Distributed Computing manuscript No.
(will be inserted by the editor)

Anurag Agarwal · Vijay K. Garg

Efficient Dependency Tracking for Relevant Events in Concurrent Systems

Received: date / Accepted: date

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. For the same case, although DCC requires shared data structures, it is still 10 times faster than vector clock in our experiments. We also study the class of chain clocks which perform optimally for posets of small width and show that a single algorithm cannot perform optimally for posets of small width as well as large width.

Keywords Vector clock · Dependency tracking · Predicate detection.

supported by the NSF Grant CNS-0509024.

This is a revised and extended version of the paper "Efficient Dependency Tracking for Relevant Events in Shared Memory System" which appeared in *PODC 2005*

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

Vijay K. Garg
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712
E-mail: garg@ece.utexas.edu

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. Some examples include:

1. Debugging and monitoring: Distributed breakpoints [16], data-race detection [27,29], global predicate detection [10,19], execution replay [30], runtime verification [32], access anomaly detection [13], sequential consistency verification[8].
2. Fault tolerance and recovery: Process Groups [6], consistent checkpoints [34,21].

3. Others : Termination detection [25], causal message delivery [7,31].

The order between the events in a concurrent computation is usually modeled through the Lamport's *happened-before* relation [23], denoted by $\rightarrow$. The happened-before imposes a partial order on the set of events in the system and hence a distributed computation is usually modeled through a partially ordered set or poset. The *vector clock* algorithm [15], [26] captures the happened-before 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, each timestamp is of size N and each of the operations 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. Due to these overheads, vector clocks are used in very few applications in practice.

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.

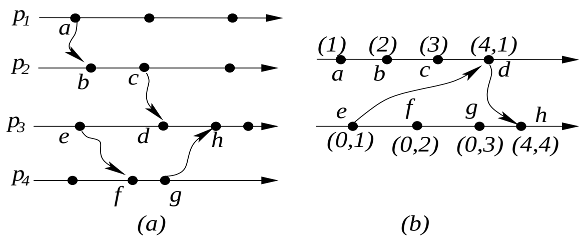

Fig. 1 (a) A computation with 4 processes (b) The relevant subcomputation

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 of relevant events 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 set of relevant events can be decomposed in terms of two chains and timestamped using vector timestamps of size 2.

Vector clock is just one instance of chain clocks where events on a process constitute a chain. Charron-Bost [9] 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 but generally requires fewer than N chains for the decomposition. 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 [11]. The width of a poset is the minimum number of chains required to decompose the poset by any algorithm (online or offline). 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 [32].

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. For applications such as predicate detection, we also exploit the fact that at a given point during the computation, we are interested in the dependencies between only a *subset* of relevant events. We define the notion of *contemporary events* for this purpose and describe an optimization which can reduce the number of components in the chain decomposition by introducing artificial dependencies between events.

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.

We also examine the effect of having process information on optimal chain decomposition. We show that for posets of small width, process information does not help. Moreover, if an algorithm performs optimally for small width, then we can force the algorithm to use more than N chains i.e. the algorithm may perform worse than the vector clock algorithm or DCC. This result shows that it is not possible to devise an algorithm which is guaranteed to perform optimally on all posets and hence the system designers need to decide if they want strong guarantees for posets of small width or posets of large width.

In summary, we make the following contributions:

1. We present a class of logical clock algorithms, called chain clocks, which can be used to track dependencies between *relevant* events accurately.
2. We give specific instances of chain clocks (DCC, ACC, VCC) which can track dependencies efficiently and are useful for different classes of applications.
3. We prove theoretical bounds on the number of chains required by any online algorithm for chain decomposition. This essentially, provides a lower bound on the number of components required by any chain clock algorithm in the worst case.

4. As an application, we consider the use of chain clocks for predicate detection and show a technique to further optimize chain clocks for predicate detection.
5. We present experimental results which demonstrate that for certain applications, chain clocks can outperform the vector clock algorithm by a factor of 10 and reduce the memory requirements also by an order of magnitude.

The paper is organized as follows. Section 2 gives our system model and the notation used in the paper. Section 3 introduces the class of chain clock algorithms and the characterization of the algorithms in this class. Sections 4 and 5 give two specific instances of chain clock algorithms called Dynamic Chain Clock (DCC) and Antichain-based Chain Clock (ACC) respectively. In section 6, we provide lower bounds on the number of chains required by any online chain decomposition algorithm. Section 7 considers the chain clock algorithm for the shared memory system and presents an instance of chain clock algorithm called Variable-based Chain Clock (VCC). In section 8, we apply the chain clock algorithm to the problem of predicate detection and consider an optimization which can further reduce the number of components required by the chain clocks. Section 9 presents some experimental results comparing the performance of chain clocks with vector clocks. Section 10 gives some possible extensions to the chain clocks. We discuss some related work in 11. Finally, section 12 concludes by summarizing our contributions and the future directions of research. Appendix gives proofs for some of the theorems which were stated without proof in the main body.

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.

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 in 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 $\rightsquigarrow$. If e is equal to f or $e \rightarrow f$, we denote it by $e \underline{\rightarrow} f$.

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 i^{th} 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 1is 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 details of the timestamping mechanism used for this computation are given in section 4.

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 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:

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*

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

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

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

// Execution point L.
     if $e \in R$ :
          $e.c := GI(V, e)$;

//The vector size may increase during this operation.
          $V[e.c]++$;
     if $e$ is a send event of message $m$:
          $m.V := V$;
```

Fig. 2 Chain clock algorithm

Proof Consider $e, f \in R$.

(⇒) $e \rightarrow f \Rightarrow e.V < f.V$

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, $e.V < f.V$.

(⇐) $e \nrightarrow f \Rightarrow e.V \nless f.V$

If $f \rightarrow e$, then $f.V < e.V$ and hence $e.V \nless f.V$. 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.

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, $e.V \nless f.V$. □

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:*

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

Proof Follows from the proof of Theorem 1 □

The *GI* primitive can be any function which satisfies the chain decomposition property. In the following sections, we consider some *GI* primitives which satisfy the chain decomposition condition and are suited for certain applications. At this point, it can be seen that vector clock is also a chain clock where the *GI* primitive simply returns $e.p$ as the index for an event e. This *GI* primitive satisfies the chain decomposition property as all the events in a process are totally ordered. As a result, the vector clock algorithm algorithm decomposes R into chains based on the processes and hence the size of the vector clocks is N.

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 remote procedure call (RPC). From an algorithmic perspective, it is equivalent to using a shared data structure and we describe our algorithms assuming shared data structures.

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

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

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

Fig. 3 An implementation of GI for chain clocks

Call	V	Z	V'	Z'
$GI(V,a)$	ϕ	ϕ	(1)	(1)
$GI(V,b)$	(1)	(1)	(2)	(2)
$GI(V,e)$	ϕ	(2)	$(0,1)$	$(2,1)$
$GI(V,f)$	$(0,1)$	$(2,1)$	$(0,2)$	$(2,2)$
$GI(V,g)$	$(0,2)$	$(2,1)$	$(0,3)$	$(2,3)$
$GI(V,c)$	(2)	$(2,3)$	(3)	$(3,3)$
$GI(V,d)$	$(3,1)$	$(3,3)$	$(4,1)$	$(4,3)$
$GI(V,h)$	$(4,3)$	$(4,3)$	$(4,4)$	$(4,4)$

Fig. 4 A partial run of the computation given in Figure 1

that if there are more than one up-to-date components, then any one of them could be incremented.

The Figure 1(b) shows the timestamp assignment by DCC for the relevant events in computation corresponding to the run given in Figure 4 in Figure 1(a). 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').

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 Appendix.

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 $\exists g \in H : e \rightarrow g$, then $e \rightarrow f$ leading to contradiction. If $\forall g \in H : g \rightarrow e$, 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

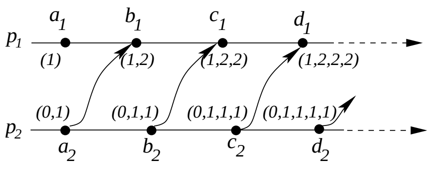

Fig. 5 A computation timestamped with simple DCC requiring more than N components

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 5 with all the events being relevant events. Figure 6 gives a prefix of a run of the computation with the result of the calls to *GI* made in the order given. Variable names follow the same convention as in Figure 4. 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.

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

Fig. 6 A partial run of the computation given in Figure 5

Figure 7 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 7 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 5 timestamped using the new version of *GI* in Figure 8. The crucial difference is in the way the 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

```
$GI(V,e)$:: //synchronized
  var
      $Z$: vector of integer
      $F$: vector of integer
          initially ($Z = \phi$, $F = \phi$)

  if $\exists i : F[i] = e.p$
      let $j$ be such that $F[j] = e.p$;
  else
      if $\exists i : Z[i] = V[i]$
          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 $j$;
```

Fig. 7 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)	$(0,1)$	$(1,1)$	$(2,1)$
$GI(V,b_1)$	$(1,1)$	$(1,1)$	$(2,1)$	$(2,1)$	$(2,1)$	$(2,1)$
$GI(V,b_2)$	$(0,1)$	$(2,1)$	$(2,1)$	$(0,2)$	$(2,2)$	$(2,1)$
$GI(V,c_1)$	$(2,2)$	$(2,2)$	$(2,1)$	$(3,2)$	$(3,2)$	$(2,1)$
$GI(V,c_2)$	$(0,2)$	$(3,2)$	$(2,1)$	$(0,3)$	$(3,3)$	$(2,1)$

Fig. 8 A partial run of the computation with new *GI* given in Figure 5

information about second component and the 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 never exceeds N as shown by the next theorem.

Theorem 3 *The primitive GI in Figure 7 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 $(e.V).size \leq N$. □

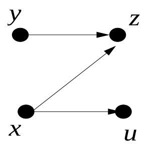

Fig. 9 A poset of width 2 forcing an algorithm to use 3 chains for decomposition

```
var
    $B_1, \ldots, B_k$: sets of queues
        $\forall i : 1 \leq i \leq k, |B_i| = i$
        $\forall i : q \in B_i, q$ is empty

When presented with an element $z$:
    for $i = 1$ to $k$
        if $\exists q \in B_i$ : $q$ 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 $B_i \setminus \{q\}$
            return
```

Fig. 10 Chain Partitioning algorithm

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 [12]. 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 [22] 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 [37] 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 [14] 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 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 10. The algorithm maintains $\binom{k+1}{2}$ chains as queues partitioned into k sets $B_1, B_2 \ldots, B_k$ 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 the head 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 10 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.

To implement the ACC algorithm efficiently, we maintain a linked list L_i for every queue B_i and a vector Z containing the maximum global state as in the DCC algorithm. A node in a linked list L_i contains the index of vector Z corresponding to the chain which the node represents. To compare an element e with the top element of a node n with value j, we compare $e.V[j]$ and $Z[j]$. The element e is comparable with the top element of node n if $e.V[j] = Z[j]$. To swap the queues in B_i and B_{i-1} while keeping the queue represented by n in its place, we remove the node n from L_i, insert it in L_{i-1} and swap the pointers for L_i and L_{i-1}. This way the entire swap operation can be done in $O(1)$ time and doesn't require any change to the representation of the individual vector timestamps. Note that in this implementation, the values stored in linked list nodes are never written to and we just need to ensure that the swapping and the modifications to the linked lists are atomic. A simple way to ensure this is to lock the linked lists L_i and L_{i-1} whenever we are reading the linked list L_i. A more complex scheme involving read/write locks per linked lists can also be devised which would allow multiple readers to read from a linked list.

6 Chain Decomposition with Process Information

In the previous section, the problem of optimal chain decomposition did not take into account the process information associated with every element of the poset. Intuitively, this process information should aid in the chain decomposition of the poset and we should be able to decompose a poset into fewer than $\binom{k+1}{2}$ chains at least for some cases. In this section, we present some results for this problem.

We define the chain decomposition problem for a computation poset with process information, $\mathcal{T}(k,N)$, as a game between players Bob and Alice in the following way:

Bob presents elements of an up-growing partial order of width k to Alice. Information about the decomposition of the poset into N chains is given to Alice in the form of a chain label assigned to every element that is presented. Alice needs to decompose the poset into as few chains as possible.

Let $\eta(N)$ be the integer which satisfies $\binom{\eta(N)+1}{2} \leq N < \binom{\eta(N)+2}{2}$. When N is clear from the context, we will simply use η to denote $\eta(N)$. Note that $\eta = \Theta(\sqrt{N})$.

It is also important to mention that the bound given by Felsner [14] also holds for problem $\mathcal{T}(k,N)$ for the case when $N >> k$ as all the elements in the Felsner's proof for the lower bound can come from different chains, rendering the process information useless.

The lower bound given by Felsner holds regardless of the knowledge of k to Alice. However, for the problem $\mathcal{T}(k,N)$, the bounds are different depending upon Alice's knowledge of k.

For the problem $\mathcal{T}(k,N)$, we show two results

1. The bound given by Felsner still holds for problem $\mathcal{T}(k,N)$ when $k \leq \eta$.
2. If Alice's algorithm meets Felsner's bound for the problem $\mathcal{T}(k,N)$ with $k \leq \eta$ and Alice does not have knowledge of k, then Bob can force Alice to use $N+\beta-1$ chains. Here $\beta = \eta(N-1)$.

For our problem of assigning timestamps to events in a distributed computation, k is in fact unknown to the algorithm. So, these results show that as an algorithm designer, one needs to make a choice between the algorithms which perform well when width is small and the ones which do not use more than N chains in the worst case (such as DCC). We can not get the best of both the worlds with any algorithm.

The proof for the first result closely follows Felsner's proof. Here we essentially show that with $N \geq \binom{k+1}{2}$ chains in the given decomposition, we have enough freedom to generate the posets in Felsner's original proof which can force Alice to use $\binom{k+1}{2}$ chains. We do not need to show anything else as the given chain decomposition into N chains only al-

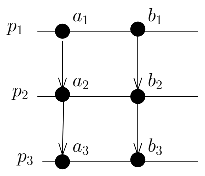

Fig. 11 Ladder Poset of 3 processes

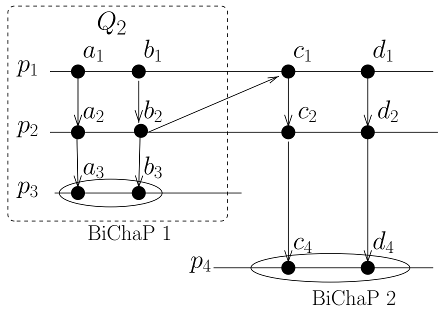

Fig. 12 Building Q_3 from Q_2. Here BiChaPs are chosen arbitrarily

lows Alice to give one decomposition preference over other decompositions and Felsner's proof works for any decomposition chosen.

Theorem 5 *For the problem $\mathcal{T}(k,N)$ with $k \leq \eta$, Bob can force Alice to use $\binom{k+1}{2}$ chains.*

Proof See Appendix. □

Let $\mathcal{S}$ be the class of the algorithms which uses less than or equal to $\binom{k+1}{2}$ chains for a poset of width k when $k \leq \eta$. If k is known to Alice, then at the start of the game, Alice can use an algorithm from $\mathcal{S}$ when $k \leq \eta$ and use an algorithm like DCC which bounds the number of chains by N when $k > \eta$. However, the interesting case is to consider the problem when k is unknown to Alice. For this problem, we show that if Alice uses an algorithm from $\mathcal{S}$ then Bob can force Alice to use $N + \eta(N-1) - 1$ chains in the worst case.

We introduce some of the terminology that would be used in proving this result

- **Ladder Poset:** A ladder poset of a subset $G = \{g_1, \ldots, g_m\}$ of processes consists of two elements a_i and b_i from every process g_i such that $a_i \to a_j$ and $b_i \to b_j$ when $i < j$. Figure 11 shows a ladder poset of three processes $\{p_1, p_2, p_3\}$.
- **BiChaP (Bi Chained Process):** During chain decomposition of the ladder poset, if the elements a_i and b_i for a process p_i are assigned to different chains, then p_i is called a Bi-Chained process or BiChaP.

A BiChaP is very useful from Bob's point of view as it has forced Alice to use more than one chain to decompose the elements of one process. As a result, by generating BiChaPs Bob can force Alice to use more than N chains to decompose a poset of N processes.

Assuming Alice uses an algorithm from class $\mathcal{S}$, we proceed in the following manner to prove the result

1. Bob constructs a poset Q_k of width at most k and $N = \binom{k+1}{2} + 1$ such that Alice is forced to generate at least $k-1$ BiChaPs.
2. Then, Bob extends Q_k to a poset P_k which forces Alice to use at least $N + k - 1$ chains for the decomposition of P_k.

Lemma 2 *For $\mathcal{T}(k,N), k > 1$ with k unknown to Alice, Bob can construct a poset Q_k of width at most k with $N = \binom{k+1}{2} + 1$ such that Alice would be forced to generate at least $k-1$ BiChaPs.*

Proof Let the set of all elements belonging to non-BiChaPs be X_k. We use induction to prove the lemma.

Induction Hypothesis For $\mathcal{T}(k,N)$ with $k > 1, N = \binom{k+1}{2} + 1$, Bob can construct a poset Q_k of width at most k which would force Alice to generate $k-1$ BiChaPs. The elements from non-BiChaPs of Q_k can be decomposed in two chains. In addition, either there exists an element $x_k \in X_k$ such that for all $e \in X_k : e \underline{\to} x_k$, or $X_k = \phi$.

Base Case: k = 2 For $k = 2$ and $N = 4$, Alice can use at most $\binom{k+1}{2} = 3$ chains. Bob constructs a ladder poset and continues to present elements in the process order till Alice generates a BiChaP. Alice has to generate a BiChaP for at least one of the 4 processes otherwise each process would start a new chain and by the end of the 4th process, Alice would have used 4 chains for decomposition. This violates the assumption that Alice uses at most 3 chains for decomposing a poset of width 2.

Hence, Alice was forced to generate $k-1=1$ BiChaP for this case and the resulting poset be Q_2. This poset has width at most 2 as it can be decomposed into two chains along the sides of the ladder poset. The structure of ladder poset ensures that the last element generated by a non-BiChaP is suitable to serve as x_2 provided $X_2 \neq \phi$.

Inductive Step Suppose that poset Q_m exists for $k=m$ with $m-1$ BiChaPs. Bob starts with Q_m and constructs a new ladder poset using processes other than the BiChaPs. This is shown in Figure 12. Let f be the first event executed by a non-BiChaP when the new ladder poset was started from Q_k. If x_k and f exist, then Bob adds a dependency from x_k to f. Figure 12 shows this dependency between the elements b_2 and c_1. Otherwise, there are only inter-process dependencies between the elements.

The BiChaPs themselves form $m-1$ chains and have an element which is incomparable to the new ladder subposet and to the other BiChaPs. Since the non-BiChaPs could be decomposed into two chains in Q_m, they can still be decomposed in two chains as all the elements in the new ladder subposet are greater than the elements in non-BiChaPs in Q_m. This is possible due to the additional dependency that was introduced between x_k and f. In Figure 12, these chains are $\{a_1,a_2,c_1,c_2,c_4\}$ and $\{b_1,b_2,d_1,d_2,d_4\}$. Hence, the new poset Q_{m+1} has width at most $m+1$. If there is a non-BiChaP in the new ladder subposet, then x_{k+1} is the last element from a non-BiChaP in the new ladder subposet. Otherwise, $x_{k+1}=x_k$.

Bob has $\binom{m+2}{2}+1-(m-1)=\binom{m+1}{2}+3$ (non-BiChaPs) processes available for constructing the new ladder subposet. The $2(m-1)$ chains used up by BiChaPs would be unavailable to the other processes. Thus, the number of chains available to Alice for use in the new ladder subposet is $\binom{m+2}{2}-2(m-1)=\binom{m}{2}+3$ as she uses an algorithm from $\mathcal{S}$. So Alice cannot assign elements of one process to one chain for all the processes as the number of chains is less than the number of processes. Hence, Alice would be forced to generate a BiChaP and the total number of BiChaPs now is m.

Hence, the inductive hypothesis holds and the lemma is proved. □

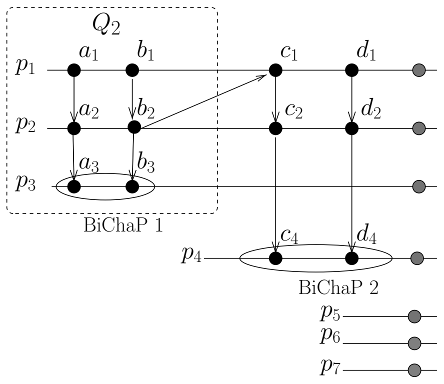

Fig. 13 Extending Q_3 to force Alice to use $N+\beta-1$ chains

Theorem 6 *Let $\beta=\eta(N-1)$. For $\mathcal{T}(k,N), k>1$ with k unknown to Alice, if Alice uses an algorithm from the set $\mathcal{S}$, then Bob can force Alice to use at least $N+\beta-1$ chains.*

Proof Bob first constructs the poset Q_β. Since $N \geq \binom{\beta+1}{2}+1$, using Lemma 2 Bob can force Alice to generate $\beta-1$ BiChaPs in Q_β. After constructing Q_β, Bob makes every process execute an event which is concurrent to every other such event executed by other processes. As a result, the new poset would have width N and Alice would end up using at least $2(\beta-1)+N-(\beta-1)=N+\beta-1$ chains - the BiChaPs using at least $2(\beta-1)$ chains and the others using at least $N-(\beta-1)$ chains. Figure 13 shows this step for $N=7$. □

The ACC algorithm discussed in the previous section belongs to the set $\mathcal{S}$ but it does not make any use of the process information. It does not guarantee that the number of chains in the decomposition is less than N and in the worst case, when the width of the poset is N, ACC can give a decomposition consisting of $\binom{N+1}{2}$ chains. We can modify ACC to perform better than this using process information in the following way.

The algorithm uses ACC till the number of chains is below a bound l and switches to DCC thereafter. The chains produced by ACC can be reused for DCC but at this point, it might be possible that the elements from a single process occur at multiple chain heads. As a consequence, we would not able to guarantee that the total number of chains used by DCC would be less than N in this case. If l is chosen to be small, the upper bound for chain decomposition is close to N but the algorithm may require 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, for $l = N+1$, we show the following result.

Lemma 3 *Let $M(k,N,N+1)$ be the maximum number of chains required by modified ACC for partitioning any poset of width k obtained from a computation involving N processes with $l = N+1$. Then,*

$$M(k,N,N+1) = \begin{cases} \binom{k+1}{2} & \text{if } k \leq \eta \\ 2N - \eta & \text{otherwise} \end{cases}$$

Proof If $k \leq \eta$, then normal ACC does not require more than $\binom{\eta+1}{2}$ chains and hence the modified ACC does not switch to DCC. As a result, $M(k,N,N+1) = \binom{k+1}{2}$ for $k \leq \eta$.

For $k > \eta$, the number of queues may grow more than N. In that case, we switch to DCC. At the time of switching, all the η queues in B_η would be non-empty as $\binom{\eta+1}{2} \leq N$. By the invariant on the queue sets, the queue heads in B_η would be incomparable. Since any two events on the same process are comparable, the different queue heads in B_η come from different processes. Using *DCC* after this point would ensure that the algorithm does not require more than $N-\eta$ additional queues as DCC uses at most one queue per process. So the total number of queues in that case would be at most $N + N - \eta = 2N - \eta$. □

There is still a gap between the lower bound given by Theorem 6 and the bound that the modified ACC achieves. It may be possible to strengthen the lower bound or to have a better algorithm which meets the proven lower bound.

7 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.

7.1 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 [32]. The vector clock and DCC algorithm described earlier for distributed systems work for this system model as well.

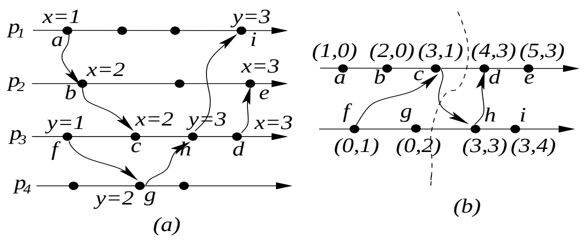

Fig. 14 (a) A computation with shared variables x and y (b) Relevant subcomputation timestamped with VCC

7.2 Chain Clock Algorithm

The chain clock algorithm for the shared memory system is given in Figure 15. In the next section, we give some more strategies for choosing a component to increment in chain clock for shared memory systems.

7.3 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.

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

 On occurrence of event $e$ :

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

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

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

Fig. 15 Chain Clock Algorithm for Shared Memory Systems

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 [19] 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 14(a). 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 the first component for x and the second component for y as shown in the Figure 14(b). 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.

As mentioned earlier, we can also modify VCC to be used in tools such as JMPaX. JMPaX models reads of a shared variable as concurrent events and uses a modified version of vector clocks which captures this relationship. If the model in JMPaX is changed to our model, then VCC can be directly used instead of the vector clock algorithm in JMPaX. Moreover, VCC can also be modified to support the JMPaX memory model. To support the JMPaX model, two vectors V and V_w are associated with every variable instead of a single vector V. The vector V for a variable maintains the maximum timestamp seen for an access event for the variable and the vector V_w maintains the maximum timestamp seen for a write event for the variable. The *GI* primitive used now corresponds closely to the *GI* primitive used for DCC; it creates chains in a dynamic fashion as opposed to chains just being associated statically with variables. The chain clock algorithm modified for this model is given in Figure 16.

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

 On occurrence of event $e$ :

    if $e$ is a read event of shared variable $x$:
        $V := max(V, x.V_w)$;
    if $e$ is a write event of shared variable $x$:
        $V := max(V, x.V)$;

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

    if $e$ is a read event of shared variable $x$:
        $x.V := max(V, x.V)$;
    if $e$ is a write event of shared variable $x$:
        $x.V := max(V, x.V)$;
        $x.V_w := x.V$;
```

Fig. 16 Chain Clock Algorithm for Shared Memory Systems with concurrent reads

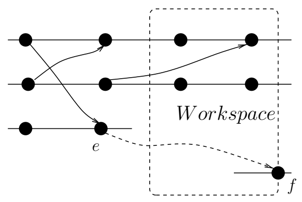

Fig. 17 Adding a new edge while maintaining the same set of contemporary consistent cuts

8 Predicate Detection Using Chain Clocks

Till now we had assumed that we were interested in maintaining the happened before relation between *all* the relevant events executed in the system. However, an application such as predicate detection tries to find a consistent cut only between a subset of relevant events. In general, predicate detection involves a centralized checker process. On the occurrence of a relevant event, the timestamp of the relevant event is sent to the checker process. The checker process repeatedly looks for a consistent cut among the relevant events such that the cut satisfies the predicate, throwing away events which it knows for sure cannot be a part of such a consistent cut. As a result, at any time during the predicate detection, the checker process needs to compare only a subset of events in its *workspace*. This provides an opportunity to reduce the components required for chain clocks for predicate detection. Figure 17 shows a poset and a workspace, where adding an edge from event e to f can reduce the number of components required for chain decomposition without affecting the set of consistent cuts in the workspace. The same observation has been used earlier for bounding the size of the vector clock components [2,28] but not for reducing the number of components in the vector clock.

8.1 Contemporary Events

We use the familiar notion of consistency in event-based model modified to use chains instead of processes. A consistent cut or a global state $G \subseteq R$ is a set of events such that if $e \in G$, then $\forall f \in R : f \rightarrow e \Rightarrow f \in G$. A consistent cut is frequently identified with its *frontier*. For events $e, f \in R$, we say e and f are pairwise consistent iff $(\nexists g \in R : g.c = e.c \wedge e \rightarrow g \rightarrow f) \wedge (\nexists g \in R : g.c = f.c \wedge f \rightarrow g \rightarrow e)$. In terms of chain clocks, this condition can be captured as $f.V[e.c] \leq e.V[e.c] \wedge e.V[f.c] \leq f.V[f.c]$. The frontier of a consistent cut G, $frontier(G)$ is a set of events in G, one from every chain, such that every event is pairwise consistent with every other event.

We introduce an *older-than* relation ($\circ\!\!\rightarrow$) on the set of relevant events (R). It is a non-reflexive, asymmetric and transitive relation. For $e, f \in R$, if e is not older than f, it is denoted by $e \not\circ\!\!\rightarrow f$. If $e \not\circ\!\!\rightarrow f$ and $f \not\circ\!\!\rightarrow e$, then e and f are said to be *contemporary* events and is denoted by $e \parallel f$. The $\circ\!\!\rightarrow$ relation can be characterized by the following properties:

1. $\forall e, f \in R : e \rightarrow f \Rightarrow f \not\circ\!\!\rightarrow e$
2. $\forall e, f, g \in R : (e \circ\!\!\rightarrow f) \wedge (f \underline{\rightarrow} g) \Rightarrow (e \circ\!\!\rightarrow g)$
3. $\forall e, f, g \in R : (e.c = f.c) \wedge (e \underline{\rightarrow} f) \wedge (f \circ\!\!\rightarrow g) \Rightarrow (e \circ\!\!\rightarrow g)$

Given the older-than relation, we are only interested in finding consistent cuts among the contemporary events. In terms of predicate detection, the older-than relation can be interpreted as follows: For an event $f \in R$, all the events $e \in R$ which were deleted from the workspace before f arrived are older-than f. As a result, the workspace always consists of only contemporary events and the checker process finds consistent cuts consisting of contemporary events. Note that although the happened-before relation and the older-than relation are very similar, they capture slightly different orders; happened-before relation captures the dependency between the events imposed by the computation while the older-than relation captures the order in which the events are processed by the checker process.

To reduce the number of components for chain decomposition, we extend the happened-before relation by adding some edges from the older-than relation. This allows us to reuse existing chains as the events which were initially concurrent to each other, can be made part of the same chain. We show that this operation is safe i.e. if G was a consistent cut discovered by the checker process in the original poset, then G is also a consistent cut in the new poset. Note that it is important to consider only the consistent cuts which are examined by the checker process as adding new edges would make a previously consistent cut inconsistent.

We define a new relation $\boxdot\!\!\rightarrow$ on the set of events R as the transitive closure of the relation $\rightarrow \cup \circ\!\!\rightarrow$. This relation captures the partial order generated by introducing edges from $\circ\!\!\rightarrow$ in $\rightarrow$. Note that $\boxdot\!\!\rightarrow$ is well defined due to property 1 of $\circ\!\!\rightarrow$ and $\rightarrow$. Since adding edges to poset only reduces the number of consistent cuts, the new relation would not introduce any new consistent cuts that were absent in the original poset. We show that using $\boxdot\!\!\rightarrow$ instead of $\rightarrow$ does not also decrease the set of consistent cuts which are examined.

We define the notion of a *contemporary consistent cut*. A contemporary consistent cut G is a consistent cut in the poset $(R, \rightarrow)$ which satisfies the following property: $\forall e, f \in frontier(G) : e \parallel f$. We show that a contemporary consistent cut in $(R, \rightarrow)$ remains a consistent cut in the new poset $(R, \boxdot\!\!\rightarrow)$.

Lemma 4 *Given $e \in R$ and a contemporary consistent cut G with $e \in frontier(G)$, $\forall f \in G : e \not\circ\!\!\rightarrow f$.*

Proof Suppose $\exists f \in G : e \circ\!\!\rightarrow f$. Let $a \in frontier(G)$ such that $f \underline{\rightarrow} a$. Such an element a must exist due to property that a consistent cut can be defined in terms of the events which happened before the elements in the frontier. Then by property 2 of the $\circ\!\!\rightarrow$ and $\rightarrow$, $e \circ\!\!\rightarrow a$. However, both a and e belong to $frontier(G)$ and so they must be contemporary. This leads to contradiction and hence $\forall f \in G : e \not\circ\!\!\rightarrow f$. □

Lemma 5 *For $a, b \in R$, if $a \circ\!\!\rightarrow b$, then $b \in G$ implies $a \in G$ for every contemporary consistent cut G.*

Proof Consider a contemporary consistent cut G which includes b. Let $e \in R$ such that $e.c = a.c$ and $e \in frontier(G)$. Since $e.c = a.c$, either $e \rightarrow a$ or $a \rightarrow e$. If $e \rightarrow a$, then using property 3 of $\circ\!\!\rightarrow$ and $\rightarrow$, we get $e \circ\!\!\rightarrow b$. This contradicts Lemma 4 and hence, $a \rightarrow e$. Since the consistent cut G must include all events which happened before e, it includes a as well. □

Theorem 7 *Let G be a contemporary consistent cut in poset $(R, \rightarrow)$ with older-than relation $\circ\!\!\rightarrow$. Then G is also a consistent cut in the poset $(R, \boxdot\!\!\rightarrow)$.*

Proof Consider $f \in G$ and an event $e \in R$ such that $e \boxdot\!\!\rightarrow f$. Suppose the path from e to f consists of m edges of the form $a_i \circ\!\!\rightarrow b_i, i = 1 \ldots m$ such that $b_i \underline{\rightarrow} a_{i+1}, i = 1 \ldots m-1$. Then we show that e is present in the consistent cut G. The proof is by induction on m.

Base Case: m = 0 In this case, we have $e \rightarrow f$. By the definition of consistent cut, $e \in G$.

Induction Step Suppose the result holds for $m = s$. Now consider a consistent cut G with the path between e and f consisting of $s+1$ edges from $\circ\!\!\rightarrow$. Consider the events b_s and e. They satisfy $e \boxdot\!\!\rightarrow b_s$ and the path between e and b_s has s edges from $\circ\!\!\rightarrow$. Therefore, using induction hypothesis for $m = s$, e is in any consistent cut which includes b_s. Since $b_{s+1} \underline{\rightarrow} f$, $b_{s+1} \in G$ and by Lemma 5, $a_{s+1} \in G$. Again by the definition of consistent cut, $a_{s+1} \in G \Rightarrow b_s \in G$. Since all consistent cuts having b_s contain e, therefore $e \in G$. □

8.2 Conjunctive Predicate Detection using Contemporary Events

In this section we discuss the modifications that the weak conjunctive predicate (WCP) detection algorithm [19] requires to support chain clocks and the optimization involving contemporary events. A WCP is a conjunction of *local* predicates and we are interested in detecting when the predicate becomes true under the *possibly* modality. The WCP algorithm involves a centralized checker process which maintains a queue of vector timestamps corresponding to every process in the system. Whenever a process finds that its local predicate is true, it sends a timestamp of the event to the checker process. The checker process inserts the timestamp in the queue corresponding to the process. The checker process periodically examines the heads of the queues and checks if there exist two timestamps which are inconsistent. If such a pair is found, the smaller timestamp from the pair is removed. This process is repeated till the checker process cannot find any inconsistent pair of timestamps in which case it has found a consistent cut satisfying the predicate or all the queues become empty and the predicate never becomes true. There are certain optimizations which can be applied to this algorithm to reduce the number of timestamps which need to be reported to the checker process.

In section 7, the local predicates can be generalized to predicates on variables whose accesses form a total order and in our discussion, we would use this generalized notion of local predicates. The relevant events in the system consist of the events which change the value of the local predicate. This idea is very similar to that of interval clocks [1] and in a similar manner, it can be shown that by considering this set of relevant events, we can accurately check if a predicate becomes true in the computation.

With DCC the process responsible for timestamping events also serves as the checker process. This allows the timestamping algorithm to examine the workspace and introduce additional edges based on the older-than relation. The older-than relation used here is the one described in the previous section. The new timestamping algorithm with this modification is given in Figure 18. The algorithm proceeds as the normal DCC algorithm till it is unable to find an existing component which can be reused for the new event e. Before adding a new component, it checks with the workspace if it is possible to use an existing component by adding a new edge between an event deleted from the workspace and e. For this purpose, we maintain a vector D and a list of vectors H. Vector D is the maximum of the timestamp of all the deleted elements. For a component i, $H[i]$ is the timestamp of the deleted event f which satisfied $f.V[i] = D[i]$. If there are more than one such events, then any of them can be used. To add a new edge, we look for a component j in D such that $Z[j] = D[j]$. If the event corresponding to the timestamp $H[j]$ is g, we add an edge from g to e. By Theorem 7, we are guaranteed that such an edge is safe. Adding an edge from g to e amounts to updating $e.V$ to the maximum of $e.V$ and $g.V$ (or $H[j]$). Since $D[j] = Z[j]$, $e.V$ is now guaranteed to have an up-to-date value of the component j and can hence increment that component. The vectors D and the list H are updated whenever an event is deleted from the workspace. On deleting an event e, D is updated to $max(D, e.V)$. If this operation updated the component i of D, then $H[i]$ is set to $e.V$. Here again we need atomicity of operations on the individual components of D, H and Z.

As opposed to the definitions of other GI primitives, this implementation also modifies $e.V$. In case of a shared memory system, this change does not require any modifications to the chain clock algorithm. However, in case when we are using this primitive for a distributed system, we also need to communicate the vector $e.V$ back to the process.

Note that the WCP algorithm itself needs to be modified to use the component chains instead of process chains. An important effect of this change would be that the global state of the system would not be captured completely by just the frontier of the cut. Instead, we would need to keep track of each of the local predicates by remembering the value set by the last (possibly deleted) event to change that predicate. Since all the events which update a local predicate are totally ordered, the last event is well-defined.

```
$GI(V,e)$:: //synchronized
 var
     $Z$: vector of integer
     $F$: vector of integer
     $D$: vector of integer
     $H$: list of vector of integer
         initially ($Z = \phi$, $F = \phi$)

 if $\exists i : F[i] = e.p$
     let $j$ be such that $F[j] = e.p$;
 else
     if $\exists i : Z[i] = V[i]$
         let $j$ be such that $Z[j] = V[j]$;
     else
         if $\exists i : Z[i] = D[i]$
             // Add dependency from deleted node
             let $j$ be such that $Z[j] = D[j]$;
             $V := max(V, H[j])$;
         else
             // Add a new component
             $Z.size{+}{+}$;
             $F.size{+}{+}$;
             $j := Z.size$;
 $Z[j]{+}{+}$;
 $F[j] := e.p$;
 return $j$;
```

Fig. 18 Implementation of GI which uses contemporary event optimization

9 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

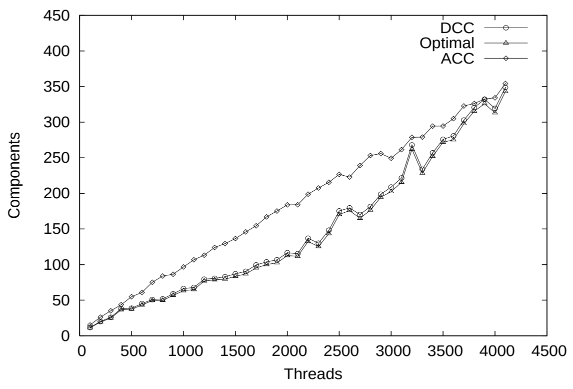

Fig. 19 Components vs Threads

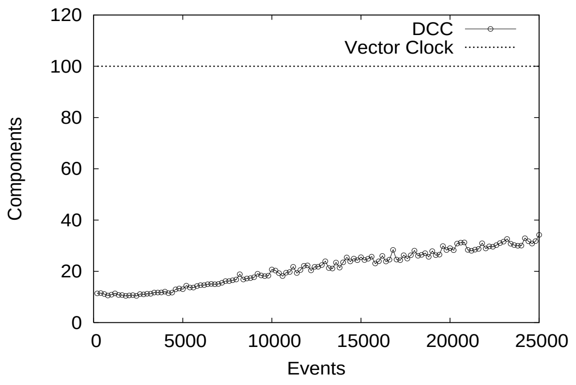

Fig. 21 Components vs Events

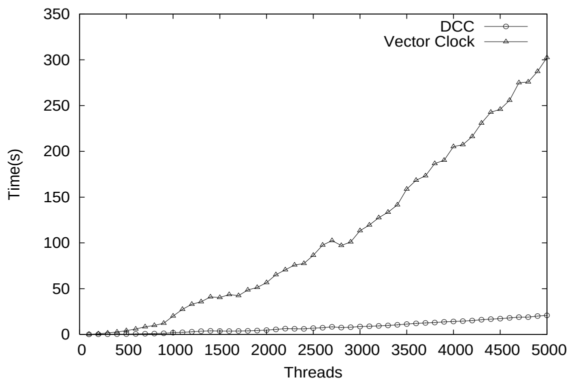

Fig. 20 Time vs Threads

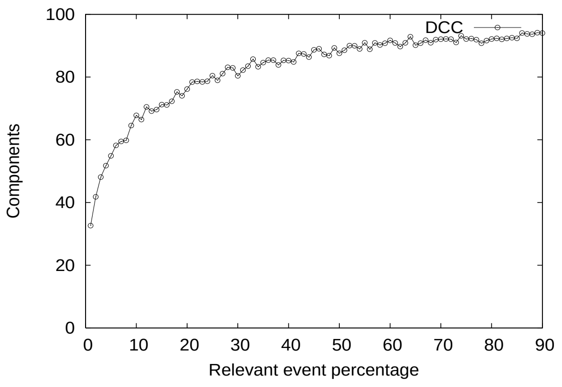

Fig. 22 Components vs Relevant Event Fraction

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 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 19-24.

Figure 19 gives the number of components used by DCC and ACC as N is varied from 100 to 4000. 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 20 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.

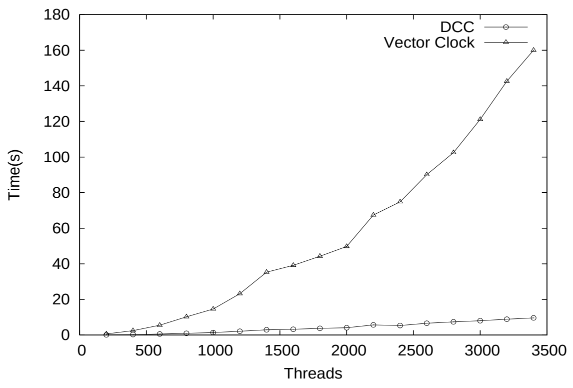

Fig. 23 Time vs Threads with a centralized server

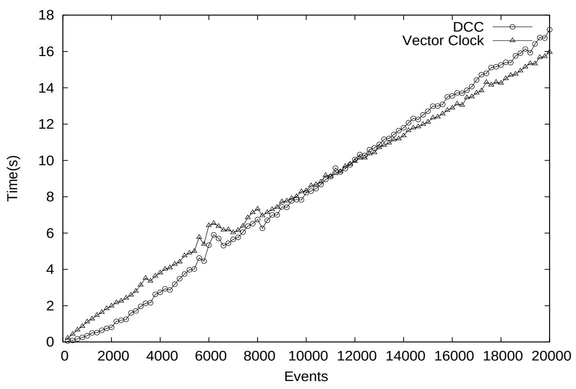

Fig. 24 Time vs Events with a centralized server

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 [33]. However, this optimization makes the assumption of FIFO channels. The system used for performing the reported experiments does not incorporate these optimizations.

In Figure 21, 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 22. 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 23 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 24. 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.

10 Extensions

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

10.1 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.

10.2 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.

10.3 Chain-based plausible clock

Plausible clocks [35] provide an approximation to the order between the events in the computation. A plausible clock based on chain clocks can also be designed in more than one ways. One solution would be to stop adding more components to the chain clock 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 [20] 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. Experiments need to be conducted for comparing the plausible clocks based chains and other plausible clocks.

10.4 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. Representing the clock as a matrix, instead of a vector allows each server to independently add components to their rows and the comparisons to be performed row by 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.

11 Related Work

Certain optimizations have been suggested for the vector clock algorithm which can reduce the bandwidth requirement. For example, in [33] 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 $O(N)$ 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* [35] 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 can track the ordering information accurately with fewer than N components in many cases.

Ward [36] proposed an approach based on *dimension* [11] 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 the 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 [17] which can be done with width-based clocks.

There has also been some work done on bounding the size of the each of the vector clock components so as to reduce the storage requirements for the timestamps. Mostefaoui and Theel [28] presented a solution in which the clocks of all the processes are reset to 0 once a clock reaches a predetermined limit. This solution requires all the processes to block at the time of reset. Arora et. al [2] use a notion of a *contract* which contains guarantees from the application about the events which can be potentially compared and the communication pattern in the computation. The advantage of this approach is that it does not require blocking resets. Some of these ideas may be used for developing a bounded version of chain clocks as well as for reducing the number of components used in chain clocks.

Vector clocks for systems satisfying certain extra properties have also been developed [3], [18]. In contrast our method is completely general. Some logical clocks like weak clocks [24] and interval clocks [1] 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 in many cases.

12 Conclusion

This paper presents a class of timestamping algorithms called chain clocks which track dependency more efficiently than vector clocks. We make four 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 study the problem of optimal online chain decomposition of a poset both with and without process information. We present an efficient algorithm for the problem and prove some new lower bounds for it. Finally, 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.

13 Acknowledgments

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

References

1. Alagar, S., Venkatesan, S.: Techniques to tackle state explosion in global predicate detection. IEEE Transactions on Software Engineering **27**(8), 704 – 714 (2001)
2. Arora, A., Kulkarni, S.S., Demirbas, M.: Resettable vector clocks. Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC) pp. 269–278 (2000)
3. Audenaert, K.: Clock trees: Logical clocks for programs with nested parallelism. IEEE Transactions on Software Engineering **23**(10), 646–658 (1997)
4. Babaoglu, O., Marzullo, K.: Consistent global states of distributed systems: Fundamental concepts and mechanisms. In: S. Mullender (ed.) Distributed Systems, pp. 55–96. Addison-Wesley (1993)
5. Baldoni, R., Melideo, G.: Tradeoffs in message overhead versus detection time in causality tracking. Tech. Rep. 06-01, Dipartimento di Informatica e Sistemistica, Univ. of Rome (2000)
6. Birman, K.P.: The process group approach to reliable distributed computing. Commun. ACM **36**(12), 37–53 (1993)
7. Birman, K.P., Joseph, T.A.: Reliable communication in the presence of failures. ACM Transactions on Computer Systems **5**(1), 47–76 (1987)
8. Cain, H.W., Lipasti, M.H.: Verifying sequential consistency using vector clocks. In: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures, pp. 153–154. ACM Press (2002)
9. Charron-Bost, B.: Concerning the size of logical clocks in distributed systems. Inf. Process. Lett. **39**, 11–16 (1991)
10. Cooper, R., Marzullo, K.: Consistent detection of global predicates. In: Proc. of the Workshop on Parallel and Distributed Debugging, pp. 163–173. ACM/ONR, Santa Cruz, CA (1991)
11. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press (1990)
12. Dilworth, R.P.: A decomposition theorem for partially ordered sets. Annals of Mathematics **51**, 161–166 (1950)
13. Dinning, A., Schonberg, E.: An empirical comparison of monitoring algorithms for access anomaly detection. In: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming, pp. 1–10. ACM Press (1990)
14. Felsner, S.: On-line chain partitions of orders. Theoretical Computer Science **175**, 283–292 (1997)
15. Fidge, C.: Timestamps in message-passing systems that preserve the partial ordering. Proc. of the 11th Australian Computer Science Conference **10**(1), 56–66 (1988)
16. Fowler, J., Zwaenepoel, W.: Causal distributed breakpoints. In: Proc. of the $10th$ International Conference on Distributed Computing Systems, pp. 134–141. IEEE, Paris, France (1990)

17. Garg, V.K., Skawratananond, C.: String realizers of posets with applications to distributed computing. In: 20th Annual ACM Symposium on Principles of Distributed Computing (PODC-00), pp. 72 – 80. ACM (2001)
18. Garg, V.K., Skawratananond, C.: On timestamping synchronous communications. In: 22nd International Conference on Distributed Computing Systems (ICDCS' 02), pp. 552–560. IEEE (2002)
19. Garg, V.K., Waldecker, B.: Detection of unstable predicates. In: Proc. of the Workshop on Parallel and Distributed Debugging. ACM/ONR, Santa Cruz, CA (1991)
20. Gidenstam, A., Papatriantafilou, M.: Adaptive plausible clocks. In: 24th International Conference on Distributed Computing Systems (ICDCS' 04), pp. 86–93. IEEE (2004)
21. Johnson, D.B., Zwaenepoel, W.: Recovery in distributed systems using optimistic message logging and checkpointing. Journal of Algorithms **11**(3), 462–491 (1990)
22. Kierstead, H.A.: Recursive colorings of highly recursive graphs. Canad. J. Math **33**(6), 1279–1290 (1981)
23. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
24. Marzullo, K., Sabel, L.: Efficient detection of a class of stable properties. Distributed Computing **8**(2), 81–91 (1994)
25. Mattern, F.: Algorithms for distributed termination detection. Distributed Computing **2**(3), 161–175 (1987)
26. Mattern, F.: Virtual time and global states of distributed systems. In: Parallel and Distributed Algorithms: Proc. of the Int'l Workshop on Parallel and Distributed Algorithms, pp. 215–226 (1989)
27. Mellor-Crummey, J.: On-the-fly detection of data races for programs with nested fork-join parallelism. In: Proceedings of the 1991 ACM/IEEE conference on Supercomputing, pp. 24–33. ACM Press (1991)
28. Mostefaoui, A., Theel, O.: Reduction of timestamp sizes for causal event ordering. Tech. rep., Irisa (1996)
29. Netzer, R., Miller, B.: Detecting data races in parallel program executions. In: Advances in Languages and Compilers for Parallel Processing, pp. 109–130. MIT Press (1990)
30. Netzer, R.H.B., Miller, B.P.: Optimal tracing and replay for debugging message-passing parallel programs. In: Proceedings of the 1992 ACM/IEEE conference on Supercomputing, pp. 502–511. IEEE Computer Society Press (1992)
31. Raynal, M., Schiper, A., Toueg, S.: The causal ordering abstraction and a simple way to implement it. Inf. Process. Lett. **39**(6), 343–350 (1991)
32. Sen, K., Rosu, G., Agha, G.: Runtime safety analysis of multithreaded programs. In: Proc. of the 11th ACM Symposium on the Foundations of Software Engineering 2003 (2003)
33. Singhal, M., Kshemkalyani, A.: An efficient implementation of vector clocks. Information Processing Letters **43**(10), 47–52 (1992)
34. Strom, R.E., Yemeni, S.: Optimistic recovery in distributed systems. ACM Transactions on Computer Systems **3**(3), 204–226 (1985)
35. Torres-Rojas, F., Ahamad, M.: Plausible clocks: Constant size logical clocks for distributed systems. In: Proc. of 10th Int'l Workshop Distributed Algorithms, pp. 71–88 (1996)
36. Ward, P.A.S.: A framework algorithm for dynamic, centralized dimension-bounded timestamps. In: Proc. of the 2000 conference of the Centre for Advanced Studies on Collaborative research, p. 14 (2000)
37. West, D.B.: The Art of Combinatorics Volume III: Order and Optimization. Preprint edition

A Some additional notation

We introduce some notation that would be used in the analysis of the chain clock algorithm. For an event e, let $\mathcal{A}(e) = \{f \in E \mid f \rightsquigarrow e\}$ 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.

Let $R(i) = \{f \in R \mid f.c = i\}$. We also extend the definition of the history of an element to a per chain history. Let $\mathcal{H}(e,i) = \{f \in R(i) \mid f \rightarrow e\}$. Intuitively, $\mathcal{H}(e,i)$ 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 2) be denoted by $e.W$. Note that in the case of relevant events, this value will be different from the timestamp to 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.

B 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 $\rhd_i$, as follows: $e, f \in R(i) : e \rhd_i f$ if the call $G(V,e)$ accessed $Z[i]$ before $G(V,f)$. Since the accesses to $Z[i]$ are totally ordered, $\rhd_i$ is a total order. The order $\rhd_i$ 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 $e,f \in R(i): e \to f \Rightarrow e \rhd_i f$. For $e \in R(i)$, we define $L(e,i) = \{f | (f \in R(i)) \wedge (f \rhd_i e)\}$ as the set of events which incremented i^{th} component before e. Clearly, $\mathcal{H}(e,i) \subseteq L(e,i)$ as $\rhd_i$ is consistent with $\to$.

The following lemma is a direct consequence of the algorithm in Figure 3.

Lemma 6 *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 $v = |\{f | (f \in R(i)) \wedge (f \rhd_i e)\}|$. □

The chain decomposition property is equivalent to showing that set of events $R(i)$ forms a chain and the following theorem proves that the implementation of GI in Figure 3 coupled with the implementation of the vector clock algorithm in Figure 2 satisfies it.

Theorem 8 *The implementation of primitive GI in Figure 3 with the vector clock algorithm in Figure 2 satisfi es the following:*

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

Proof We prove the theorem by induction on $k = rank(e)$ in the following statements:
(I1) $\forall e \in E, \forall i : 1 \le i \le s : e.W[i] = |\mathcal{H}(e,i)|$
(I2) $\forall e,f \in R(i) : (rank(f) \le rank(e)) \Rightarrow f \to e$
(I3) $\forall e,f \in E : rank(f) \le rank(e)$
$\Rightarrow (\mathcal{H}(f,i) \subseteq \mathcal{H}(e,i)) \vee (\mathcal{H}(e,i) \subseteq \mathcal{H}(f,i))$
Base case ($k = 0$): (I1) If $k = 0$, then e is an initial event and V is initialized to $\overrightarrow{0}$. So $\forall i : 1 \le i \le s : e.W[i] = 0$. Since e is an initial event, so $\forall i : 1 \le i \le s : \mathcal{H}(e,i) = \phi$. Therefore, $e.W[i] = |\mathcal{H}(e,i)|$.
(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 3, $f.c \neq e.c$. This leads to a contradiction. So, $\nexists f \in R : (f.c = e.c) \wedge (rank(f) \le rank(e))$. 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 $\mathcal{H}(e,i) = \mathcal{H}(f,i) = \phi$. So $\mathcal{H}(e,i) \subseteq \mathcal{H}(f,i)$.

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

Case 2: $\exists d \in \mathcal{A}(e)$ such that $d.c = i$:
$\{d_1, d_2 \in \mathcal{A}(e) : d_1 \parallel d_2$ and IH for (I2)$\}$
$\nexists d_1, d_2 \in \mathcal{A}(e) : (d_1 \neq d_2) \wedge (d_1.c = d_2.c)$
Let x be the unique event such that $x \in \mathcal{A}(e) : x.c = i$.
{ Definition of $\mathcal{H}(e,i)$ and IH for (I2)}
$\Rightarrow \forall f \in \mathcal{H}(e,i) : f \underline{\to} x$
{ IH for (I3) }
$\Rightarrow \mathcal{H}(x,i) \supseteq \mathcal{H}(d,i), d \in \mathcal{A}(e)$
{ IH for (I1) }
$\Rightarrow x.W[i] \ge d.W[i], d \in \mathcal{A}(e)$
$\{x.V[i] = x.W[i] + 1 \wedge d.V[i] = d.W[i], d \neq x \in \mathcal{A}(e)\}$
$\Rightarrow x.V[i] > d.V[i], d \neq x \in \mathcal{A}(e)$
{ Definition of max }
$\Rightarrow x.V[i] = max\{d.V[i] | d \in \mathcal{A}(e)\}$
{ By algorithm and IH for (I1) }
$\Rightarrow e.W[i] = |\mathcal{H}(x,i)| + 1$
{ Using $\mathcal{H}(e,i) = x \cup \mathcal{H}(x,i)$ }
$\Rightarrow e.W[i] = |\mathcal{H}(e,i)|$
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, $|L(e,i)| = |\mathcal{H}(e,i)|$. Since $L(e,i) \subseteq \mathcal{H}(e,i)$, this implies $L(e,i) = \mathcal{H}(e,i)$ and so $f \in \mathcal{H}(e,i)$ and hence $f \to e$. Now assume $e \rhd_i f$. If $rank(f) \le rank(e)$, then $e \not\to f$. Therefore, suppose $e \parallel f$. Then using the same reasoning as above, we get $e \in \mathcal{H}(f,i)$ which gives us a contradiction. Therefore, $f \to e$.

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

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

C 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 $\mathcal{H}(e,i)$ 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 7 $\forall e,f \in E : (\mathcal{H}(f,i) \subseteq \mathcal{H}(e,i)) \vee (\mathcal{H}(e,i) \subseteq \mathcal{H}(f,i))$

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

Lemma 8 *For an event $e \in E$, $\forall i : e.W[i] = |\mathcal{H}(e,i)|$.*

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 7 and given (I2) and (I3), we can prove (I1) which is the desired result. □

Lemma 9 *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 10 *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 $\mathcal{H}(e,f.c)$ and $\mathcal{H}(f,f.c)$. By Lemma 7, $\mathcal{H}(e,f.c) \subseteq \mathcal{H}(f,f.c)$ or $\mathcal{H}(f,f.c) \subseteq \mathcal{H}(e,f.c)$. Suppose $\mathcal{H}(e,f.c) \supset \mathcal{H}(f,f.c)$. Then, $\mathcal{H}(e,f.c) \supseteq \mathcal{H}(f,f.c) \cup \{f\}$ because of the chain decomposition property and definition of $\mathcal{H}(e,i)$. This implies $f \rightarrow e$ which contradicts the assumption. So $\mathcal{H}(e,f.c) \not\supset \mathcal{H}(f,f.c)$ and therefore, $\mathcal{H}(e,f.c) \subseteq \mathcal{H}(f,f.c)$. By Lemma 8, $e.W[f.c] \leq f.W[f.c]$. 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 9 *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 9, $e.V \leq f.V$. Moreover, Lemma 10 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 10. □

D Proof for Theorem 5

Let the chains of the poset produced by decomposition be C_i and $top(i)$ be the maximal element of the chain C_i. If x is the maximal element of the poset, then $private(x)$ is the set of chains C_i such that $top(i) \leq x$ and $top(i) \nleq y$ for all maximal elements $y \neq x$. The process to which an element e belongs is denoted by $p(e)$ and similarly for a set of elements A, $p(A)$ denotes the set of processes to which the elements in A belong. In particular, when we say $p(private(x))$, it implies $\bigcup p(top(i))$ where $C_i \in private(x)$. Similarly $top(private(x))$ is the set $\bigcup top(i)$, $C_i \in private(x)$.

Induction Hypothesis : For every positive integer k with $\binom{k+1}{2} \leq N$, there is a strategy $S(k)$ for Bob so that the poset P presented so far is of width k, has exactly k maximal elements and uses elements from only $\binom{k+1}{2}$ processes. Moreover, the maximal elements can be numbered $x_1,\ldots,x_k$ such that for all i, $|private(x_i)| \geq i$
and $|p(private(x_i)| = i$.

Base Case : For $k = 1$, we use one element from process 1. The hypothesis holds for this case.

Induction Step : Suppose we have strategy $S(k)$. We construct $S(k+1)$ using $S(k)$ as follows:

1. Run strategy $S(k)$. This phase ends with an order Q_1 with maximal elements $x_1,\ldots,x_k$, $|private(x_i)| \geq i$ and number of processes used $\binom{k+1}{2}$.
2. Run strategy $S(k)$ again. This time every new element is made greater than each of $x_1,\ldots,x_{k-1}$ and their predecessors but incomparable to rest of the elements in Q_1. In particular, the new elements are incomparable to elements in $top(i)$ for $C_i \in private(x_k)$. For constructing this new $S(k)$, we reuse the processes $p(Q_1) \setminus p(private(x_k))$ and add a set of k new processes. The important observation here is that for $C_i \in private(x_k)$, there does not exist any element $f \in Q_1 \setminus top(private(x_k))$ such that $top(i) < f$. This effectively implies that process $p(top(i))$ cannot be used for the new $S(k)$.
This phase ends with an order Q_2 with $k+1$ maximal elements $y_1,\ldots,y_k,x_k$. At this point, there are at least i chains in $private(y_i)$ and an additional k chains in $private(x_k)$. Similarly, at these point we have made use $\binom{k+2}{2} - 1$ processes.

3. Add a new element z so that z is greater than all elements of Q_2 and $p(z) = p(y_1)$. For the chain C_i to which z is assigned, it holds that $i \notin private(x_k)$ or $i \notin private(y_k)$. Wlog assume that $i \notin private(x_k)$. Now $private(z)$ has chains from x_k and the chain to which z belongs. So $|private(z)| \geq k+1$ and $|p(private(z))| = k+1|$. We refer to z as z_{k+1} from now on.
4. In this final phase, run strategy $S(k)$ again with all new elements greater than $y_1, \dots, y_k$. For this phase, we use the processes $p(Q_2) \setminus (p(private(x_k)) \cup p(y_1))$ and add a new process to the system. This way we again have $\binom{k+1}{2}$ processes available for this phase without violating process semantics. This phase ends with maximal elements $z_1 \dots, z_{k+1}$ so that $|private(z_i)| \geq i$ and $|p(private(z_i))| = i$.

During all the four phases we added $k+1$ new processes to the system and hence the total number of processes used till now is $\binom{k+1}{2} + (k+1) = \binom{k+2}{2}$. This completes the proof of the theorem.

Anurag Agarwal received his B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Kanpur in 2003. He is currently a student in the Department of Computer Science at the University of Texas, Austin. His research interests include distributed systems, program verification and security.

Vijay K. Garg received his B.Tech. degree in computer science from the Indian Institute of Technology, Kanpur in 1984 and M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California at Berkeley in 1985 and 1988, respectively. He is currently a full professor in the Department of Electrical and Computer Engineering and the director of the Parallel and Distributed Systems Laboratory at the University of Texas, Austin. His research interests are in the areas of distributed systems and discrete event systems. He is the author of the books Concurrent and Distributed Computing in Java (Wiley & Sons, 2004), Elements of Distributed Computing (Wiley & Sons, 2002), Principles of Distributed Systems (Kluwer, 1996) and a co-author of the book Modeling and Control of Logical Discrete Event Systems (Kluwer, 1995). Prof. Garg is also an IEEE Fellow.