Efficient Dependency Tracking for Relevant Events in Shared-Memory Systems

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

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

ABSTRACT

In a concurrent system with N processes, vector clocks of size N are used for tracking dependencies between the events. Using vectors of size N leads to scalability problems. Moreover, association of components with processes makes vector clocks cumbersome and inefficient for systems with a 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. We compared 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 timestamp traces are smaller by a factor of 100. Although DCC requires shared data structures, it is still 10 times faster than vector clock in our experiments.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*shared memory*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; I.1.1 [**Symbolic and Algebraic Manipulations**]: Algorithms—*Analysis of algorithms*

General Terms

Algorithms, Performance

Keywords

Vector clock, Shared-memory, Predicate Detection

*supported in part by the NSF Grants ECS-9907213, CCR-9988225, Texas Education Board Grant ARP-320, an Engineering Foundation Fellowship, and an IBM grant.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PODC'05, July 17–20, 2005, Las Vegas, Nevada, USA.
Copyright 2005 ACM 1-59593-994-2/05/0007 ...$5.00.

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 the ordering between events is required by many applications such as debugging and monitoring of distributed programs [7, 18] and fault tolerance [20].

The order between the events in a concurrent computation is usually modeled through the Lamport's *happened-before* relation [15], denoted by $\rightarrow$. The *vector clock* algorithm [10, 17] captures this relation by assigning a timestamp to every event in the system. The timestamp is a vector of integers with a component for every process in the system. Let the vector timestamp for an event e be denoted by $e.V$. Then, for any two events e and f, the following holds: $e \rightarrow f \Leftrightarrow e.V < f.V$. In other words, the timestamps are able to capture the ordering information completely and accurately. This guarantee provided by the vector clock is called the *strong clock condition* [4] and forms the requirement desired by the applications from vector clocks.

In the vector clock algorithm, each process maintains a vector of integers. On the send of a message, a process needs to piggyback a vector timestamp on the message which requires copying the vector timestamp. Similarly, on the receive of a message, a process needs to find maximum of two vector timestamps. For a system with N processes, these are $O(N)$ operations making the algorithm unscalable. 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 for applications like predicate detection. In these applications only the order between the *relevant* events needs to be tracked and these relevant events constitute a small percentage of the total number of events. In particular, we show that if the events which increment the same component are totally ordered (or 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. Charron-Bost [6] 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. However, we present some chain clocks 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 such that the number of

chains in this decomposition is bounded by N. Another variant of chain clock, antichain-based chain clock (ACC) gives a partition where the number of chains is bounded by $\binom{k+1}{2}$ – the optimal number of chains in the *online* decomposition of a poset of width k. For predicate detection and monitoring applications, where relevant events are infrequent, both these clocks require much fewer components than N and they can be easily incorporated in tools like JMPaX [18].

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 requires fewer components in the timestamp. All these chain clocks – VCC, DCC, ACC – 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. For purposes of debugging and replay, an application may need to produce a trace containing the vector timestamps of all the events in the system. Using DCC, the estimated trace sizes are about 100 times smaller as compared to the ones generated by vector clocks. DCC also imposes a smaller time overhead on the original computation as operations like comparison and copying are performed on smaller vectors. This can be seen in our experiments where an order of magnitude speedup was observed. DCC can also be used in an off-line fashion to compress the vector clock traces generated by a computation.

A drawback of DCC and ACC is that they require shared data structures which make them 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 a distributed system with a 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. A hybrid model, presented later in the paper, can be used for multithreaded distributed applications which can exploit shared memory for threads of one process while allowing the processes to make their own decisions without the need of a central server.

2. SYSTEM MODEL AND NOTATION

In this section, we present our model of a distributed system. Although the chain clocks are more useful for shared memory systems, we first use a distributed system model for simplicity and ease of understanding as most of the previous work on timestamping events uses this model.

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.

The order between events is given by the Lamport's happened before relation ($\rightarrow$) [15]. It is the smallest transitive relation which satisfies:
1. $e \rightarrow f$ if $e.p = f.p$ and e immediately precedes f in the sequence of events on process $e.p$.
2. $e \rightarrow f$ if e is a send event and f is the corresponding receive event.

Two events e and f are said to be *comparable* if $e \rightarrow f$ or $f \rightarrow e$. If e and f are not comparable, they are said to be *concurrent* and this relationship is denoted by $e \parallel f$. The events for which the happened-before order needs to be determined are called *relevant* events and the set of such events are denoted by $R \subseteq E$. The *history* of an event e consists of all the events f such that $f \rightarrow e$ and is denoted by $\mathcal{H}(e)$. Let $e.V$ be the vector timestamp associated with an event e and let $m.V$ be the timestamp associated with a message m.

The set of events E with the order imposed by Lamport's happened before relation defines a partially ordered set or *poset*. A subset of elements $C \subseteq E$ is said to form a *chain* iff $\forall e, f \in C : e \rightarrow f$ or $f \rightarrow e$. Similarly, a subset of elements $A \subseteq E$ is said to form an *antichain* iff $\forall e, f \in A : e \parallel f$. The *width* of a poset is the maximum size of an antichain in the poset.

For a vector V we denote its size by $V.size$. For $1 \leq i \leq V.size$, the ith component of vector V is given by $V[i]$. For performing operations such as *max* and comparison on two different sized vectors, the smaller vector is padded with zeroes and then the operations are performed in the usual way.

3. CHAIN CLOCKS

The computation 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. 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.

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 are required for timestamping events. For example, consider the computation shown in Figure 1(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 1(b), the subposet R can be decomposed in terms of two chains.

The algorithm for chain clocks is given in Figure 2. The chain clock algorithm is very similar to the vector clock algorithm and differs mainly in the component of the clock chosen to increment. The component choosing strategy is abstracted through a primitive called *GI* (for GetIndex). Process p_i maintains a local vector V which may 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. On the send

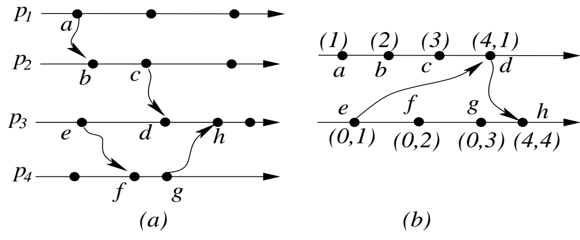

Figure 1: (a) A computation with 4 processes (b) The relevant subcomputation

of a message, a timestamp is piggybacked on it and on the receipt of a message, the local vector is updated by taking *max* with the timestamp of the message.

We define a property on the index returned by *GI*, called **Chain-Decomposition** property:

$$\text{For all distinct } e, f \in R : e.c = f.c \Rightarrow e \not\parallel f$$

Intuitively, it says that all the events which increment the same component must form a *chain*. The following theorem shows that if *GI* primitive satisfies the chain-decomposition property, the chain clock satisfies the strong clock condition.

THEOREM 1. *Given that the GI primitive satisfies the chain decomposition property, the following holds*

$$\forall e, f \in R : e \rightarrow f \Leftrightarrow e.V < f.V$$

PROOF. Consider $e, f \in R$.

$(\Rightarrow)$ $e \rightarrow f \Rightarrow e.V < f.V$
Consider the events along the path from e to f. For the events along a process, the chain clock's value never decreases and for the receive events, the chain clock is updated by taking the component-wise maximum of the local and the received vectors. Hence, $e.V \leq f.V$. Moreover, the component $f.c$ is incremented at f and hence, $e.V[f.c] < f.V[f.c]$. Therefore, $e.V < f.V$.

$(\Leftarrow)$ $e \not\rightarrow f \Rightarrow e.V \not< f.V$
If $f \rightarrow e$, then $f.V < e.V$ and hence $e.V \not< f.V$. Now consider $e \parallel f$. By chain decomposition property, $e.c \neq f.c$. Let g be the last event in the history of f such that $g.c = e.c$. This event is uniquely defined as the set of events which increment a component form a total order by the chain decomposition property.

```
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);
      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 2: Chain clock algorithm

First assume that g exists. By the chain clock algorithm, it follows that $g.V[e.c] = f.V[e.c]$. Events g and e must be comparable as both of them increment the component $e.c$. If $e \rightarrow g$, then $e \rightarrow f$ which leads to contradiction. If $g \rightarrow e$, then $e.V \geq g.V$ and e increments the component $e.c$. Therefore, $e.V[e.c] > g.V[e.c] = f.V[e.c]$.

Now suppose that g does not exist. If no event in the history of f has incremented the component $e.c$, then $f.V[e.c] = 0$. Since e increments the component $e.c$, $e.V[e.c] > f.V[e.c]$.

In both the cases, we have $e.V[e.c] > f.V[e.c]$ and hence, $e.V \not< f.V$. $\Box$

Similar to vector clocks, chain clocks also allow two events to be compared in constant time if the chains containing the events are known. The following lemma forms the basis for this constant time comparison between timestamps.

LEMMA 1. *The chain clock algorithm satisfies the following property:*

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

PROOF. Follows from the proof of Theorem 1 $\Box$

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.

4. DYNAMIC CHAIN CLOCK

Dynamic chain clock (DCC) is a chain clock which uses a dynamically growing vector. The *GI* primitive finds 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 3. 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 RPC. From an algorithmic perspective, it is equivalent to using a shared data structure and we describe our algorithms assuming shared data structures.

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

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

Figure 3: An implementation of GI for chain clocks

Intuitively, the vector Z maintains the global state information in terms of the number of events executed along every chain in the system. It is initialized with an empty vector at the start of the program and subsequently maintains the maximum value for every component that has been added so far. When a call to $GI(V, e)$ is made, it first looks for a component which has the same value in V and

Z. If the search is successful, that component is incremented. Otherwise, a new component is added to Z and incremented. Finally, the updated component is returned to the calling process. Note that if there are more than one up-to-date components, then any one of them could be incremented. The Figure 1(b) shows one possible timestamp assignment by DCC for the relevant events in computation in Figure 1(a).

For the ease of understanding, we have presented the algorithm in the given form where the whole method is synchronized. However, for correctness of the algorithm we just require the read and write (if any) to every component $Z[i]$ and size variable $Z.size$ be atomic instead of reads and writes to the complete data structure Z being atomic. The given algorithm can be modified to suit this requirement easily. The following theorem shows that the implementation of *GI* satisfies the chain decomposition property. Here we sketch the main idea and a more detailed proof can be found in the technical report [1].

THEOREM 2. *The implementation of GI in Figure 3 for the chain clock algorithm satisfies the chain decomposition property.*

PROOF. Consider a pair of minimal events $e, f \in R$ which violate the chain decomposition property i.e., $e \parallel f$ and $e.c = f.c$. By minimal it is meant that chain decomposition property holds between e and events in $\mathcal{H}(f)$ and also between f and events in $\mathcal{H}(e)$. Since *GI* is synchronized, the calls to *GI* for different events in the system appear in a total order. Value of Z just before the call $GI(V,e)$ is made is denoted by Z_e and value of Z just after the call $GI(V,e)$ is completed is denoted by Z'_e. Similarly, $e.V$ and $e.V'$ denote the values of V just before and after the call $GI(V,e)$ is made, respectively. Without loss of generality, assume that e completes the call to *GI* before f. Then e increments the component $e.c$ and the vector Z is updated so that $Z'_e[e.c] = e.V'[e.c]$. When f calls *GI*, it can update the component $e.c$ only if $f.V[e.c] = Z_f[e.c]$. Since the value of a component is never decreased, $Z_f[e.c] \geq Z'_e[e.c]$ and hence $f.V[e.c] \geq e.V'[e.c]$.

Let the events in the history of event f which increment the component $e.c$ be H. For all $g \in H$, g must be comparable to event e as e and f are a minimal pair which violate the chain decomposition property. If $\exists g \in H : e \to g$, then $e \to f$ leading to contradiction. If $\forall g \in H : g \to e$, then $f.V[e.c] < e.V'[e.c]$ which again leads to contradiction. Hence the *GI* primitive satisfies the chain decomposition property for all the events in the computation. □

Using a central server raises the issues of reliability and performance for distributed systems. A central server is 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 consists only of Z vector which can be reconstructed by taking the maximum of the timestamps of the latest events on each process.

Some simple optimizations can be used to improve DCC's performance and mitigate the communication overhead. The key insight behind these optimizations is that an application does not need to know the timestamp until it communicates with some other process in the system. A process after sending a timestamp request to the server need not wait for the server's reply and can continue with its computation. Similarly, it can combine the *GI* requests for multiple internal events into one message.

4.1 Bounding the number of chains for DCC

Although the algorithm in Figure 3 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 4 with all the events being relevant events.

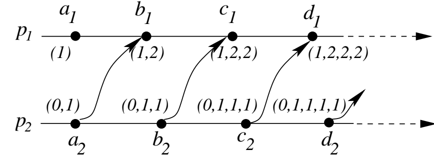

Figure 4: A computation timestamped with simple DCC requiring more than N components

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

Figure 5: A partial run of the computation given in Figure 4

Figure 5 gives a prefix of a run of the computation with the result of the calls to *GI* made in the order given. The values of the variables just after starting execution of *GI* are shown under the variable names themselves (V and Z) and the updated values after the completion of the call are listed under their primed counterparts (V' and Z'). Note the call $GI(V,b_1)$. Here, V 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_2 is executed, it is forced to start a new chain. A series of such bad choices can result in a chain clock with an unbounded number of components as in the example described above.

Figure 6 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 $Z[i]$. 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. If process p was the last to increment component i, then it must have the latest value of component i and in this way, this revised algorithm just gives preference to one component ahead of others in some cases. The proof of correctness for this algorithm follows from that of the previous algorithm assuming that accesses to $F[i]$ and $Z[i]$ are atomic.

The algorithm in Figure 6 maintains the following invariants:

(I1) $Z.size = F.size$
Sizes of Z and F are increased together.

(I2) $\forall i : p_{F[i]}.V[i] = Z[i]$
$F[i]$ maintains the value of the process which last updated the component i.

(I3) $\forall i, j : F[i] \neq F[j]$
Before setting $F[i] = p$, F is scanned to check that there is no j such that $F[j] = p$.

Now consider the same run of the computation in Figure 4 timestamped using the new version of *GI* in Figure 7. The crucial difference is in the way two algorithms timestamp event b_1. At b_1, V 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 b_2 still has up-to-date information about second component and addition of a new component is avoided. Continuing this way, the

```
GI(V,e):: //synchronized
 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.size++;
         F.size++;
         j := Z.size;
 Z[j]++;
 F[j] := e.p;
 return j;
```

Figure 6: Improved implementation of GI

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

Figure 7: A partial run of the computation with new *GI* given in Figure 4

algorithm timestamps the computation using two components only. In fact, this algorithm guarantees that the number of components in the clock never exceeds N as shown by the next theorem.

THEOREM 3. *The primitive GI in Figure 6 with the chain clock algorithm satisfies:* $\forall e \in E$, $(e.V).size \leq N$.

PROOF. From invariant **(I3)**, F contains unique values. Since F contains the process ids, $F.size \leq N$ throughout the computation. By invariant **(I1)**, this implies that $Z.size \leq N$. Moreover, for any event e, $(e.V).size \leq Z.size$ and hence $(e.V).size \leq N$. □

5. ANTICHAIN-BASED CHAIN CLOCK

The DCC algorithm does not provide any bound on the number of chains in the decomposition in terms of the optimal chain decomposition. Dilworth's famous theorem states that a finite poset of width k requires at least k chains for decomposition [**?**]. However, constructive proofs of this result require the entire poset to be available for the partition. The best known *online* algorithm for partitioning the poset is due to Kierstead [14] which partitions a poset of width k into $(5^k - 1)/4$ chains. The lower bound on this problem due to Szemérdi (1982) as given in [23] states that there is no online algorithm that partitions all posets of width k into fewer than $\binom{k+1}{2}$ chains.

However, the problem of online partitioning of the computation poset is a special version of this general problem where the elements are presented in a total order consistent with the poset order. Felsner [9] has shown that even for the simpler problem, the lower bound of $\binom{k+1}{2}$ holds. As an insight into the general result, we show how any algorithm can be forced to use 3 chains for a poset of width 2. Consider the poset given in Figure 9. Initially two

```
var
    B_1,...,B_k: sets of queues
        ∀i : 1 ≤ i ≤ k, |B_i| = i
        ∀i : q ∈ B_i, q is empty

When presented with an element z:
    for i = 1 to k
        if ∃q ∈ B_i : q is empty or q.head < z
            insert z at the head of q
            if i > 1
                swap the set of queues B_{i-1} and B_i \ {q}
            return
```

Figure 8: Chain Partitioning algorithm

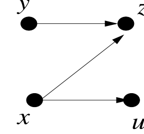

Figure 9: A poset of width 2 forcing an algorithm to use 3 chains for decomposition

incomparable elements x and y are presented to the chain decomposition algorithm. It is forced to assign x and y to different chains. Now an element z greater than both x and y is presented. If algorithm assigns z to a new chain, then it has already used 3 chains for a poset of width 2. Otherwise, without loss of generality assume that the algorithm assigns z to x's chain. Then the algorithm is presented an element u which is greater than x and incomparable to y and z. The algorithm is forced to assign u to a new chain and hence the algorithm uses 3 chains for poset of width 2.

Furthermore, Felsner showed the lower bound to be strict and presented an algorithm which requires at most $\binom{k+1}{2}$ chains to partition a poset. However, the algorithm described maintains many data structures and it can require a scan of the whole poset for processing an element in the worst case. We present a simple algorithm which partitions the poset into at most $\binom{k+1}{2}$ chains and requires at most $O(k^2)$ work per element.

The algorithm for online partitioning of the poset into at most $\binom{k+1}{2}$ chains is given in Figure 8. The algorithm maintains $\binom{k+1}{2}$ chains as queues partitioned into k sets $B_1, B_2..., B_k$ such that B_i has i queues. Let z be the new element to be inserted. We find the smallest i such that z is comparable with heads of one of the queues in B_i or one of the queues in B_i is empty. Let this queue in B_i be q. Then z is inserted at the head of q. If i is not 1, queues in B_{i-1} and $B_i \setminus q$ are swapped. Every element of the poset is processed in this fashion and in the end the non-empty set of queues gives us the decomposition of the poset.

The following theorem gives the proof of correctness of the algorithm.

THEOREM 4. *The algorithm in Figure 8 partitions a poset of width k into* $\binom{k+1}{2}$ *chains.*

PROOF. We claim that the algorithm maintains the followings invariant:
(I) For all i: Heads of all nonempty queues in B_i are incomparable with each other.

Initially, all queues are empty and so the invariant holds. Suppose that the invariant holds for the first m elements. Let z be the next

element presented to the algorithm. The algorithm first finds a suitable i such that z can be inserted in one of the queues in B_i.

Suppose the algorithm was able to find such an i. If $i = 1$, then z is inserted into B_1 and the invariant is trivially true. Assume $i \geq 2$. Then z is inserted into a queue q in B_i which is either empty or has a head comparable with z. The remaining queues in B_i are swapped with queues in B_{i-1}. After swapping, B_i has $i-1$ queues from B_{i-1} and the queue q and B_{i-1} has $i-1$ queues from $B_i \setminus q$. The heads of queues in B_{i-1} are incomparable as the invariant I was true for B_i before z was inserted. The heads of queues in B_i which originally belonged to B_{i-1} are incomparable to each other due to the invariant I. The head of q, z, is also incomparable to the heads of these queues as i was the smallest value such that the head of one of the queues in B_i was comparable to z. Hence, the insertion of the new element still maintains the invariant.

If the algorithm is not able to insert z into any of the queues, then all queue heads and in particular, queue heads in B_k are incomparable to z. Then z along with the queue heads in B_k forms an antichain of size $k+1$. This leads to a contradiction as the width of the poset is k. Hence, the algorithm is always able to insert an element into one of the queues and the poset is partitioned into fewer than $\binom{k+1}{2}$ chains. □

Note that our algorithm does not need the knowledge of k in advance. It starts with the assumption of $k = 1$, i.e., with B_1. When a new element cannot be inserted into B_1, we have found an antichain of size 2 and B_2 can be created. Thus the online algorithm uses at most $\binom{k+1}{2}$ chains in decomposing posets without knowing k in advance. This algorithm can be used to implement the *GI* primitive in a way similar to DCC by associating a component of the chain clock with every queue in the system to obtain ACC. Note that this association has to be between the actual queues and the components such that it does not change due to swapping of the queues between different sets.

Although ACC gives an upper bound of $O(k^2)$ in terms of the width of the poset k, it does not guarantee that the number of chains is less than N. In the worst case, when the width of the poset is N, ACC can give a decomposition consisting of $\binom{N+1}{2}$ chains. Note that the problem of finding the chain decomposition in our case is a slightly more special version of the problem considered by Felsner as we also have the knowledge of one possible partition of the poset into N chains. The lower bound for this problem is not known.

However, one can use an approach where ACC is used till the number of chains is below a bound l and switches to the normal vector clock algorithm thereafter. Some of the chains produced by ACC can be reused for the vector clock algorithm and new chains can be added for the remaining processes. If l is chosen to be small, the upper bound for chain decomposition is close to N but the algorithm requires more than $\binom{k+1}{2}$ chains for decomposing many posets. On the other hand, a bigger value of l results in the algorithm requiring many more chains than N in the worst case but less than $\binom{k+1}{2}$ for a large number of posets. In particular, it can be shown [1] that choosing $l = O(\sqrt{N})$ results in an algorithm which does not require more than $\binom{k+1}{2}$ chains when $\binom{k+1}{2} < N$ and uses $2N - O(\sqrt{N})$ chains in the worst case.

6. CHAIN CLOCKS FOR SHARED MEMORY SYSTEM

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

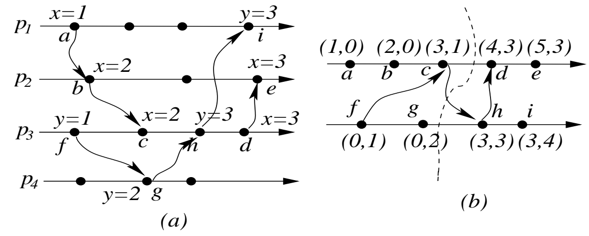

Figure 10: (a) A computation with shared variables x and y (b) Relevant subcomputation timestamped with VCC

6.1 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 are 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.

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.

Here we have considered a shared memory model which considers only the "happened-before" relation between the processes as opposed to other models for shared memory systems. This model, with slight modifications, is the one which is generally used in runtime verification tools like JMPaX [18]. The vector clock and DCC algorithm described earlier for distributed systems work for this system model as well.

6.2 Chain Clock Algorithm

The chain clock algorithm for the shared memory system is given in Figure 11. In the next section, we give some more strategies for choosing a component to increment in chain clock for shared memory systems.

6.3 Variable-based Chain Clock

In this section, we present 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 the set Y. 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 is a subset of the access events for variables in Y. For many predicate detection problems, the size of set Y is much

smaller than the number of processes in the system and hence VCC results 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 [13] which are based on local predicates work for shared variables without any significant change. As an example, consider the computation involving two shared variables x and y given in Figure 10. 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 requires 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 suffices to keep just one component for both of them. 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 happens when the access to a set of variables is guarded by the same lock. 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 is beneficial to use VCC over DCC for systems when percentage of relevant events is high but the events access a small set of variables.

A dynamic strategy based on the variables can also be devised. The improved implementation of *GI* for DCC can be modified such that F keeps track of the last variable to increment a component. This results in a dynamic chain clock with the number of components bounded by the number of variables to be tracked.

7. EXPERIMENTAL RESULTS

We performed experiments to compare the performance of DCC and ACC with the vector clocks. 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, makes it relevant. The messages are exchanged through a set of shared queues and the communication pattern between the threads was chosen randomly. The relevant events are timestamped through DCC, ACC or vector clock. Three parameters were varied

```
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 11: Chain Clock Algorithm for Shared Memory Systems

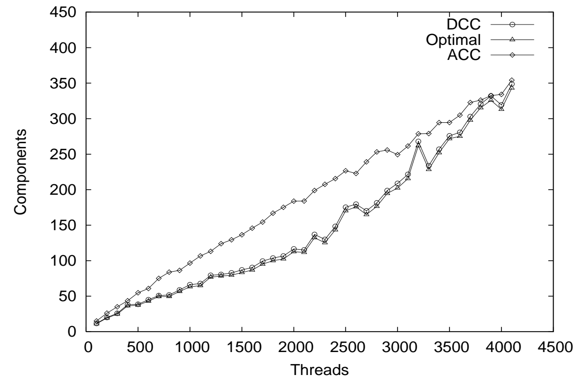

Figure 12: Components vs Threads

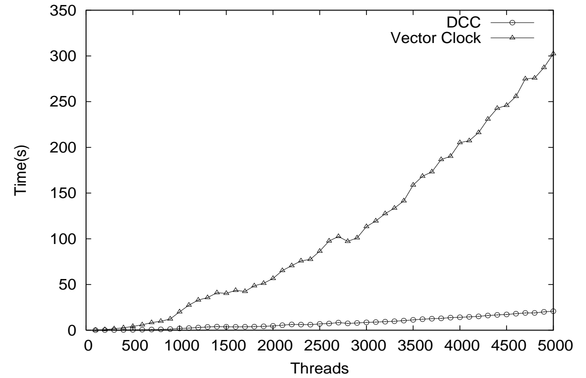

Figure 13: Time vs Threads

during the tests: the number of threads (N), the number of events per thread (M) and the percentage of relevant events (α). The performance was measured in terms of three parameters: the number of components used in the clock, the size of trace files and the execution time. A trace file logs the timestamps assigned to relevant events during the execution of the computation. In our experiments we only estimate the size of the trace files and not actually write any timestamps to the disk. The default parameters used were: $N = 100$, $M = 100$ and $\alpha = 1\%$. One parameter was varied at a time and the results for the experiments are presented in Figures 12-17.

Figure 12 gives the number of components used by DCC and ACC as N is varied from 100 to 5000. The number of components used by the vector clock algorithm is always equal to N. We compared the number of components required by chain clocks to the width of the relevant poset which is a measure of the performance of the optimal algorithm. 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. For our experiments ACC did not perform as well as DCC. The reason for this is that ACC can use new queues even when the incoming event can be accommodated in existing queues. For our experiments, this turned out to be detrimental but for some applications ACC might perform better than DCC. However, we only used DCC for the rest of our experiments as it was performing better than ACC and it is a simpler algorithm with the worst case complexity bounded by that of the vector clocks.

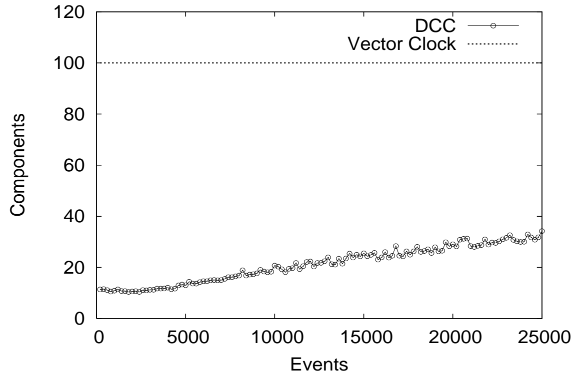

Figure 14: Components vs Events

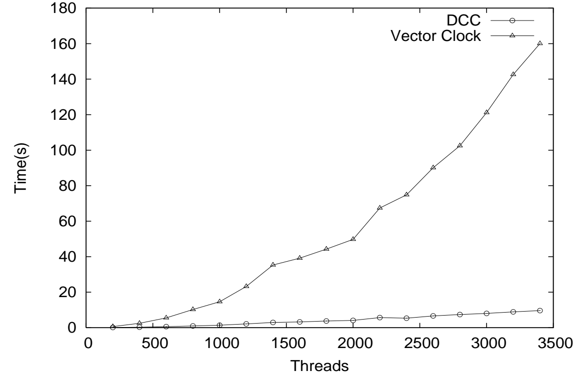

Figure 16: Time vs Threads with a centralized server

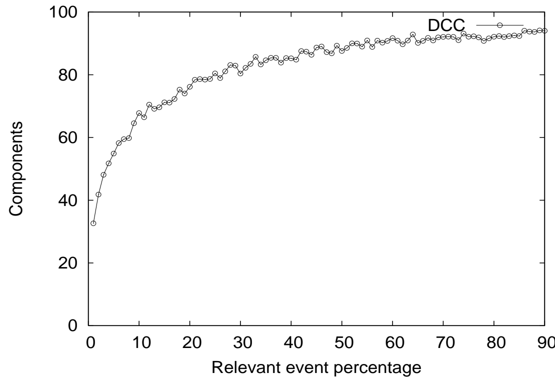

Figure 15: Components vs Relevant Event Fraction

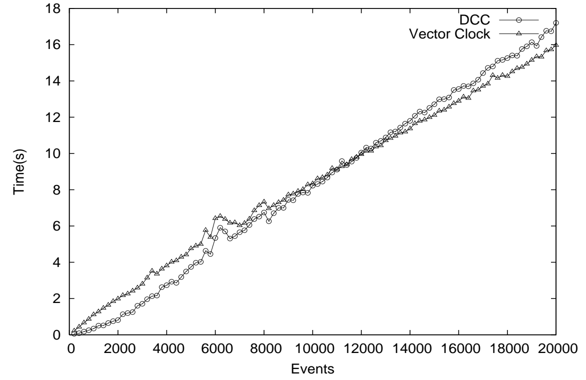

Figure 17: Time vs Events with a centralized server

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 the lower cost of copying and comparing smaller vectors in DCC as compared to vector clocks and the profiling results confirmed this by showing copying and computing the maximum 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 incurs smaller overhead than vector clock despite using shared data structures. The difference between the execution times of vector clocks and DCC is reduced by the optimization of sending only updated components [19]. However, this optimization imposes an overhead of $O(N^2)$ memory per process in the case of vector clocks which could be prohibitive for a system with large number of processes. The system used for performing the reported experiments does not incorporate these optimizations.

In Figure 14, we observe the effect of M 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. There are two reasons for this behavior. Firstly, as M increases, there is more chance of generating a bigger antichain and secondly, the algorithm is likely 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. Due to smaller vectors used by DCC, the estimated trace sizes were about 100 times smaller than generated by vector clock as the average chain clock size during the run of the algorithm is even smaller.

Finally, α is an important determinant of the performance of DCC as shown by Figure 15. With $\alpha > 10\%$, DCC requires more than 60 components and around this point, DCC starts to incur more overhead as compared to vector clock due to contention for the shared data structure. The benefit of smaller traces and lower memory requirements still remains but the applications to really benefit from DCC would have $\alpha < 10\%$. This is true for many predicate detection algorithms where 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 server was located on a separate machine in the same LAN. Figure 16 shows the result of these experiments when N was varied. We observe that although the time taken by DCC increases in this case, 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 in Figure 17. Again depending upon the external factors, these results might vary but they show that for a moderately large distributed system, DCC can compete with vector clock. However,

for small number of processes, it is still better to use vector clocks if execution time is the main concern.

8. EXTENSIONS

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

8.1 Static Components

The DCC algorithm can be modified to associate some components of the clock with static chains like processes. For example, to associate component i of the clock with process p_j, the component i is marked as "static". Now, process p_j always increments component i and the other processes do not consider component i while updating their clocks. The shared data structures need to maintain the list of "static" components, but they do not need to track the static components themselves. Moreover, process p_j does not have to go through the shared data structures to increment the static component. 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.

8.2 Component Choosing Strategy

Different strategies can be adopted to choose the component to be incremented in primitive $GI(V,e)$ if the calling process does not already have a component in F. The decision can be based on the communication pattern or could simply be 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.

8.3 Hybrid Clocks for Distributed Computing

For a large distributed system where a centralized server is infeasible, a hybrid algorithm which distributes the work among several servers and reduces the synchronization overhead is more suitable. The processes in the system are divided into groups and a server is designated to each group which is responsible for finding the component to increment for events on processes in that group. Considering the chain clock as a matrix with a row for every server, each server is made responsible for the components in its corresponding row. 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, it reduces to the vector clock algorithm. The hybrid algorithm in general uses more components than DCC as it discovers chains within the same group. However, it still requires fewer components than vector clock and distributes the load among several servers. It could be very effective for multithreaded distributed programs as the threads within one process can use shared memory to reduce the number of components and different processes can proceed independently in a manner similar to vector clocks.

9. RELATED WORK

Certain optimizations have been suggested for the vector clock algorithm which can reduce the bandwidth requirement. For example, in [19] by transferring only the components of a vector that have changed since previous transfer, the overhead of vector transfer can be reduced. However, it requires the channels to be FIFO and imposes an $O(N^2)$ space overhead per process. This technique can also be used with DCC or any other chain clock but the savings might vary depending upon the type of the clock.

One approach to tackle the scalability issue with vector clocks is to weaken the guarantees provided and use clocks with bounded size. Two techniques based on this idea have been proposed in the literature: *plausible* [21] clocks and *k-dependency* [5] vector clocks. Plausible clocks approximate the causality relation by guaranteeing the weak clock condition and try to satisfy the strong clock condition in most cases. The k-dependency vector clocks provide causal dependencies that, when recursively exploited, reconstruct the event's vector timestamp. These ideas are not used by many real applications as either complete ordering information is required or Lamport clock is sufficient. In contrast, DCC tracks the ordering information accurately with fewer than N components.

Ward [22] proposed an approach based on *dimension* of the poset for timestamping events. In general, the dimension of a poset is smaller than its width and hence this algorithm may require fewer components than chain clocks. However, it is an off-line algorithm and requires the complete poset before assigning the timestamps. 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 lack some properties satisfied by the width-based clocks like chain clocks. 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 [11] which can be done with width-based clocks.

Vector clocks for systems satisfying certain extra properties have also been developed [3, 12]. In contrast our method is completely general. Some logical clocks like weak clocks [16] and interval clocks [2] have been proposed to track the *relevant* events in the system but these clocks still require N components as opposed to DCC which can track relevant events with fewer than N components.

10. CONCLUSION

This paper presents a class of timestamping algorithms called chain clocks which track dependency more efficiently than 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 a low percentage of relevant events. We obtain speedup of an order of magnitude as compared to the vector clocks and cut down trace sizes by a factor of 100. Thirdly, we present the variable-based 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.

11. ACKNOWLEDGEMENTS

We thank Michel Raynal, Neeraj Mittal and Jayadev Misra for their suggestions on the initial draft of the paper. Members of the PDS lab at UT Austin provided great help in preparing the final version of the paper. Finally, we thank the anonymous referees for their thorough comments and suggestions.

12. REFERENCES

[1] A. Agarwal and V. K. Garg. Chain clocks: Efficient dependency tracking for shared-memory systems. Technical report, University of Texas at Austin, 2004. Available as "http://maple.ece.utexas.edu/TechReports/2004/TR-PDS-2004-005.ps".

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

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

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

[5] 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.

[6] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39:11–16, July 1991.

[7] 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.

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

[9] S. Felsner. On-line chain partitions of orders. *Theoretical Computer Science*, 175:283–292, 1997.

[10] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proc. of the 11th Australian Computer Science Conference*, 10(1):56–66, Feb 1988.

[11] 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, Aug. 2001.

[12] 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.

[13] 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.

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

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

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

[17] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the Int'l Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[18] 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*, Sept. 2003.

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

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

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

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

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