Chain Clock: Efficient Dependency Tracking for Shared-Memory Systems

Anurag Agarwal
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-0233
anurag@cs.utexas.edu

Vijay K. Garg *
Department of Electrical and Computer Engg.
University of Texas at Austin
Austin, TX 78712-1084
garg@ece.utexas.edu

Abstract

In a concurrent system with N processes, vector clocks of size N, with a component associated with every process, are used for tracking dependencies between the events. Using vectors of size N leads to scalability problems and association of components with processes makes vector clocks cumbersome and inefficient for systems with dynamic number of processes. We present a class of logical clock algorithms, called chain clock, for tracking dependencies between relevant events based on generalizing a process to any chain in the computation poset. Chain clocks are generally able to track dependencies using fewer than N components and also adapt automatically to systems with dynamic number of processes. In particular, we compare the performance of Dynamic Chain Clock (DCC) with vector clock for multithreaded programs in Java. With 1% *of total events being relevant events, DCC requires* 10 *times fewer components than vector clock and the traces are smaller by a factor of* 100. *Although DCC requires shared data structures, it is still* 10 *times faster than the vector clock algorithm in our experiments.*

1 Introduction

A concurrent computation consists of a set of processes executing a sequence of events. Some of these events are unrelated and can be carried out in parallel with each other. Other events must happen in a certain sequence. This information about ordering between events is required by many applications, such as:

1. Debugging and monitoring: Distributed breakpoints [FZ90], data-race detection [MC91, NM90], global predicate detection [CM91, GW91], execution replay [NM92], runtime verification [SRA03], access anomaly detection [DS90], sequential consistency verification[CL02].
2. Fault tolerance and recovery: Process Groups [Bir93], consistent checkpoints [SY85, JZ90].
3. Others : Termination detection [Mat87], causal message delivery [BJ87, RST91].

The order between events in a concurrent computation is usually captured through Lamport's *happened-before* model [Lam78], denoted by $\rightarrow$. To order the events, a timestamp is associated with every event such that comparing the timestamps provides some information about the order of events.

Lamport introduced the notion of *Lamport clock* [Lam78] which assigns an integer to every event as the timestamp. Let the Lamport timestamp for an event e be denoted by $e.L \in \mathcal{N}$. Then for any two events e and f the following, also called the *weak clock condition*[BM93], holds: $e \rightarrow f \Rightarrow e.L < f.L$. In other words, if two events are ordered by the happened-before relation, then their timestamps are also comparable. However, the converse is not true i.e. given two events e and f with $e.L < f.L$, does not imply that $e \rightarrow f$.

This limitation can be removed by using Fidge-Mattern's *vector clock* [Fid89, Mat89]. Here the timestamp is a vector of integers, with a component for every process in the system. With the vector timestamp for an event e denoted by $e.V$, for any two events e and f, the following holds: $e \rightarrow f \Leftrightarrow e.V < f.V$. The comparison between two vectors is done component-wise. It is now possible to ascertain the ordering between two events by comparing their timestamps. The guarantee provided by the vector clock is called the *strong clock condition*[BM93].

The vector clock algorithm requires the vectors to be transferred from one process to another and the

*This document is available as "http://maple.ece.utexas.edu/TechReports/2004/TR-PDS-2004-005.ps". Generated on September 17, 2004.

operations performed on the vectors take time linear in the size of the vectors i.e. are $O(N)$ operations for a system with N processes, making the algorithm unscalable. Charron-Bost [CB91] showed that for all N there exists a computation with N processes which requires a vector clock of size at least N to capture the ordering accurately. As a result, we are forced to have a vector of size N in general to track dependency between the events. Moreover, vector clocks become inefficient and cumbersome in systems where the number of processes can change with time.

In this paper, we present a class of timestamping algorithms called *chain clocks* which alleviate some of the problems associated with vector clocks. The vector clock algorithm is inherently based on the processes in the system; it increments the local component of a process in the timestamp whenever a relevant event occurs. Instead, we abstract out the strategy for choosing the component to be incremented and give a sufficient condition for this strategy to satisfy strong clock condition. In particular, we show that if the events incrementing the same component are totally ordered or in other words, form a *chain*, then the timestamps capture the ordering accurately. Vector clock is just one instance of chain clocks where events on a process constitute a chain. We present other instances of chain clock which can decompose a computation into fewer than N chains in many cases. The dynamic chain clock (DCC) introduced in this paper, finds a chain decomposition of the poset in an online fashion and associates a component of the clock with every chain. The number of chains in this decomposition is bounded by N and in systems with low percentage of relevant events, it requires much fewer components than N. Variable based chain clock (VCC) is an instance of chain clocks which uses chains based on access events of relevant variables in a shared memory system instead of processes in the system. For predicate detection, the number of variables on which the predicate depends is often smaller than N and in such a case, VCC would require fewer components in the timestamp. Both VCC and DCC adapt automatically to process creation and termination as the components of the clock are not bound to specific processes.

We compared the performance of DCC with vector clocks by using a multithreaded program which generated a random poset of events. The results show that DCC provides tremendous savings as compared to vector clocks. For a system with 1% of total events being relevant events, DCC requires 10 times fewer components than the vector clock. As a consequence, the memory requirements of the programs are reduced and the estimated trace sizes are about 100 times smaller. DCC also imposes smaller time overhead on the original computation as operations like comparison and copying are performed on smaller vectors and an order of magnitude speedup was observed in the experiments. DCC can also be used in an offline fashion to compress the vector clock traces generated by a computation.

A drawback of DCC is that it requires shared data structures which makes it more suitable for shared memory systems than distributed system. For the shared memory system, the DCC performs uniformly better than vector clocks on time and space requirements. Our experiments show that DCC is also a viable option for distributed system with large number of processes. In such cases, the time overhead of DCC was also lower than that of vector clock in addition to savings in bandwidth and trace size. Using a server to provide shared data structures needed by DCC is not a limitation in applications like monitoring and predicate detection which in general use an observation system separate from the computation.

The rest of the paper is organized as follows. Section 2 gives the distributed system model and the notation used for the paper. Although the chain clocks are more useful for shared memory systems, we describe our algorithm based on a distributed system for simplicity and ease of understanding as most of the previous work on timestamping events uses a model of distributed systems. We also review the theory of vector clocks. In section 3, we first present the intuition behind chain clock and then give the algorithm for it. Section 4 shows that vector clock is an instance of chain clocks and also presents the DCC algorithm. Section 5 then describes the chain clock algorithm for a shared-memory system and introduces the VCC algorithm. Experimental results comparing the performance of chain clocks with vector clocks are presented in section 6. We propose some possible extensions and optimizations to the basic algorithm in section 7. Section 8 puts our work in perspective with some related work and Section 9 summarizes the contributions of the paper.

2 System Model, Notation and Background

The system consists of N sequential processes (or threads) denoted by $p_1, p_2, \ldots, p_N$. A *computation* in the happened before model is defined as a tuple $(E, \rightarrow)$ where E is the set of events and $\rightarrow$ is a partial order on events in E. Each process executes a sequence of events. Each event is an *internal*, a *send* or a *receive* event. For an event $e \in E$, $e.p$ denotes

the process on which e occurred. We define a relation *precedes*, denoted by $\rightsquigarrow$ between events as follows:
1. $e \rightsquigarrow f$ if $e.p = f.p$ and e immediately precedes f in the sequence of events on process $e.p$.
2. $e \rightsquigarrow f$ if e is a send event and f is the corresponding receive event.

Then Lamport's happened-before relation ($\rightarrow$) on E is the transitive closure of the relation $\rightsquigarrow$. Two events e and f are said to be concurrent if $e \not\rightarrow f$ and $f \not\rightarrow e$ and is denoted by $e \parallel f$. If $(e \rightarrow f) \vee (e = f)$, it is denoted by $e \rightarrow\!\!\!\!= f$.

The events for which the happened-before order needs to be determined are called *relevant* events and the set of such events will be denoted by $R \subseteq E$. Let $e.V$ be the vector timestamp associated with an event e and let $m.V$ be the timestamp associated with a message m.

For a vector u we denote its size by $u.s$. For $1 \leq i \leq u.s$, the i^{th} component of vector u is given by $u[i]$. For performing operations such as max and comparison on two different sized vectors, pad the smaller vector with zeroes till the length of the other vector and then perform the operations in the usual way.

2.1 Vector Clocks

In this section, we give a brief overview of the vector clock algorithm [Fid89, Mat89] for relevant events. The Figure 1 describes the algorithm in terms of the actions performed on occurrence of an event. Each process maintains a local vector V of size N, with a component associated with every process in the system. The process p_i increments its own component on occurrence of a relevant event. It also includes a copy of the local vector in every outgoing message. On receiving a message, the local vector is updated by taking max with the received timestamp. The crucial property satisfied by the vector clock is the *strong clock condition*: $e \rightarrow f \Leftrightarrow e.V < f.V$. From now on, whenever we refer to the vector clock algorithm, it would be the algorithm which increments the clock for relevant events only (also called *weak clocks* in [MS94]).

3 Chain Clocks

As discussed in the system model, we use a poset to represent a distributed computation. This poset is generally represented as a set of chains corresponding to the processes, with edges between the chains corresponding to messages exchanged. The same poset can also be represented in terms of a different set of chains with dependencies among these chains. For example, in Figure 2(a) we represent the computation in the normal way – the processes p_1, p_2, and p_3 are represented as chains and there are dependencies between the chains. We could also represent the same computation as shown in Figure 2(b). In terms of the poset model, these are equivalent representations and the properties and algorithms based on the poset would work regardless of the representation chosen. Chain clocks use this idea to generalize the vector clock algorithm. In the vector clock algorithm, a component of the vector is associated with a process in the system. Instead, chain clocks decompose the poset into a set of chains, which are potentially different from the process chains, and then associate a component in the vector timestamp with every chain. We show that using any set of chains, not necessarily the process chains, suffices to satisfy the strong clock condition.

p_i::
var
V: array[1..N] of integer
initially ($\forall j : V[j] := 0$)

On occurrence of event e:

if e is a receive of message m:
$V := max(V, m.V)$;

if $e \in R$:
$V[i]++$;

if e is a send event of message m:
$m.V := V$;

Figure 1: Vector clock algorithm

With this intuition, we devise different strategies for decomposing the subposet R of relevant events into chains. In many cases, especially when the percentage of relevant events is small, the subposet R can be decomposed into fewer chains than the number of processes in the system. As a result, smaller vectors would be required for timestamping events. As an illustration, consider the computation shown in Figure 3(a). We are interested in detecting the predicate "there is no message in transit". For this predicate, the set R is the set of all send and receive events. The original computation is based on 4 processes or chains. However, as shown in Figure 3(b), the subposet R can be decomposed in terms of two chains.

The algorithm for chain clocks is given in Figure 4. The chain clock algorithm is very similar to the vector clock algorithm and differs mainly in the component of

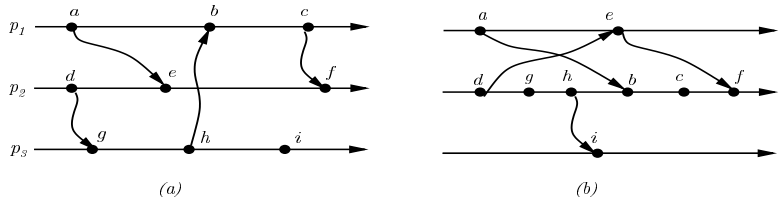

Figure 2: Two representations of the same computation

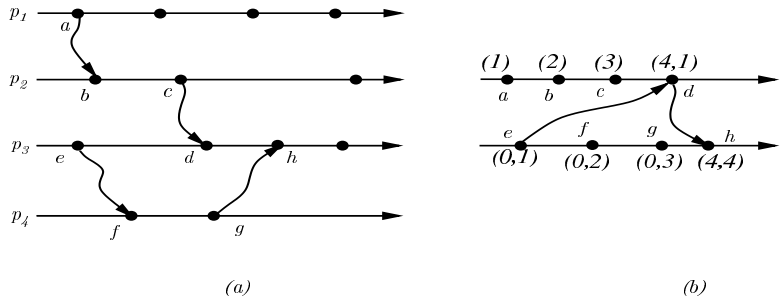

Figure 3: (a) A computation with 4 processes (b) The relevant subcomputation

the clock chosen to increment. The component choosing strategy is abstracted through a primitive called GI (for GetIndex). The primitive GI takes the local vector clock and the event as arguments and returns the index in the vector to be incremented. In the algorithm, process p_i maintains a local vector V. This vector can be of constant size, like in the Fidge-Mattern's vector clock, or can grow during the course of the algorithm. A component of vector V is incremented when a relevant event occurs. The component to be incremented for a relevant event e is decided by the primitive GI and is denoted by $e.c$. Note that, if the index $e.c$ does not exist in the vector V, then V is padded with zeroes till the size of V is $e.c$ and then the $e.c$ component is incremented. While sending a message, we attach a timestamp with it and on receiving a message, we update the local vector by taking max with the timestamp of the message.

```
p_i::
 var
     V: vector of integer
         initially (∀j : V[j] := 0)

 On occurrence of event e:
     if e is a receive of message m:
         V := max(V, m.V);

// Execution point L.
     if e ∈ R :
         e.c := GI(V, e);
//The vector size may increase during this operation.
         V[e.c]++;

     if e is a send event of message m:
         m.V := V;
```

Figure 4: Chain clock algorithm

We define a property on the index chosen by the chain clock algorithm to increment, called *chain-decomposition* property :

(Chain-decomposition) $\forall e, f \in R : e.c = f.c \Rightarrow e \not\parallel f$

Intuitively, it says that if two events increment the same component of the chain clock, they must be comparable or in other words, all the events incrementing the same component must form a *chain*. Considering the chain of events that increment the same component as a process, the chain clock algorithm is very similar to the vector clock algorithm and it can be shown that the chain clock satisfies the strong clock condition. The proof of correctness of the algorithm is given in the Appendix. It can also be shown that two events can be compared in constant time using the chain clock, like the vector clock, if we know the

chain to which the events belong i.e.

$$\forall e, f \in R : e \rightarrow f \Leftrightarrow \quad (e.V[e.c] \leq f.V[e.c]) \\ \wedge (e.V[f.c] < f.V[f.c])$$

4 Strategies for Chain Decomposition

Strategy for chain decomposition is crucial to the performance of the chain clock algorithm as most of the operations in the algorithm are linear in the size of the vectors or equivalently, to the number of chains used for decomposing the set of relevant events that have been executed. The strategies differ in the assumptions required on the subposet R and the information that needs to be maintained. We divide the strategies for chain decomposition into two classes: static and dynamic. Static strategies decompose the subposet R into chains based on some properties satisfied by R and the number of chains does not change with the actual subposet R as long as R satisfies the required properties. On the other hand, dynamic strategies aim to exploit the actual structure of R and discover the chains in an online fashion. We would discuss a static strategy and a dynamic strategy in this section.

4.1 Vector Clock

Vector clock is also a chain clock where the algorithm decomposes R into chains based on the processes. As a result, the size of the vector clocks is N and the call $GI(V, e)$ returns the index as $e.p$. It clearly satisfies the chain decomposition property as the events in a process are totally ordered. It is an example of static strategy as the number of components in the vector are determined by the number of processes in the system, irrespective of the actual computation.

4.2 Dynamic Chain Clock

Dynamic chain clock(DCC) is a chain clock based on a dynamic strategy. The main idea here is to increment a component of the clock such that any concurrent event does not increment the same component. We first present a simple version of the GI primitive for DCC in Figure 5. It uses a vector Z shared between the processes in the system. In a distributed system, a shared data structure can be hosted on a server and the operations on the structure can be performed through remote procedure calls. From an algorithmic perspective, it is equivalent to using a shared data structure and we describe our algorithms assuming shared data structures.

The vector Z is initialized with an empty vector at the start of the program and maintains the maximum value for every component that has been added so far. Line 1 looks for a component which has the same value

```
GI(V, e)::
 var
      Z: vector of integer
           //vector with no components
           initially (Z = φ)

(1)   if ∃i : Z[i] = V[i]
(2)        let j be such that Z[j] = V[j];
(3)   else
           //add a new component
(4)        Z.s++;
(5)        j := Z.s;
(6)   Z[j]++;
(7)   return j;
```

Figure 5: A simple implementation of GI for chain clocks

in V and Z. If the search is successful, then that component is marked as j in line 2. Note that if there are more than one such component, then any one of them could be used. Otherwise, lines 4-5 add a new component to Z and assign j to this new component. Finally, the j^{th} component of Z is incremented and j is returned as the component which the calling process should increment.

As an example consider using DCC for timestamping the computation in Figure 3. Let V_i be the local vector of process p_i. Consider a run of the system in which calls to GI are made in the following order of events: a, b, e, c, d, f, g, h. The call $GI(V_1, a)$ would find that both V_1 and Z are empty and so it would add a component to Z, increment that component to 1 and return 1 as the answer. V_1 would then be set to (1). Upon receipt of message from p_1 in event b, V_2 would be set to (1). As a result, $GI(V_2, b)$ finds that b has up-to-date value for first component and so $Z[1]$ is incremented to 2 and V_2 is set to (2). When $GI(V_3, e)$ is executed, it finds that $V_3[1]$ does not match $Z[1]$ and so a new component is added to Z. So, Z is updated to $(2, 1)$ and V_3 gets set to $(0, 1)$. The timestamps for the remaining events can also be assigned in the same way and the results are shown in Figure 3(b).

For the ease of understanding, we have presented the algorithm in the given form but for correctness of the algorithm we require that read and write(if any) to every component $Z[i]$ occur in an atomic fashion. The given algorithm can be modified to suit this requirement easily and henceforth we assume this guarantee. Intuitively, the given implementation of GI satisfies the chain decomposition property as an event

e is added to a chain only if the last event on that chain was in the causal history of e. The formal proof for this is given in Appendix.

As noted earlier, DCC for distributed systems would require a central server. In most cases this would be a cause of concern for fault-tolerance reasons as the server becomes a single point of failure. However, in our system the state of the server can be easily reconstructed from the states of the processes in the system and in case of a failure a new server can be started with little overhead.

4.2.1 Bounding the number of chains for DCC

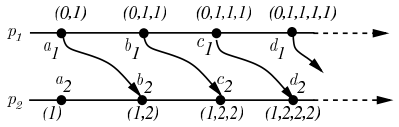

Figure 6: A computation timestamped with simple DCC requiring more than N components

Although the algorithm in Figure 5 provides the chain-decomposition property, it may decompose the computation in more than N chains. For example, consider the computation involving two processes given in Figure 6 with all the events being relevant events. Figure 7 gives a prefix of a run of the computation with the result of the calls to GI made in the order given. We use Z' to denote the updated value of Z after the corresponding call has completed. Note the call $GI(V_2, b_2)$. Here, V_2 has up-to-date information about both first and second components but it chooses to increment the second component. This is a bad choice to make because when b_1 is executed, it is forced to start a new chain. A series of such bad choices can result in a chain clock with unbounded number of components as in the example described above.

The Figure 9 presents an improved version of the GI algorithm which bounds the number of chains in the decomposition. This algorithm maintains another shared data structure called F such that $F[i]$ is the last process to increment $Z[i]$. When looking for a component to increment in $GI(V, e)$, the algorithm checks if there is a component i such that $F[i] = e.p$. If such a component exists, it is incremented otherwise the algorithm looks for an up-to-date component to increment. If no such component exists, then a new component is added. It is easy to see that if process p was the last to increment component i, then it must have the latest value of component i. In this regard, this revised algorithm just gives preference to one component ahead of others in some cases. So the proof of the correctness for this algorithm follows from that of the previous algorithm assuming that accesses to $F[i]$ and $Z[i]$ are atomic.

Consider the same run of computation in Figure 6 which was timestamped using the earlier version of GI. Using this new version of GI, the events in the run can be timestamped as given in Figure 8. The crucial difference is in the way two algorithms timestamp event b_2. At b_2, V_2 has up-to-date information about both the components but the new version of GI chooses to increment the first component as it was the component which was last incremented by p_2. As a result, now c_1 still has up-to-date information about second component and addition of a new component is avoided. Continuing this way, the algorithm timestamps the computation using two components only. In fact, this algorithm guarantees that the number of components in the clock would never exceed N and the next theorem proves this result.

Theorem 1 *Using the primitive GI in Figure 9, the chain clock algorithm ensures that* $\forall e \in E$, $(e.V).s \leq N$.

Proof: At the start of the program, the vector F is empty. Subsequently, a component of F is changed or added to be assigned value x only if there was no other component of F which had value x. As a result, throughout the execution of the algorithm, the

Call	W	Z	V	Z'
$GI(V, a_2)$	ϕ	ϕ	(1)	(1)
$GI(V, a_1)$	ϕ	(1)	$(0,1)$	$(1,1)$
$GI(V, b_2)$	$(1,1)$	$(1,1)$	$(1,2)$	$(1,2)$
$GI(V, b_1)$	$(0,1)$	$(1,2)$	$(0,1,1)$	$(1,2,1)$
$GI(V, c_2)$	$(1,2,1)$	$(1,2,1)$	$(1,2,2)$	$(1,2,2)$
$GI(V, c_1)$	$(0,1,1)$	$(1,2,2)$	$(0,1,1,1)$	$(1,2,2,1)$

Figure 7: A prefix of a run of the computation given in Figure 6

Call	W	Z	F	V	Z'	F'
$GI(V, a_2)$	ϕ	ϕ	ϕ	(1)	(1)	(2)
$GI(V, a_1)$	ϕ	(1)	(1)	$(0,1)$	$(1,1)$	$(2,1)$
$GI(V, b_2)$	$(1,1)$	$(1,1)$	$(2,1)$	$(2,1)$	$(2,1)$	$(2,1)$
$GI(V, b_1)$	$(0,1)$	$(2,1)$	$(2,1)$	$(0,2)$	$(2,2)$	$(2,1)$
$GI(V, c_2)$	$(2,2)$	$(2,2)$	$(2,1)$	$(3,2)$	$(3,2)$	$(2,1)$
$GI(V, c_1)$	$(0,2)$	$(3,2)$	$(2,1)$	$(0,3)$	$(3,3)$	$(2,1)$

Figure 8: A prefix of a run of the computation with new GI given in Figure 6

following invariant holds:

$$\forall i, j : 1 \leq i, j \leq F.s : (i \neq j) \Rightarrow F[i] \neq F[j]$$

Since the values assigned to components of $F[i]$ are between 1 and N, $F.s \leq N$. Also, $\forall e \in E : (e.V).s \leq F.s$ and hence, $\forall e \in E : (e.V).s \leq N$.

■

The actual implementation of the algorithm can be speeded up by using an inverse map for F and we use that for our implementation used in the experiments discussed later.

5 Chain Clock for Shared Memory System

In this section, we adapt the chain clock algorithm for shared memory systems. We first present the system model for shared memory systems.

5.1 System Model

The system consists of N sequential processes (or threads) denoted by $p_1, p_2, \ldots, p_N$. Each process executes a set of events. Each event is an *internal*, *read* or a *write* event. Read and write events are the read and write of the shared variables respectively and generically they would be referred to as *access* events.

A *computation* is modeled by an irreflexive partial order on the set of events of the underlying program's execution. We use $(E, \prec)$ to denote a computation with the set of events E and the partial order $\prec$. The partial order $\prec$ is the smallest transitive relation that satisfies:
1. $e \prec f$ if $e.p = f.p$ and e is executed before f.
2. $e \prec f$ if e and f are access events on the same variable and e was executed before f.

Let the set of variables (shared and local) in the system be $X = \{x_1, x_2, \ldots, x_n\}$. With each shared variable x, a vector $x.V$ is associated. The access to a variable is assumed to be *sequentially consistent*. The rest of the notation can be defined in a way similar to the distributed system model using the relation $\prec$ instead of the $\rightarrow$ relation.

5.2 Chain Clock Algorithm

The chain clock algorithm for the shared memory system is given in Figure 10. Here we have described a shared memory system in which reads are totally ordered but we could also use a model where reads are concurrent [SRA03]. The algorithms described later for our system model can be easily adapted for this model as well.

The vector clock and DCC algorithm described earlier for distributed systems would work for this system model as well. For the vector clocks, the GI primitive would return $e.p$. For DCC, GI would be a call to a method using shared data structures and the actual algorithm would still be the same. In the next section, we give a static and a dynamic strategy for chain clocks useful for shared memory systems.

```
GI(V, e)::
 var
     Z: vector of integer
     F: vector of integer
          initially (Z = φ, F = φ)

 if ∃i : F[i] = e.p
     let j be such that F[j] = e.p;
 else
     if ∃i : Z[i] = V[i]
          let j be such that Z[j] = V[j];
     else
          //add a new component
          Z.s++;
          F.s++;
          j := Z.s;
 Z[j]++;
 F[j] := e.p;
 return j;
```

Figure 9: Improved implementation of GI

```
p_i::
 var
     V: vector of integer
          initially (∀j : V[j] := 0)

 On occurrence of event e :

     if e is an access event of shared variable x:
          V := max(V, x.V);

     if e ∈ R :
          e.c := GI(V, e);
//The vector size may increase during this operation.
          V[e.c]++;

     if e is an access event of shared variable x:
          x.V := V;
```

Figure 10: Chain Clock Algorithm for Shared Memory Systems

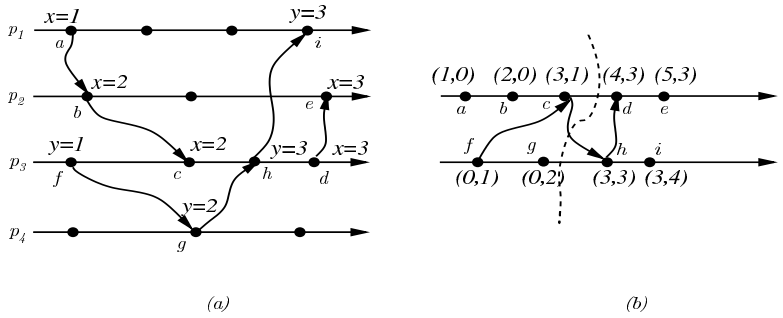

Figure 11: (a) A computation with shared variables x and y (b) Relevant subcomputation timestamped with VCC

5.3 Variable-based Chain Clock

Earlier we described two chain clocks – vector clock and dynamic chain clock – based on the idea of events on a process being a chain. In this section, we present corresponding chain clocks based on variables, called Variable-based Chain Clock (VCC). Since the access events for a variable are assumed to be sequentially consistent, they form a chain. As a result, the set of relevant events can be decomposed in terms of chains based on the variables. Suppose the set of relevant events, R, consists of access events for variables in $Y \subseteq X$. Then, we can have a chain clock which associates a component in the clock for every variable $x \in Y$ in the following way. Let $\theta : Y \rightarrow [1 \ldots |Y|]$ be a bijective mapping from the set of variables to the components in the vector. Then $GI(V, e)$ for an event e which accesses a variable x simply returns $\theta(x)$. It is easy to see that this GI primitive satisfies the chain decomposition property.

VCC is very useful for predicate detection in shared memory systems. Consider a predicate Φ whose value depends on a set of variables Y. In a computation where Φ is being monitored, the set of relevant events would be a subset of the access events for variables in Y. For many predicate detection problems, the size of set Y would be much smaller than the number of processes in the system and hence VCC would result in substantial savings over vector clocks. In fact, using VCC we can generalize the local predicates to predicates over shared variables. Then the predicate detection algorithms like conjunctive predicate detection [GW94] which are based on local predicates would work for shared variables without any significant changes. As an example, consider the computation involving two shared variables x and y given in Figure 11. We are interested in detecting the predicate $(x = 2) \wedge (y = 2)$. Using VCC, we can timestamp the access events of x and y using first component for x and second component for y as shown in the figure. Now the conjunctive predicate detection can be done assuming the access events of x and y as two processes with the vector clock timestamps given by VCC.

In some cases, VCC would require fewer components than the number of variables that need to be tracked. For example, if two local variables belonging to the same process need to be tracked, it would suffice to keep one component for both of them instead of two different components. Here we are exploiting the fact that any two events on a process are totally ordered. We can generalize this idea and use one component for a set of variables whose access events are totally ordered. This could happen when the access to a set of variables is guarded by the same lock. More details for the variable based chain clocks can be found in [AG04]. VCC does not require any shared memory data structure other than that required for any chain clock algorithm in shared memory systems and so it would be beneficial to use VCC over DCC for systems when percentage of relevant events is high but the events are access events of a small set of variables.

We can also have a dynamic strategy based on the variables. In the improved implementation of GI for dynamic chain clocks, we use vector F to keep track of the process which last incremented a component. Instead, with a variable based clock, we can keep track of the last variable which incremented a component and we would get a dynamic chain clock which has the number of components bounded by the number of variables to be tracked.

6 Experimental Results

We performed experiments to compare the performance of chain clocks with the vector clocks. We used DCC for the experiments as VCC is a static chain clock and its performance can be exactly gauged by

the number of variables involved without the need for experiments. For evaluating the performance of DCC, we created a multithreaded application in Java with the threads generating internal, send or receive events. On generating an event, a thread with some probability makes it a send or receive event and based on another probability measure, the event is made relevant. The messages are exchanged through a set of shared queues and the communication pattern between the threads was chosen randomly. The relevant events could be timestamped through either DCC or vector clock. Three parameters were varied for performing the tests: number of threads, number of events per thread and percentage of relevant events. The performance was measured in terms of three parameters: number of components used in the clock, size of trace files and the execution time. For comparing the actual time overhead imposed by timestamping events and removing discrepancies due to varying system loads, we also ran the computations without any timestamping along with the DCC and vector clock. The time overhead was then calculated by subtracting the normal simulation time from the clock times. The experiments were performed on a 2.4GHz Pentium-4 machine running Linux with JDK1.5beta as the Java platform. The default parameters used were: number of threads = 100, number of events per thread = 100 and percentage of relevant events = 1%. One parameter was varied at a time and the results for the experiments are presented in Figures 12-18.

Figure 12 gives the number of components used by DCC as the number of threads in the system is varied from 100 to 5000. We have not shown the vector clock algorithm in the graph as the number of components used there would be equal to the number of threads. Instead, we compare the number of components required by DCC to the width of the relevant poset. The width of the relevant poset is the minimum number of components that any width based clock would require to capture the ordering accurately and so it is a measure of the performance of the optimal algorithm. The algorithm in [TG97] was used for calculating the width of the poset. The results show that DCC requires about 10 times fewer components than vector clock algorithm and hence can provide tremendous savings in terms of space and time. DCC gives nearly optimal results even though we have N as the only provable upper bound.

Figure 13 compares the time required by DCC and vector clock when N is increased from 100 to 5000. Initially, the time taken by the two algorithms is comparable but the gap widens as N increases. For $N = 5000$, DCC was more than 10 times faster than vector clock. This can be attributed to lower cost of copying and comparing smaller vectors in DCC as compared to vector clocks. The profiling results confirmed this by showing copying and comparison of vectors as the two most time consuming operations in the algorithm. Although the time measurements are not truly reliable as they are susceptible to many external factors and depend on the hardware being used, these results strongly suggest that DCC would incur smaller overhead as compared to vector clock despite using shared data structures. Due to smaller vectors used by DCC, the savings in terms of trace size are also substantial as shown in Figure 14. The trace sizes are about 100 times smaller than generated by vector clock and it would become even smaller as the number of events in the system increase. Note that in our experiments, we are estimating the trace sizes and not actually generating the traces.

In Figure 15, we observe the effect of the number of events on the number of components used by DCC as we vary it from 100 to $25,000$ keeping N fixed at 100. The number of components used by DCC gradually increases from 10 to 35. The are two reasons for this behavior. Firstly, as the number of events in the system increases, there is more chance of generating a bigger antichain and secondly, the algorithm is liable to diverge further from the optimal chain decomposition with more events. However, the increase is gradual and even with $25,000$ events, we are able to timestamp events using 35 components which is a reduction of a factor of about 3 over vector clocks.

On the other hand, the percentage of relevant events in the system is an important determinant of the performance of DCC as shown by Figure 16. With more than 10% of events being relevant events, DCC requires more than 60 components and around this point, DCC would start to incur more overhead as compared to vector clock due to contention for the shared data structure. There would still be advantages of smaller traces and lower memory requirements but we feel that the applications to really benefit from DCC would have less than 10% of events as relevant events. This is true for many predicate detection algorithms and in fact, in many cases, less than 1% of the events are relevant events.

To test the viability of DCC for distributed systems, we performed a simple experiment in which a server hosted the shared data structures and the calls to GI were performed through message exchange. The processes were threads running on the same machine communicating with each other using queues but the

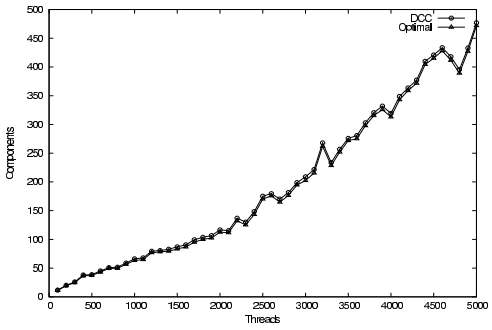

Figure 12: Components vs Threads

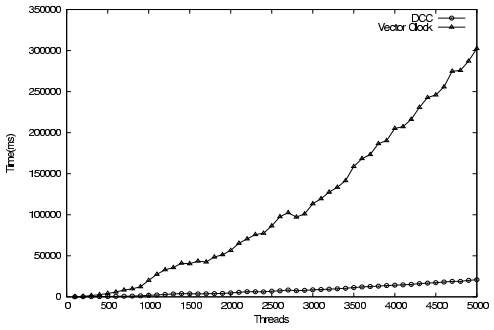

Figure 13: Time vs Threads

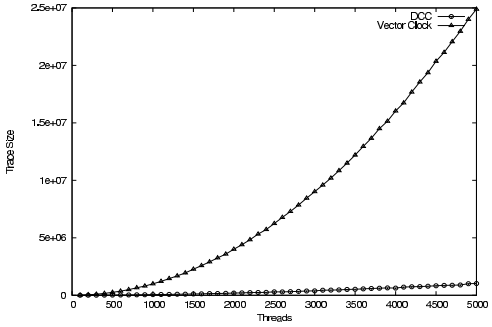

Figure 14: Trace Size vs Threads

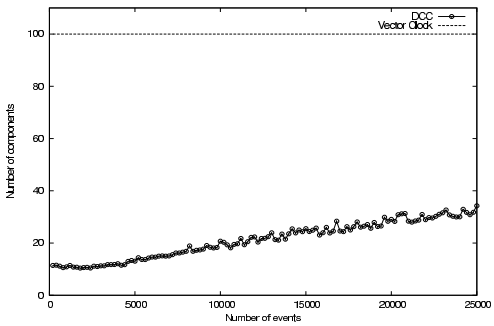

Figure 15: Components vs Events

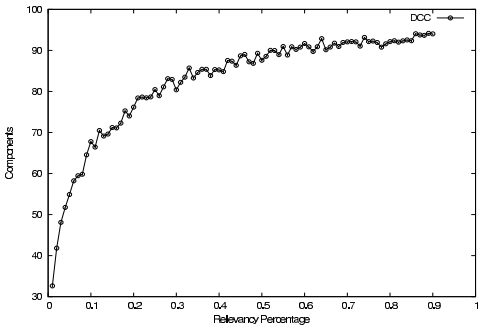

Figure 16: Components vs Percentage of Relevant Events

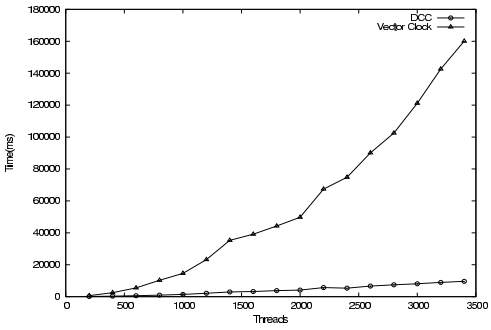

Figure 17: Time vs Threads with a centralized server

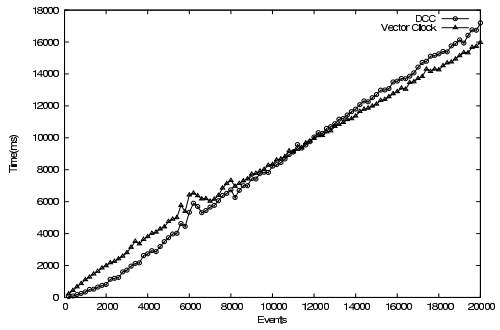

Figure 18: Time vs Events with a centralized server

server was located on a separate machine in the same LAN. Figure 17 shows the result of these experiments when the number of threads in the system was varied. We observe that although the time taken by DCC increases on using a central server, it is still much less than that used by vector clock. The performance of DCC deteriorates as we increase the number of events in the system and in those cases vector clock performs better than DCC as shown by experimental results given in Figure 18. However, note that the execution time is dependent upon many external factors and in a real distributed system the difference between performance of vector clock and DCC may not be as large due to concurrent execution of events but these observations strongly suggest that DCC should outperform vector clock for a system with large N and relatively moderate number of events. For small number of processes, it is still better to use vector clocks if execution time is the main concern. In terms of trace size and bandwidth used by the application, DCC is uniformly better than vector clock.

7 Extensions

In this section, we present some variations of the DCC algorithm which are more suited for certain systems and communication patterns.

7.1 Sticky Components

The DCC algorithm associates the components of the clock with chains that are figured out dynamically at runtime. Instead, we can choose to associate one or more components of the clock with processes or some other static chains. Suppose we want to associate a component i of the clock with process p_j. Then, we mark the component i as "sticky" and the process p_j will always increment component i of the clock. The other processes, while updating their clocks, will not take the component i in consideration. The shared data structures now need to keep the "sticky" information, but they don't need to track the sticky components themselves. As a result, process p_j doesn't have to go through the shared data structures and the update becomes faster. It may also require fewer components in some cases. For instance, if most of the events generated by process p_j are relevant, then it might be better to associate one component with p_j rather than associating the events of p_j with different chains. The "sticky" components can also be turned on/off dynamically based on certain criteria like the percentage of events which are relevant. Vector clocks can be considered to be a special case of DCC with all sticky bits turned on.

7.2 Component Choosing Strategy

The primitive $GI(V, e)$ increments a component about which event e has the latest information in V. Different strategies can be adopted to choose the component to be incremented if the calling process does not already have a component in F. The choice of the strategy can be based on the communication pattern or could be something as simple as choosing the first component which is up-to-date. Consider the case when processes are divided in process groups such that the processes within the group communicate more often with each other than with processes from other group. In such a case, the events from processes within the same group are more likely to form longer chains and so a good strategy would be to give preference to a component which was last incremented by some process in the same process group.

7.3 Hybrid Clocks for Distributed Computing

For a very large distributed system where a centralized server is infeasible, we can design a hybrid algorithm which distributes the work among several servers and reduces the synchronization overhead. The processes in the system are divided into groups and a server is designated to each group which is responsible for finding the chain clock component to increment for events on processes in that group. Considering the chain clock as a matrix with a row for every server instead of a linear vector, each server is made responsible for the components in its corresponding row. Every server can then manage its set of components in the same way as the server in its group. At one extreme, when all the processes are in one group this scheme reduces to the centralized scheme for DCC. On the other extreme, when a group is just one process, this scheme reduces to the vector clock algorithm. The hybrid algorithm would in gen-

eral use more components than DCC as it would be discovering chains within the same group but it would still require fewer components than vector clock and would distribute the load among several servers. It could be very effective for a system where processes can be divided into groups such that the communication within groups is more frequent than outside the group.

7.4 Chain-based Plausible Clock

Plausible clocks [TRA96] provide an approximation to the order between the events in the computation as they only guarantee weak clock condition. A plausible clock based on DCC can also be designed in more than one way. One solution would be to stop adding more components to the DCC once its size has reached a threshold. After that, an event increments the component for which it has the up-to-date information. If no such component exists, then the event chooses one component of the clock based on certain strategy and increments that. Some strategies given in [GP04] may be applicable for chain clocks as well. Other possibility would be to maintain multiple copies of the shared data structures which are independently updated by the processes in the system. This option makes the approach more feasible for distributed systems as it shares the load on multiple servers.

8 Related Work

Charron-Bost [CB91] showed that there exists a computation for which we require a vector clock of size N to capture the dependencies accurately. However, several attempts have been made to develop a space-efficient and scalable timestamping mechanism and various approaches have been tried. Certain optimizations have been suggested for the vector clock algorithm which can reduce the space requirement. For example, Singhal and Kshemkalyani [SK92] present a technique of reducing the overhead of transferring vectors, by transferring only those components of a vector that have changed since previous transfer. This technique can also be used with DCC or any other chain clock.

Another approach to tackle the scalability issues is to relax the vector clock requirements and use some *bounded* vector clocks (whose size is bounded by a constant less than number of processes) which provide weaker guarantees. Two techniques based on this idea have been proposed in the literature: *plausible* [TRA96] clocks and *k-dependency* [BM00] vector clocks. Plausible clocks is a class of logical clocks which aim at approximating the causality relation such that they always satisfy the weak clock condition but may violate its converse in some cases. One simple mechanism is based on a vector clock of fixed size $k < N$, where N is the number of processes in the system, with a component shared by more than one process. This clock gives us better accuracy than the Lamport clock with a constant number of components. The k-dependency vector clocks use a different approach in which an event's bounded timestamp provides causal dependencies that, when recursively exploited, reconstruct the event's vector timestamp. These ideas although attractive, are not used by many real applications as the applications either require accurate ordering information or can do with just Lamport clock.

Ward [War00] proposed an approach based on *dimension* of the poset [DP90] for timestamping events. Although in the worst-case the algorithm still requires N components, in general the size of the vector required is much smaller. The drawback of the algorithm is that it is centralized and computationally expensive, which could considerably slow down a computation. In comparison, DCC incurs a much smaller overhead on the ongoing computation and is more suitable for runtime verification and monitoring. In addition, dimension based clocks do not satisfy some other useful properties satisfied by the clocks based on *width* of the poset [DP90]. In particular, using width-based clocks, two events can be compared in constant time if the chain to which events belong is known. For dimension-based clocks, we need to do a complete component-wise comparison to obtain the order between events. Similarly, using dimension-based clocks we cannot capture the ordering between consistent cuts [GS01] which can be done with width based clocks.

Vector clocks have also been developed for systems which satisfy certain additional properties. For programs with nested parallelism, the idea of clock trees [Aud97] was introduced. A mechanism for timestamping synchronous computations was given in [GS02]. In contrast our method is completely general and we don't require any additional guarantees on the computation poset. Vector clocks have also been used for timestamping events in multithreaded programs [SRA03]. Some logical clocks like weak clocks [MS94] and interval clocks [AV01, CG04] have been proposed to track the *relevant* events in the system but these clocks still require N components for a system with N processes as opposed to DCC which can track relevant events with fewer than N components.

A lot of work has also been done in finding the online chain decomposition of a poset. A result by

Szemérdi reported in [Wes] provides a lower bound for such algorithms. It states that there is no on-line algorithm that partitions all posets of width k into fewer than $\binom{k+1}{2}$. No algorithm is known to achieve this bound and the best known algorithm by Kierstad [Kie81] partitions a poset of width k into $(5^k - 1)/4$ chains. However, our problem is a special case of the general problem where the events are presented in-order to the online algorithm and there is a possibility of better algorithms and lower bounds for it.

9 Conclusion

This paper presents a class of timestamping algorithms called chain clocks which facilitate more efficient dependency tracking in many situations as compared to vector clocks. We make three principal contributions. First, we generalize the vector clocks to a whole class of timestamping algorithms called chain clocks by generalizing a process to any chain in the system. Secondly, we introduce the dynamic chain clock (DCC) which provides tremendous benefits over vector clocks for shared memory systems with low percentage of relevant events. We obtain speedup by an order of magnitude as compared to vector clock and cut down trace sizes by a factor of 100. Thirdly, we present variable chain clock (VCC) which is another useful mechanism for dependency tracking in shared memory systems and is especially suited for predicate detection and monitoring applications.

References

[AG04] A. Agarwal and V. K. Garg. Efficient vector clocks for multithreaded programs. Technical report, University of Texas at Austin, 2004. Available as "http://maple.ece.utexas.edu/TechReports/2004/TR-PDS-2004-003.ps".

[Aud97] K. Audenaert. Clock trees: Logical clocks for programs with nested parallelism. *IEEE Transactions on Software Engineering*, 23(10):646–658, October 1997.

[AV01] S. Alagar and S. Venkatesan. Techniques to tackle state explosion in global predicate detection. *IEEE Transactions on Software Engineering*, 27(8):704 – 714, August 2001.

[Bir93] K. P. Birman. The process group approach to reliable distributed computing. *Commun. ACM*, 36(12):37–53, 1993.

[BJ87] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[BM93] Ozalp Babaoglu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.

[BM00] R. Baldoni and G. Melideo. Tradeoffs in message overhead versus detection time in causality tracking. Technical Report 06-01, Dipartimento di Informatica e Sistemistica, Univ. of Rome, 2000.

[CB91] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39:11–16, July 1991.

[CG04] A. Chakraborty and V. K. Garg. On reducing the global state graph for verification of distributed computations. Technical report, University of Texas at Austin, 2004. Available as "http://maple.ece.utexas.edu/TechReports/2004/TR-PDS-2004-002.ps".

[CL02] H. W. Cain and M. H. Lipasti. Verifying sequential consistency using vector clocks. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 153–154. ACM Press, 2002.

[CM91] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, CA, May 1991. ACM/ONR.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[DS90] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 1–10. ACM Press, 1990.

[Fid89] C. J. Fidge. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):183–194, January 1989.

[FZ90] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *Proc. of the 10th International Conference on Distributed Computing Systems*, pages 134–141, Paris, France, May 1990. IEEE.

[GP04] A. Gidenstam and M. Papatriantafilou. Adaptive plausible clocks. In *24th International Conference on Distributed Computing Systems (ICDCS' 04)*, pages 86–93. IEEE, 2004.

[GS01] V. K. Garg and C. Skawratananond. String realizers of posets with applications to distributed computing. In *20th Annual ACM Symposium on Principles of Distributed Computing (PODC-00)*, pages 72 – 80. ACM, August 2001.

[GS02] V. K. Garg and C. Skawratananond. On timestamping synchronous communications. In *22nd International Conference on Distributed Computing Systems (ICDCS' 02)*, pages 552–560. IEEE, 2002.

[GW91] V. K. Garg and B. Waldecker. Detection of unstable predicates. In *Proc. of the Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, May 1991. ACM/ONR.

[GW94] V. K. Garg and B. Waldecker. Detection of weak unstable predicate in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.

[JZ90] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491, sep 1990.

[Kie81] H. A. Kierstead. Recursive colorings of highly recursive graphs. *Canad. J. Math*, 33(6):1279–1290, 1981.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Mat87] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, 1987.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[MC91] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 24–33. ACM Press, 1991.

[MS94] K. Marzullo and L. S. Sabel. Efficient detection of a class of stable properties. *Distributed Computing*, 8(2):81–91, 1994.

[NM90] R. Netzer and B. Miller. Detecting data races in parallel program executions. In *Advances in Languages and Compilers for Parallel Processing*, pages 109–130. MIT Press, 1990.

[NM92] R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 502–511. IEEE Computer Society Press, 1992.

[RST91] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, July 1991.

[SK92] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43(10):47–52, August 1992.

[SRA03] K. Sen, G. Rosu, and G. Agha. Runtime safety analysis of multithreaded programs. In *Proc. of the 11th ACM Symposium on the Foundations of Software Engineering 2003*, September 2003.

[SY85] R. E. Strom and S. Yemeni. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[TG97] A. I. Tomlinson and V. K. Garg. Monitoring functions on global states of distributed

programs. *Journal for Parallel and Distributed Computing*, 1997. a preliminary version appeared in Proc. of the ACM Workshop on Parallel and Distributed Debugging, San Diego, CA, May 1993, pp.21 – 31.

[TRA96] F.J. Torres-Rojas and M. Ahamad. Plausible clocks: Constant size logical clocks for distributed systems. In *Proceedings of 10th Int'l Workshop Distributed Algorithms*, pages 71–88, New York, 1996. Springer-Verlag.

[War00] P. A. S. Ward. A framework algorithm for dynamic, centralized dimension-bounded timestamps. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, page 14. IBM Press, 2000.

[Wes] Douglas B. West. *The Art of Combinatorics Volume III: Order and Optimization*. Preprint edition.

Appendix

Some additional notation

We introduce some notation that would be used in the analysis of the chain clock algorithm. For an event e, let $A(e) = \{f \in E | f \rightsquigarrow e\}$ be the set of events that precede an event e. An event which is preceded by no other event is called an initial event. A chain in $(E, \rightarrow)$ is a sequence of events $e_0, e_1, \ldots, e_n$ such that $e_i \rightsquigarrow e_{i+1}$. For an event e, $rank(e)$ is defined as the length of the longest chain from an initial event to e.

For an event e, we define $H(e) = \{f \in E | f \rightarrow e\}$. Intuitively, $H(e)$ represents the history of the event e. By the transitivity of happened-before relation, if $e \rightarrow f$, $H(e) \subseteq H(f)$. Let $R(i) = \{f \in R | f.c = i\}$ and $H(e,i) = \{f \in R(i) | f \rightarrow e\}$. Intuitively, $H(e,i)$ is the set of events in the history of e which have incremented component i. On the occurrence of event e, let the value of V at the execution point L in chain clock algorithm (marked in Figure 4) be denoted by $e.W$. Note that in the case of relevant events, this value will be different from the timestamp of event e.

To simplify the analysis, we consider the vectors involved to be of constant size s with zeroes padded from their actual length till s, where s is the maximum length attained by any vector during the course of the algorithm. It is easy to see that this does not change the semantics of the algorithm.

Proof of Correctness of GI for DCC

We now prove that the given implementation of GI along with the chain clock algorithm satisfies the chain-decomposition property. We first introduce some notation that is used in the analysis. We define a relation "accesses i" between events in $R(i)$, denoted by $\triangleright_i$, as follows: for $e, f \in R(i)$, $e \triangleright_i f$ if the call $G(V, e)$ accessed $Z[i]$ before $G(V, f)$. Since the accesses to $Z[i]$ are totally ordered, $\triangleright_i$ is a total order. The order $\triangleright_i$ is also consistent with $\rightarrow$ because for $e, f \in R(i)$, if $e \rightarrow f$, then the call $G(V, f)$ could not have been made before the predecessors of f (including e) occurred. Therefore, for $e, f \in R(i) : e \rightarrow f \Rightarrow e \triangleright_i f$. For $e \in R(i)$, we define $L(e,i) = \{f | (f \in R(i)) \wedge (f \triangleright_i e)\}$ as the set of events which incremented i^{th} component before e. Clearly, $H(e,i) \subseteq L(e,i)$ as $\triangleright_i$ is consistent with $\rightarrow$.

The following lemma is a direct consequence of the algorithm in Figure 5.

Lemma 1 *For $e \in R(i)$, the function call $GI(V, e)$ reads the value of $Z[i]$ as $|L(e,i)|$.*

Proof: The vector Z is initialized to $\overrightarrow{0}$ and $Z[i]$ is incremented in a call $GI(V, f)$ whenever $f.c = i$. Since the accesses are totally ordered, the value of $Z[i]$ read by $GI(V, e)$ would be equal to the number of events that incremented $Z[i]$ before the call to $GI(V, e)$. So the value read is equal to $|L(e,i)| = |\{f | (f \in R(i)) \wedge (f \triangleright_i e)\}|$.

■

The chain decomposition property is equivalent to showing that set of events $R(i)$ form a chain and the following theorem proves that the implementation of GI in Figure 5 coupled with the implementation of the chain clock algorithm in Figure 4 satisfies it.

Theorem 2 *The implementation of primitive GI in Figure 5 with the vector clock algorithm in Figure 4 satisfies the following:*

$$\forall e, f \in R(i) : (e \rightarrow f) \vee (f \rightarrow e)$$

Proof: We prove the theorem by induction on $k = rank(e)$ in the following statements:
(I1) $\forall e \in E, \forall i : 1 \leq i \leq s : e.W[i] = |H(e,i)|$
(I2) $\forall e, f \in R(i) : (rank(f) \leq rank(e)) \Rightarrow f \rightarrow e$
(I3) $\forall e, f \in E : rank(f) \leq rank(e)$
$\Rightarrow (H(f,i) \subseteq H(e,i)) \vee (H(e,i) \subseteq H(f,i))$
Base case($k = 0$): (I1) If $k = 0$, then e is an initial event and V is initialized to $\overrightarrow{0}$. So $\forall i : 1 \leq i \leq s : e.W[i] = 0$. Since e is an initial event, so $\forall i : 1 \leq i \leq s : H(e,i) = \phi$. Therefore, $e.W[i] = |H(e,i)|$.
(I2) If $rank(e) = 0$, then we need to consider f such

that $rank(f) = 0 \wedge f.c = e.c = i$. Suppose such an f exists. Without loss of generality assume that $e \rhd_i f$. Then e increments $Z[i]$ and so $Z[i] \neq 0$. If $rank(f) = 0$, then $f.W[i] = 0$ and so during the execution of $GI(V, f)$, $f.W[i] \neq Z[i]$. So by the algorithm for GI in Figure 5, $f.c \neq e.c$. This leads to a contradiction. So, $\nexists f \in R : (f.c = e.c) \wedge (rank(f) \leq rank(e))$. Hence, the statement holds.
(I3) If $rank(e) = 0$, then we need to consider $f \in E$ such that $rank(f) = 0$. If $rank(e) = rank(f) = 0$, then $H(e,i) = H(f,i) = \phi$. So $H(e,i) \subseteq H(f,i)$.

Induction($k > 0$): (I1) By the chain clock algorithm, $e.W[i] = max\{d.V[i] | d \in A(e)\}$. Now we have two cases:
Case 1: $\nexists d \in A(e)$ such that $d.c = i$:
Then $H(e,i) = \bigcup_{d \in A(e)} H(d,i)$
{ Using the induction hypothesis(IH) for (I3) }
$\Rightarrow \bigcup_{d \in A(e)} H(d,i) = H(g,i)$ for some $g \in A(e)$
{ Using IH for (I1) }
$\Rightarrow max\{d.W[i] | d \in A(e)\} = g.W[i]$
{ $\forall f \in A(e) : f.c \neq i \Rightarrow f.W[i] = f.V[i]$}
$\Rightarrow g.W[i] = max\{f.V[i] | f \in A(e)\}$
{ By algorithm and IH for (I1) }
$\Rightarrow e.W[i] = g.W[i] = |H(g,i)| = |H(e,i)|$.

Case 2: $\exists d \in A(e)$ such that $d.c = i$:
$\{d_1, d_2 \in A(e) : d_1 \parallel d_2$ and IH for (I2)$\}$
$\nexists d_1, d_2 \in A(e) : (d_1 \neq d_2) \wedge (d_1.c = d_2.c)$
Let x be the unique event such that $x \in A(e) : x.c = i$.
{ Definition of $H(e,i)$ and IH for (I2)}
$\Rightarrow \forall f \in H(e,i) : f \underline{\rightarrow} x$
{ IH for (I3) }
$\Rightarrow H(x,i) \supseteq H(d,i), d \in A(e)$
{ IH for (I1) }
$\Rightarrow x.W[i] \geq d.W[i], d \in A(e)$
{ $x.V[i] = x.W[i] + 1 \wedge d.V[i] = d.W[i], d \neq x \in A(e)$}
$\Rightarrow x.V[i] > d.V[i], d \neq x \in A(e)$
{ Definition of max }
$\Rightarrow x.V[i] = max\{d.V[i] | d \in A(e)\}$
{ By algorithm and IH for (I1) }
$\Rightarrow e.W[i] = |H(x,i)| + 1$
{ Using $H(e,i) = x \cup H(x,i)$ }
$\Rightarrow e.W[i] = |H(e,i)|$
Hence Proved.

(I2) First, suppose $f \rhd_i e$. Then, if $GI(V, e)$ returns i as the answer then $Z[i] = e.W[i]$. As a result, $|L(e,i)| = |H(e,i)|$. Since $L(e,i) \subseteq H(e,i)$, this implies $L(e,i) = H(e,i)$ and so $f \in H(e,i)$ and hence $f \rightarrow e$. Now assume $e \rhd_i f$. If $rank(f) \leq rank(e)$, then $e \not\rightarrow f$. Therefore, suppose $e \parallel f$. Then using the same reasoning as above, we get $e \in H(f,i)$ which gives us a contradiction. Therefore, $f \rightarrow e$.

(I3) If $H(e,i) = \phi$ or $H(f,i) = \phi$, the result trivially holds. So assume that $H(e,i) \neq \phi$ and $H(f,i) \neq \phi$. For $g \in H(e,i) : rank(g) < k$, so by using (I2) for elements in $H(e,i)$, $H(e,i)$ is totally ordered. Similarly, $H(f,i)$ is totally ordered. Let $x = maxH(e,i)$ and $y = maxH(f,i)$. Since $x, y \in R(i)$ and $rank(x), rank(y) < k$, (I2) implies that $x \underline{\rightarrow} y$ or $y \underline{\rightarrow} x$. Suppose $x \underline{\rightarrow} y$. Then, $\forall g \in H(e,i) : g \underline{\rightarrow} y$. This implies, that $\forall g \in H(e,i) : g \in H(f,i)$ and so $H(e,i) \subseteq H(f,i)$. The case when $y \underline{\rightarrow} x$ can be handled similarly.

Hence by induction, the claim (I2) is true for all events $e \in R$. This gives us the required result.

■

Proof of Correctness of Chain Clock Algorithm

In this section we assume that the chain clock algorithm satisfies the chain decomposition theorem. As a result, the sets $R(i)$ and $H(e,i)$ are totally ordered for all e and i.

The following lemma shows that the histories of any two events with respect to a chain are ordered.

Lemma 2 $\forall e, f \in E : (H(f,i) \subseteq H(e,i)) \vee (H(e,i) \subseteq H(f,i))$

Proof: If $H(e,i) = \phi$ or $H(f,i) = \phi$, the result trivially holds. So assume that $H(e,i) \neq \phi$ and $H(f,i) \neq \phi$. By chain decomposition property, elements in $H(e,i)$ are totally ordered. Similarly, $H(f,i)$ is totally ordered. Let $x = maxH(e,i)$ and $y = maxH(f,i)$. Since $x, y \in R(i)$, $x \underline{\rightarrow} y$ or $y \underline{\rightarrow} x$. Suppose $x \underline{\rightarrow} y$. Then, $\forall g \in H(e,i) : g \underline{\rightarrow} y$. This implies, that $\forall g \in H(e,i) : g \in H(f,i)$ and so $H(e,i) \subseteq H(f,i)$. The case when $y \underline{\rightarrow} x$ can be handled similarly.

■

Lemma 3 *For an event* $e \in E$, $\forall i : e.W[i] = |H(e,i)|$.

Proof: The proof follows from the proof in previous section as (I2) is guaranteed by the chain decomposition property and (I3) is given by Lemma 2 and given (I2) and (I3), we can prove (I1) which is the desired result.

■

Lemma 4 *For $e, f \in E$, $e \rightarrow f \Rightarrow e.V \leq f.V$.*

Proof: If $e \rightarrow f$, then there is a path from e to f. For the events along a process the vector clock's value never decreases and on receive events we update the vector clock by taking component wise maximum, so $e.V \leq f.V$.

■

Lemma 5 *If $e, f \in R$ and $f \not\rightarrow e$, then $e.V[f.c] < f.V[f.c]$.*

Proof: If $e.c = f.c$, then chain decomposition property and the given condition $f \not\rightarrow e$ imply $e \rightarrow f$. By the chain algorithm, $f.W \geq e.V$ and f increments the component $f.c$. As a result, $e.V[f.c] < f.V[f.c]$.

If $e.c \neq f.c$, consider the sets $H(e, f.c)$ and $H(f, f.c)$. By Lemma 2, $H(e, f.c) \subseteq H(f, f.c)$ or $H(f, f.c) \subseteq H(e, f.c)$. Suppose $H(e, f.c) \supset H(f, f.c)$. Then, $H(e, f.c) \supseteq H(f, f.c) \cup \{f\}$ because of the chain decomposition property and definition of $H(e, i)$. This implies $f \rightarrow e$ which contradicts the assumption. So $H(e, f.c) \not\supset H(f, f.c)$ and therefore, $H(e, f.c) \subseteq H(f, f.c)$. By Lemma 3, $e.W[f.c] \leq f.W[f.c]$. Since $e.c \neq f.c$, $e.V[i] = e.W[i]$ and f increments component $f.c$ and so $e.V[f.c] < f.V[f.c]$.

■

The following theorem shows that the vector clock algorithm satisfies the *strong clock condition* for the relevant events.

Theorem 3 *For $e, f \in R$, $e \rightarrow f \equiv e.V < f.V$*

Proof: We first prove that $(e \rightarrow f)$ implies $(e.V < f.V)$. If $e \rightarrow f$, then by Lemma 4, $e.V \leq f.V$. Moreover, Lemma 5 implies $e.V[f.c] < f.V[f.c]$. Hence, $e.V < f.V$. Thus $e \rightarrow f \Rightarrow e.V < f.V$. The converse follows from Lemma 5.

■