves the problem of I/O performance.

17.7.5 Efficient representation of operations

Currently a UniOperation is stored as a single object in the PersistStore. Instead it is proposed that the maps keyed by FieldId are based on persistent B+Trees. This allows an operation to apply changes to millions of fields, yet indexing by FieldId is efficient – which is needed for OT.

Note that there can be substantial efficiency gains to B+Trees that are optimized to deal with keys that tend to have large common prefixes. There is a concept of sharing prefixes in the tree structure, reducing the storage requirements and reducing the time for key comparisons. These techniques are described in the literature on B+Trees.

17.7.6 On the fly compression of the HB

An interactive session could conceivably go on for days, with a dozen users, generating hundreds of thousands of operations. Compression of the HB is important for space efficiency and for allowing latecomers to enter the interactive session efficiently. It is also important for supporting updates from the ceda repository.

Note that lossy compression is quite different from merging of operations. The latter may have very little impact on the size taken up by the operations. By contrast compression throws away historical information about who performed the edits and when. It is particularly significant for assignment operations (where compression simply records the one and only dominating assignment on a given field). Note as well that insertions that were subsequently deleted are removed during compression.

During an interactive session there is a concept of a vector time v_c broadcast to all parties, indicating the set of operations that can be compressed. It is assumed that these operations will no longer take part in OT during the session.

Consider that the HB is recorded in two parts. A HB-prefix which is compressed, and a HB-suffix which is merged but uncompressed. It is assumed that the uncompressed part is small enough to fit into memory. Therefore algorithms that depend on efficient random access are possible.

It is assumed that any remote operation Or received by the site has an execution context that contains the HB-prefix. Therefore there is no need to IT any part of the HB-prefix past Or.

Each time v_c advances, it is necessary to remove intervals i from HB-suffix satisfying $i.t < vc(i.s)$ and add them into HB-prefix. The removal requires negative offsets to be applied to the intervals to exclude the insertions by the remaining operations (that are outside $\chi(vc)$) in the HB-suffix.

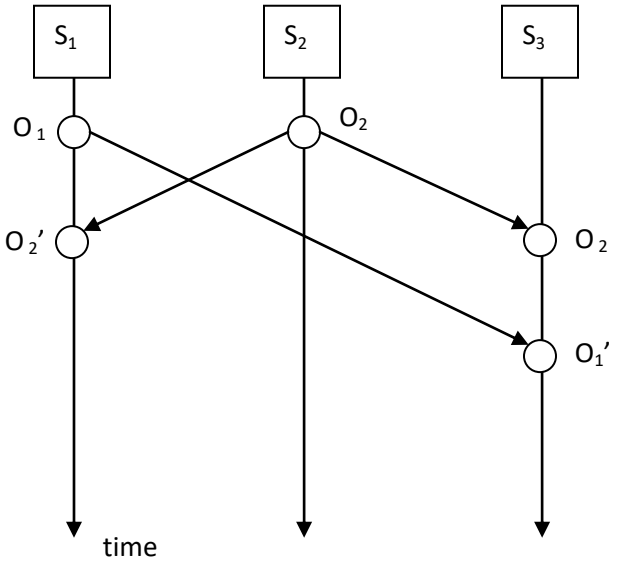
17.7.7 Sending operations

The conventional approach is to send operations in the tail of the linear HB that are not currently present on the remote site. Instead we need an effective mechanism that works directly with the persistent HB-prefix and HB-suffix.

A latecomer won't even have the HB-prefix (or its HB-prefix is too far out of date). In that case the first step is to transmit the entire HB-prefix. Fortunately it is compressed so that should be reasonably efficient. However how do we deal with an extended shared read lock? It would seem useful to use MVCC!

There is also the need to send all the operations in the HB-suffix that are not present on the remote site. This is easiest if we again have access to a fixed snapshot of the HB-suffix, giving us time to send everything in $\chi(hv)\backslash\chi(v)$ where vector time v is the intersection of hv with our best estimate of what's currently present on the remote site. Note that trying to send less than that (ie $\chi(v')\backslash\chi(v)$ for some vector time $v' < hv$) introduces the need to ET operations before they are sent! This seems a backward step because the remote site would eventually need to IT them again.

Note that the merging efforts by one site may contribute greatly to a reduction in merging effort by other sites – in a far superior fashion to an approach where linear HB's are sent. Consider the following example:



Conventionally S_3 would have to IT O_1 past O_2 when it receives O_1 from S_1. However if S_1 happens to sends O_1 after receiving O_2' then it is able to send it in the form O_1' – ie in the context of having performed O_2, eliminating the need for transformation on S_3!

17.7.7.1 Finding the operations to be sent

We always send a *single* UniOperation that represents everything in $\chi(hv)\backslash\chi(v)$ for an appropriate vector time v. Note therefore that the sender naturally sends batches of operations merged into single operations. This is exactly what's needed to avoid the quadratic complexity problem.

The UniOperation is obtained from the HB-Suffix simply by filtering intervals with given (s,t) depending on whether $t < v(s)$. This can be done efficiently using the backward chains. ie for a given field we have a map keyed by SiteId that points at the interval with largest t. We following the backward chain, decrementing t while $t < v(s)$. This is very efficient.

To avoid holding a shared read lock for too long we make a copy of the UniOperation to be sent in memory. This is subsequently sent in the background without a lock on the containing PSpace.

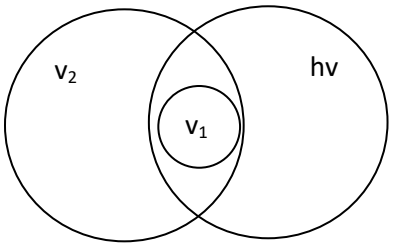
17.7.8 Transforming a received batch of operations

Let Or represent the merge of a batch of received operations. The set of operations in Or is given by $\chi(v_2)\backslash\chi(v_1)$ for two given vector times v_1,v_2 satisfying $v_1 \le v_2$. v_1 represents the execution context of Or.

The sender should have ensured that nothing in the execution context of Or is missing on the receiver. In other words v_1 is an underestimate of hv

$$v_1 \le hv$$

The set containment relationships can be depicted in the following Venn diagram:



The receiver of Or may find that some of these operations are already present. ie

$$\chi(hv) \cap (\chi(v_2)\backslash\chi(v_1)) \;=\; (\chi(hv) \cap \chi(v_2)) \setminus \chi(v_1) \neq \varnothing$$

Therefore it is possible that only a subset of Or should be applied. This subset is characterized as

$$\chi(v_2) \setminus \chi(hv)$$

Consider that we pretend we cannot see any of the intervals in Or inside $\chi(hv)$, and at the same time pretend that the execution context of Or is $hv \cap v_2$ instead of v_1. Since intervals in Or are expressed in post insertion coords in the context of all of $\chi(v_2)$ we find that there is no change to the q positions when we pretend the execution context is larger.

So pretending that Or is in fact representing the set of operations in $\chi(v_2) \setminus \chi(hv)$ and having execution context $hv \cap v_2$ makes it clear that it is straightforward to merge Or into the HB.

However is it efficient? We seem to need to transform against the entire HB-suffix which could be expensive. In addition we seem to need to test every interval for whether it's inside a vector time extent, and that can be substantially more expensive than the actual IT. In fact in the worse case $\chi(hv)\backslash\chi(v_2) = \varnothing$ so there should be nothing to do, and yet we iterate through every interval in the HB-suffix.

This is quite nasty because it means there is a significant CPU load even though a given pair of sites are in sync with each other.

It's difficult to see how to make this efficient without introducing the concept of a per-session transient HB-suffix.

17.7.9 Using both persistent as well as transient HB suffixes

Consider that we don't store a linear HB, but instead persist one HB-prefix and HB-suffix for the site (ie working set), and in addition each session with a given peer stores a single transient HB-suffix.

On a given site, for each session there is a dedicated thread to process incoming operations and another thread to send outgoing operations. Consider that these share the same transient HB-suffix. Note therefore that there tends to be contention for the transient HB-suffix, but not for the persistent HB in the PSpace.

It can be assumed that a transient HB-suffix always contains a subset of the operations in a persistent HB-suffix.

A transient HB suffix works as follows

- It is initially calculated from the persistent HB-suffix for a given vector time which represents an underestimate of the remote site's hv. Backward chaining in the persistent HB-suffix is very useful to make this efficient.
- When a merged batch Or is received, intervals in the transient HB-suffix that are inside the execution context of Or are removed. This can be done efficiently using forward chaining on the transient HB-suffix.
- Applying a merged batch of remote operations Or involves 1) removal of intervals within Or that are already present in the local HB, then 2) merging in the normal way without needing to test whether intervals are contained inside vector time extents.
- The session records a vector time rhv that indicates the set of operations that are known to have already been sent through the message queue to the remote site. The sender thread uses backward chaining to quickly find the operations in the HB suffix that are outside χ(rhv), and therefore need to be sent to the remote computer. rhv is then updated to ensure the same intervals are never sent again.

17.7.10 Incremental compression

Definition: Given operation O, let C(O) denote the result of compressing operation O. This has the effect of

- Removing all (s,t) related information
- Coalescing intervals where possible
- Removing redundancy such as insertions that were subsequently deleted, or redundant move operations.

Note that removing redundancy will impact the correctness of the PtoQ maps. Therefore for now we will ignore this complicating issue and disallow redundancy removal.

Let O_1, O_2 be contextually serialized operations. An incremental compression algorithm concerns the calculation of C($[O_1O_2]$) from C(O_1) and O_2. We could break this into two parts

- From O_2 calculate C(O_2)
- From C(O_1), C(O_2) calculate C($[O_1O_2]$)

17.7.10.1 Compression of a single UniOperation

Let's consider the special case of pure insertion operations on a single vector<T> field. We can scan left to right removing the (s,t) information. This may allow for many of the intervals to coalesce.

For a Create-Delete operation, we can similarly coalesce adjacent delete intervals. Furthermore we could also introduce the concept of a separate list of intervals that represent creation of deleted characters! These come from the intersection of the creation and deletion intervals. In theory this could be used to back-out redundant changes to the PtoQ maps.

Note that factoring out the intersection of creations and deletions should make it even more likely that we can subsequently coalesce creation and deletion intervals. ie it may be quite important to achieve decent compression.

17.7.10.2 Merging of compressed operations

Let C(O_1) and C(O_2) be contextually serialized, compressed operations. We want to calculate C([O_1 O_2]).

Given [C_1 D_1 C_2 D_2] we can easily transpose the middle two to give [C_1 C_2 D_1' D_2]. It is then straightforward to merge C_1 and C_2. C_1 insertions positions must be shifted to the right to include the effect of insertions by C_2. In addition adjacent intervals are coalesced.

Of course these algorithms have all been written before. However there are some small differences – such as the idea to ignore (s,t) when looking for opportunities to coalesce intervals. We can actually get code reuse if we first apply the same (s,t) to all intervals.

17.7.11 Transient HB suffix

A linear HB that only grows at the end is well suited for high transaction throughput – because new operations only need to be appended to the end, and this is well suited to a Log Structured Store (LSS). Furthermore, sending of operations basically involves asynchronously playing the tail of the HB which tends to be cached in memory and in any case will exhibit good clustering properties on disk – because the HB is read in the same order it is written.

Therefore we prefer the technique of a *transient* representation of the HB suffix – probably one for each session.

17.7.12 Potential approach: sharing a transient HB suffix across all sessions

If we share the same HB suffix across all sessions then we need a version of dualIT that works with a given vector time. However it must be remembered that the test i.t < v(i.s) is comparatively expensive (ie compared to the interval comparisons performed by IT). In practice when the system is under heavy load we expect that operations will be sent in batches with slowly changing

execution contexts. Therefore it can be quite useful to perform the filtering as rarely as possible. That means don't try to share a transient HB suffix across multiple sessions – ie this proposal is rejected.

18 Bogus solution for assignment

18.1 Rfactor of vector times

Definition : For vector times v_1, v_2, let $v_2 \backslash v_1$ be the vector time satisfying for each s,

$$(v_2 \backslash v_1)(s) \;=\; (v_2(s) > v_1(s)) \;?\; v_2(s) : 0$$

Claim: $v_1 \uparrow (v_2 \backslash v_1) \;=\; v_1 \uparrow v_2$

Proof: Let s be given

if $v_2(s) > v_1(s)$

$(v_1 \uparrow (v_2 \backslash v_1))(s)$

$= \max(v_1(s), (v_2 \backslash v_1)(s))$

$= \max(v_1(s), (v_2(s) > v_1(s)) \;?\; v_2(s) : 0)$

$= \max(v_1(s), v_2(s))$

$= (v_1 \uparrow v_2)(s)$

else

$(v_1 \uparrow (v_2 \backslash v_1))(s)$

$= \max(v_1(s), (v_2 \backslash v_1)(s))$

$= \max(v_1(s), (v_2(s) > v_1(s)) \;?\; v_2(s) : 0)$

$= \max(v_1(s), 0)$

$= \max(v_1(s), v_2(s))$ (because $v_1(s) \geq v_2(s)$)

$= (v_1 \uparrow v_2)(s)$

Claim: $v_1 \leq v_2 \;\Rightarrow\; v_1 \uparrow (v_2 \backslash v_1) = v_2$

Proof: Suppose $v_1 \leq v_2$

$v_1 \uparrow (v_2 \backslash v_1) = v_1 \uparrow v_2$ (previous claim)

$= v_2$ ($v_1 \leq v_2 \;\Rightarrow\; v_1 \uparrow v_2 = v_2$)

Claim: $v \leq v_1 \;\Rightarrow\; v_1 \uparrow v_2 = v_1 \uparrow (v_2 \backslash v)$

Proof:

Suppose $v \leq v_1$

Let s be given. We want to show $(v_1 \uparrow v_2)(s) = (v_1 \uparrow (v_2 \backslash v))(s)$

$v(s) \leq v_1(s)$ (because $v \leq v_1$)

if $v_1(s) < v_2(s)$

$v(s) \leq v_1(s) < v_2(s)$

so $(v_2 \backslash v)(s) = (v_2(s) > v(s) \;?\; v_2(s) : 0) = v_2(s)$

$(v_1 \uparrow v_2)(s)$

$= \max(v_1(s), v_2(s))$

$= \max(v_1(s), (v_2 \backslash v)(s))$ (because $v_2(s) = (v_2 \backslash v)(s)$)

$= (v_1 \uparrow (v_2 \backslash v))(s)$

else

$v_1(s) \geq v_2(s)$

$(v_1 \uparrow v_2)(s)$

$= \max(v_1(s), v_2(s))$

$= \quad v_1(s)$
$= \quad \max(v_1(s), 0)$
$= \quad \max(v_1(s), v_2(s) > v(s) \ ? \ v_2(s) : 0)$
$= \quad \max(v_1(s), (v_2 \backslash v)(s))$
$= \quad (v_1 \uparrow (v_2 \backslash v))(s)$

Definition: $\text{domf}(v_1, v_2, v) = \text{dom}(v_1, v_2 \uparrow v)$

Claim: $\text{domf}(v_1, v_2 \backslash v, v) \Rightarrow \text{dom}(v_1, v_2)$
Proof: $\text{domf}(v_1, v_2 \backslash v, v) = \text{dom}(v_1, (v_2 \backslash v) \uparrow v)$
$= \text{dom}(v_1, v_2 \uparrow v)$
$\Rightarrow \text{dom}(v_1, v_2)$

Conjecture: $v \leq v_1$ and $\text{dom}(v_1, v_2) \Rightarrow \text{domf}(v_1, v_2 \backslash v, v)$
Proof: ?

We need a specialised version of the merge operator! Actually a different definition of dom(), that accounts for RFactors. We don't want v_3 to appear to dominate $(v_2 \backslash v_1)$, just because $(v_2 \backslash v_1)$ has recorded some zeros. We only consider $v_3(s) > (v_2 \backslash v_1)(s)$ to be significant if $v_3(s) > v_1(s)$ - which means than v_3 has actually performed an assignment outside of $X(v_1)$ that beats v_2. Note that $(v_2 \backslash v_1)(s) > v_3(s)$ is always significant.

i.e. if $(v_2 \backslash v_1)(s) > v_3(s)$ then v_2 dominates v_3 for that s
if $v_3(s) > (v_2 \backslash v_1)(s)$ and $v_3(s) > v_1(s)$ then v_3 dominates v_2 for that s
otherwise neither dominates the other for that s.

We choose which operation dominates the other by finding the smallest s for which one dominates the other.

Note: we can use $v_1 \uparrow (v_2 \backslash v_1)$ instead of $(v_2 \backslash v_1)$, and go back to the conventional definition of dom!

18.2 Definition of merge

Definition: Given operations O_1, O_2, the merge $O_1 \oplus O_2$ is an operation defined as follows:

$$(O_1 \oplus O_2).v_{end} = O_1.v_{end} \uparrow O_2.v_{end}$$

$$(O_1 \oplus O_2).value = \text{dom}(O_1, O_2) \ ? \ O_1.value \ : \ O_2.value$$

Claim: $O_1 = O_2 \Rightarrow O_1 \oplus O_2 = O_1 = O_2$

Claim: $\neg\text{dom}(O_1, O_1 \oplus O_2)$
Proof: Follows from
$$O_1.v_{end} \leq (O_1.v_{end} \uparrow O_2.v_{end}) = (O_1 \oplus O_2).v_{end}$$

Claim: If X satisfies convergence then $\forall O_1, O_2 \in X, \quad O_1 \oplus O_2 = O_2 \oplus O_1$
Proof: $(O_1 \oplus O_2).v_{end} = O_1.v_{end} \uparrow O_2.v_{end}$

$= \quad O_2.v_{end} \uparrow O_1.v_{end}$ ($\uparrow$ commutes)

$= \quad (O_2 \oplus O_1).v_{end}$

Let $s' = s_m(O_1,O_2) = s_m(O_2,O_1)$

if $O_1 = O_2$

$\neg dom(O_1,O_2) \wedge \neg dom(O_2,O_1)$ (X satisfies convergence)

$(O_1 \oplus O_2).value = O_2.value$ ($\neg dom(O_1,O_2)$)

$= O_1.value$ ($O_1 = O_2$)

$= (O_2 \oplus O_1).value$ ($\neg dom(O_2,O_1)$)

else

$O_1.v_{end} \neq O_2.v_{end}$ (X satisfies convergence)

$dom(O_1,O_2) \Leftrightarrow \neg dom(O_2,O_1)$

$(O_1 \oplus O_2).value = dom(O_1,O_2) \ ? \ O_1.value \ : \ O_2.value$

$= dom(O_2,O_1) \ ? \ O_2.value \ : \ O_1.value$

$= (O_2 \oplus O_1).value$

Claim: $O_1 \oplus (O_1 \oplus O_2) = O_1 \oplus O_2$ (absorption property)

Proof: $(O_1 \oplus (O_1 \oplus O_2)).v_{end} = O_1.v_{end} \uparrow (O_1 \oplus O_2).v_{end}$

$= O_1.v_{end} \uparrow (O_1.v_{end} \uparrow O_2.v_{end})$

$= O_1.v_{end} \uparrow O_2.v_{end}$ (absorption of $\uparrow$)

$= (O_1 \oplus O_2).v_{end}$

$(O_1 \oplus (O_1 \oplus O_2)).value = dom(O_1, (O_1 \oplus O_2)) \ ? \ O_1.value \ : \ (O_1 \oplus O_2).value$

$= (O_1 \oplus O_2).value$ ($\neg dom(O_1, (O_1 \oplus O_2))$)

Claim: If X satisfies convergence then $\forall O_1,O_2,O_3 \in X$, $(O_1 \oplus O_2) \oplus O_3 = O_1 \oplus (O_2 \oplus O_3)$

Proof:

$((O_1 \oplus O_2) \oplus O_3).v_{end}$

$= \quad (O_1 \oplus O_2).v_{end} \uparrow O_3.v_{end}$

$= \quad (O_1.v_{end} \uparrow O_2.v_{end}) \uparrow O_3.v_{end}$

$= \quad O_1.v_{end} \uparrow (O_2.v_{end} \uparrow O_3.v_{end})$ ($\uparrow$ is associative)

$= \quad O_1.v_{end} \uparrow (O_2 \oplus O_3).v_{end}$

$= \quad (O_1 \oplus (O_2 \oplus O_3)).v_{end}$

if $O_1 = O_2$

$O_1 \oplus O_2 = O_2$ ($O_1 = O_2$)

if $O_2 = O_3$

$O_1 = O_2 \oplus O_3$ ($O_1 = O_2 = O_3$)

$O_2 \oplus O_3 = O_1 \oplus (O_2 \oplus O_3)$ ($O_1 = O_2 \oplus O_3$)

$((O_1 \oplus O_2) \oplus O_3).value = (O_2 \oplus O_3).value$ ($O_1 \oplus O_2 = O_2$)

$= (O_1 \oplus (O_2 \oplus O_3)).value$ ($O_1 = O_2 \oplus O_3$)

else

$((O_1 \oplus O_2) \oplus O_3).value = (O_2 \oplus O_3).value$ ($O_1 \oplus O_2 = O_2$)

$= (O_1 \oplus (O_2 \oplus O_3)).value$ (absorption)

else

if $O_2 = O_3$

todo

else

$((O_1 \oplus O_2) \oplus O_3).value$

$= dom(O_1 \oplus O_2,O_3) \ ? \ (O_1 \oplus O_2).value \ : \ O_3.value$

$= dom(O_1 \oplus O_2,O_3) \ ? \ (dom(O_1,O_2) \ ? \ O_1.value \ : \ O_2.value) \ : \ O_3.value$

= ...
= $(O_1 \oplus (O_2 \oplus O_3))$.value
todo

There is a concept of an overall winner that assigns its value in the merge. The merge is associative because the one and only winner always wins in the end, no matter what order we take, because the dom relation is asymmetric and transitive.

Definition: Let X be a set of assignment operations on a given field. The merge closure of X is denoted by cl(X) and is the smallest set satisfying both:

1. $X \subseteq cl(X)$; and
2. $O_1,O_2 \in cl(X) \Rightarrow O_1 \oplus O_2 \in cl(X)$

Claim: X satisfies convergence $\Rightarrow$ cl(X) satisfies convergence
Proof: Need to show $\forall O_1,O_2 \in cl(X)$, $O_1.v_{end} = O_2.v_{end} \Rightarrow O_1.value = O_2.value$
Let $O_1,O_2 \in cl(X)$ and $O_1.v_{end} = O_2.v_{end}$. We will show $O_1.value = O_2.value$.
Proof is by induction.
If $O_1,O_2 \in X$ then result follows from assumption that X satisfies convergence.
todo

We are now set for a proof of convergence at quiescence using an inductive proof. We need to formalise the control algorithm which is based on the merge operator. todo

Definition: O' = Rf(O,v) is the operation satisfying
1. O'.value = O.value
2. for each s, $O'.v_{end}(s) = O.v_{end}(s) > v(s)\ ?\ O.v_{end}(s) : 0$

[Note: If v(s) = t+1, then op(s,t) is the last op in X(v). If op(s,t) is in O, then $O.v_{end}(s) = t+1 = v(s)$. So need $O.v_{end}(s) > v(s)$ in order for there to be an operation in O not in X(v)]

Definition: Given operations O_1,O_2, the merge $O_1 \oplus O_2$ is an operation defined as follows:

$(O_1 \oplus O_2).v_{end} = O_1.v_{end} \uparrow O_2.v_{end}$

$(O_1 \oplus O_2).value = dom(O_1,O_2)\ ?\ O_1.value : O_2.value$

We need a specialised version of the merge operator! Actually a different definition of dom(), that accounts for RFactors. We don't want O_1 to appear to dominate $Rf(O_2,v)$, just because $Rf(O_2,v)$ has recorded some zeros in its v_{end}. We only consider $O_1.v_{end}(s) > Rf(O_2,v).v_{end}(s)$ to be significant if $O_1.v_{end}(s) > v(s)$ - which means than O_1 has actually performed an assignment outside of X(v) that beats O_2. Note that $Rf(O_2,v).v_{end}(s) > O_1.v_{end}(s)$ is always significant.

i.e. if $Rf(O_2,v).v_{end}(s) > O_1.v_{end}(s)$ then O_2 dominates O_1 for that s
if $O_1.v_{end}(s) > Rf(O_2,v).v_{end}(s)$ and $O_1.v_{end}(s) > v(s)$ then O_1 dominates O_2 for that s
otherwise neither dominates the other for that s.

We choose which operation dominates the other by finding the smallest s for which one dominates the other.

Note: we can use $v \uparrow Rf(O_2,v).v_{end}$ instead of $Rf(O_2,v).v_{end}$, and go back to the conventional definition of dom!

Claim: $v \le O_1.v_{end} \Rightarrow O_1 \oplus O_2 = O_1 \oplus Rf(O_2,v)$

Proof:

Suppose $v \le O_1.v_{end}$

Let s be given.

$v(s) \le O_1.v_{end}(s)$ (because $v \le O_1.v_{end}$)

if $O_1.v_{end}(s) < O_2.v_{end}(s)$

$v(s) \le O_1.v_{end}(s) < O_2.v_{end}(s)$

so $Rf(O_2,v).v_{end}(s) = (O_2.v_{end}(s) > v(s) \ ? \ O_2.v_{end}(s) : 0) = O_2.v_{end}(s)$

$(O_1.v_{end} \uparrow O_2.v_{end})(s)$

$= \max(O_1.v_{end}(s), O_2.v_{end}(s))$

$= \max(O_1.v_{end}(s), Rf(O_2,v).v_{end}(s))$

$= (O_1.v_{end} \uparrow Rf(O_2,v).v_{end})(s)$

$= (O_1 \oplus Rf(O_2,v)).v_{end}(s)$

else

$O_1.v_{end}(s) \ge O_2.v_{end}(s)$

$(O_1.v_{end} \uparrow O_2.v_{end})(s)$

$= \max(O_1.v_{end}(s), O_2.v_{end}(s))$

$= O_1.v_{end}(s)$

$= \max(O_1.v_{end}(s), 0)$

$= \max(O_1.v_{end}(s), O_2.v_{end}(s) > v(s) \ ? \ O_2.v_{end}(s) : 0)$

$= \max(O_1.v_{end}(s), Rf(O_2,v).v_{end}(s))$

$= (O_1.v_{end} \uparrow Rf(O_2,v).v_{end})(s)$

$= (O_1 \oplus Rf(O_2,v)).v_{end}(s)$

if $dom(O_1,O_2)$

$\exists s'$ st $O_1.v_{end}(s') > O_2.v_{end}(s')$ and $\forall s < s'$, $O_1.v_{end}(s) = O_2.v_{end}(s)$

$O_1.v_{end}(s') > O_2.v_{end}(s') \ge Rf(O_2,v)).v_{end}(s')$

$\forall s < s'$, $O_1.v_{end}(s) = O_2.v_{end}(s) \ge Rf(O_2,v)).v_{end}(s)$

i.e. $O_1.v_{end}(s') > Rf(O_2,v)).v_{end}(s')$ and $\forall s < s'$, $O_1.v_{end}(s) \ge Rf(O_2,v)).v_{end}(s)$

Hence $dom(O_1, Rf(O_2,v))$

$(O_1 \oplus O_2).value$

$= dom(O_1,O_2) \ ? \ O_1.value : O_2.value$

$= O_1.value$

$= dom(O_1, Rf(O_2,v)) \ ? \ O_1.value : Rf(O_2,v).value$

$= (O_1 \oplus Rf(O_2,v)).value$

else if $dom(O_2,O_1)$

$\exists s'$ st $O_2.v_{end}(s') > O_1.v_{end}(s')$ and $\forall s < s'$, $O_2.v_{end}(s) = O_1.v_{end}(s)$

$v(s') \le O_1.v_{end}(s) < O_2.v_{end}(s')$

so $Rf(O_2,v).v_{end}(s') = ((O_2.v_{end}(s') > v(s')) \ ? \ O_2.v_{end}(s') : 0) = O_2.v_{end}(s')$

Let $s < s'$

$v(s) \le O_1.v_{end}(s) = O_2.v_{end}(s)$

Suppose $v(s) = O_1.v_{end}(s) = O_2.v_{end}(s)$

$Rf(O_2,v).v_{end}(s) = ((O_2.v_{end}(s) > v(s)) \ ? \ O_2.v_{end}(s) : 0) = 0$

We want to show $dom(Rf(O_2,v),O_1)$
i.e. need to show
$\exists s'$ st $Rf(O_2,v).v_{end}(s') > O_1.v_{end}(s')$ and $\forall s < s'$, $Rf(O_2,v).v_{end}(s) \geq O_1.v_{end}(s)$
$Rf(O_2,v).v_{end}(s') = ((O_2.v_{end}(s') > v(s')) \ ? \ O_2.v_{end}(s') : 0)$

We need to show $\neg dom(O_1, Rf(O_2,v))$

$(O_1 \oplus O_2).value$
= $dom(O_1,O_2) \ ? \ O_1.value \ : \ O_2.value$
= $O_2.value$
?
= $dom(O_1, Rf(O_2,v)) \ ? \ O_1.value \ : \ O_2.value$
= $dom(O_1, Rf(O_2,v)) \ ? \ O_1.value \ : \ Rf(O_2,v).value$
= $(O_1 \oplus Rf(O_2,v)).value$

TODO: This may be unprovable suggesting we need a new definition of merge that accounts for using RFactors.

18.2.1 Problem with the approach

It turns out that the above approach is incorrect! i.e. basically we cannot use a dom relation on vector times and use this to pick which operation dominates the other. The problem is that it doesn't account for causal relationships between operations.

For example, consider 3 sites S_0,S_1,S_2 with $S_0 < S_1 < S_2$. Consider the following sequence

1. S_0 assigns value 0, S_1 assigns value 1
2. S_2 receives 'assign 0' from S_0
3. S_2 assigns value 2
4. S_0 receives 'assign 1' from S_1
5. S_0 receives 'assign 2' from S_2

The problem occurs in step 5, where S_0 has $S_0.v_{out} = (1,1,0)$, and receives operation from S_2 with $S_2.v_{out} = (1,0,1)$. When we compare these vector times we find $dom(S_0.v_{out}, S_2.v_{out})$, yet the assignment by S_2 should dominate all other assignments.

How to fix the problem: Our earlier approach had a concept of discarding either the Rfactor of O_1 or of O_2 under the merge. We need to do exactly this with v_{end} when we merge operations. i.e. instead of $(O_1 \oplus O_2).v_{end} = O_1.v_{end} \uparrow O_2.v_{end}$, we instead have:

Definition: Given operations O_1,O_2, the merge $O_1 \oplus O_2$ is an operation defined as follows:

$(O_1 \oplus O_2).v_{end} = dom(O_1,O_2) \ ? \ O_1.v_{end} \ : \ O_2.v_{end}$

$(O_1 \oplus O_2).value = dom(O_1,O_2) \ ? \ O_1.value \ : \ O_2.value$

This allows vend to discard information (i.e. losers because of causality). The problem with losers is that they can have low siteids and appear to be winners.

Note that a given site doesn't store historical information about vend, and instead trusts the received vend in order to revert due to losers.

Note however that it is not clear whether we can follow the idea of zeroing each vend(s) that doesn't exceed v(s) when we send a Rfactor.

18.3 Implementation notes

A universal operation stores information about all the assignments in a map keyed by fieldId. For each field it records the value (using a VectorOfByte), and the vector time v_{end}.

From section 8, the control algorithm is:

Generate and apply local operation:
    generate O;
    O_{hb} = O;

Send operation:
    send Rf(O_{hb}, v) for a v satisfying v ≤ remote-O_{hb}.hv;

Receive operation, merge with local HB and apply required changes
    receive O;
    O_{hb} = O_{hb} ⊕ O;

It is assumed that O_{hb} records the vector time hv - which describes the entire set of operations in the history buffer.

The control algorithm involves

- Generation of an assignment operation by applying op(s,t) = (assign field f with value v).
    - workingset.fields[f].value = v
    - O_{hb}.fields[f].value = v
    - O_{hb}.fields[f].v_{end}(s) = t+1
    - O_{hb}.hv(s) = t+1
- Extracting Rfactor to send, w.r.t. some underestimate of what's already present on the remote machine. This involves:
    - For each field f, for each s, if O_{hb}.fields[f].v_{end}(s) > v(s) then need to send this. Only need to send O_{hb}.fields[f].value if there exists such an s.
- Merging O_{hb} = O_{hb} ⊕ O where O is an Rfactor.

- For each field f, for each s, O_{hb}.fields[f].v_{end} ↑= O.fields[f].v_{end}. and assign O_{hb}.fields[f].value = O.fields[f].value if dom(O,O_{hb}) on this field. We assume dom(O,Ohb) on field f if :

18.4 More on merging

A site receives an operation O. It needs to calculate $O_{hb} = O_{hb} \oplus O$. This requires update to O_{hb}.fields[f].v_{end} and O_{hb}.fields[f].value. The update to vend is easy. The update to the value depends on whether O dominates Ohb.

O is RFactor w.r.t. some vector time v. It only sends O.vend(s) values when O.vend(s) > v(s).

[Note: If v(s) = t+1, then op(s,t) is the last op in X(v). If op(s,t) is in O, then O.v_{end}(s) = t+1 = v(s). So need O.v_{end}(s) > v(s) in order for there to be an operation in O not in X(v)]

Algorithm: Iterate in order of increasing s. If O.vend(s) != 0 then we potentially have a dominating operation from O. We can assert O.vend(s) > v(s). We simply compare to Ohb.vend(s). If O.vend(s) > Ohb.vend(s) then we know that O dominates Ohb. Otherwise if Ohb.vend(s) > O.vend(s) then we know that Ohb dominates O. Otherwise it would appear that we have already received this operation! In that case we need to continue scanning!

Alternatively consider that O.vend(s) = 0. That means that O has performed no more ops by site s inside X(v). Therefore we evaluate Ohb.vend(s) and compare to v(s). If Ohb.vend(s) > v(s) then Ohb necessarily dominates O. If Ohb.vend(s) ≤ v(s) then we assume that both sites have recorded the same assignments by site s, and so neither dominates the other.

Algo:

```
loop s in increasing order
{
    if (O.vend(s))
    {
        assert(O.vend(s) > v(s));
        if (Ohb.vend(s) > O.vend(s)) return <Ohb dominates>
        if (Ohb.vend(s) < O.vend(s)) return <O dominates>
    }
    else
    {
        if (Ohb.vend(s) > v(s)) return <Ohb dominates>
    }
}
return <Ohb dominates>
```

19 References

[1]	Operational transform - Control Algorithm Aug 2005 David Barrett-Lennard.
[2]	Operational transform – Merging operations Sep 2005 David Barrett-Lennard

[3]	Lossy Assignment Mar 2008 David Barrett-Lennard.
[4]	Vector Time Feb 2008 David Barrett-Lennard