Self-Organizing Hierarchical Cluster Timestamps

Paul A.S. Ward and David J. Taylor*

Shoshin Distributed Systems Group, Department of Computer Science, University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
{pasward,dtaylor}@shoshin.uwaterloo.ca

Abstract. Distributed-system observation tools require an efficient data structure to store and query the partial-order of execution. Such data structures typically use vector timestamps to efficiently answer precedence queries. Many current vector-timestamp algorithms either have a poor time/space complexity tradeoff or are static. This limits the scalability of such observation tools. One algorithm, centralized hierarchical cluster timestamps, has potentially a good time/space tradeoff provided that the clusters accurately capture communication locality. However, that algorithm, as described, uses pre-determined, contiguous clusters. In this paper we extend that algorithm to enable a dynamic selection of clusters. We present experimental results that demonstrate that our extension is more stable with cluster size and provides timestamps whose average size is consistently superior to the pre-determined cluster approach.

1 Motivation

Tools for distributed-system observation and control, such as ATEMPT [9,10], Object-Level Trace [6], and POET [11], can be broadly described as having the architecture shown in Fig.1. The distributed system is instrumented with monitoring code that captures process identifier, event number and type, and partner-event information. This information is forwarded from each process to a central monitoring entity which, using this information, incrementally builds and maintains a data structure of the partial order of events that form the computation [12]. That data structure may be queried by a variety of systems, the most common being visualization engines and control entities, which in turn may be used to control the distributed computation. It is the efficient representation of this partial-order data structure that is the focus of this paper.

Systems we are aware of maintain the transitive reduction of the partial order, typically accessed with a B-tree-like index. This enables the efficient querying of events given a process identifier and event number. It does not, however, enable efficient event-precedence querying, which is one of the most common query types on such structures. To enable efficient precedence testing, the Fidge/Mattern vector timestamp [1,14] is computed for each event and stored with that event

* The authors would like to thank IBM for supporting this work.

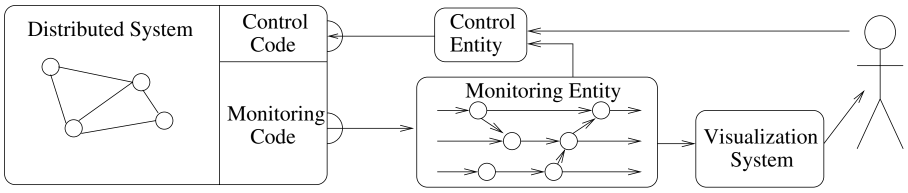


Fig. 1. Distributed-System Observation and Control

in the data structure (Note that for distributed-system observation, these timestamps are computed centrally in the monitoring entity, rather than within the distributed computation.). The choice of the Fidge/ Mattern timestamp is dictated by its ability to answer precedence queries in constant time and the fact that it is dynamic (that is, computable without requiring complete knowledge of the event set). Unfortunately, the Fidge/Mattern timestamp requires a vector of size equal to the number of processes in the computation. Such a requirement does not allow the data structure to scale with the number of processes.

Recently we proposed centralized hierarchical cluster timestamps [17]. These timestamps are also dynamic and can efficiently answer precedence queries, but require up to order-of-magnitude less space than do Fidge/Mattern timestamps. The drawback with such cluster timestamps is that the space consumption is heavily dependent on whether or not the clusters used accurately capture locality of communication. If they do not, then the space-consumption saving is substantially impaired. The algorithm described in that paper uses pre-determined, contiguous clusters. That is, the clusters represent a continuous range of the Fidge/Mattern timestamp. The results show a significant variation in average cluster-timestamp size with small variations in cluster choice. In this paper, we address this problem by extending that algorithm to enable a dynamic selection of clusters. We call this new algorithm self-organizing cluster timestamps. The actual method of selecting clusters is orthogonal to our timestamp algorithm extension, though it must be made dynamically, which substantially limits the choice of clustering method. For our experimental analysis we use a relatively simple dynamic clustering based on first communication.

In the remainder of this paper we first briefly review related work. In Sect.3 we will describe our extension to the hierarchical cluster algorithm to provide self-organizing clusters. We do this in two steps, first dealing with a flat collection of non-overlapping clusters, and then extending it to an arbitrary hierarchy of clusters. We then present experimental results that demonstrate the value of our algorithm.

2 Related Work

The centralized hierarchical cluster timestamp [17] is an attempt to reduce the space requirement of vector timestamps while still providing efficient precedence determination. It is based on two observations. First, events within a cluster can only be causally dependent on events outside that cluster through receive events from transmissions that occurred outside the cluster. Such events are called "cluster receives." By identifying such cluster-receive events, it is possible to shorten the timestamp of all other events within the cluster to the number of processes in the cluster. Second, in many parallel and distributed computations, most communication of most processes is with a small number of other processes. If the clusters capture this communication locality, then there should be few cluster-receive events, and so the average space-consumption per timestamp will be significantly less than with the Fidge/Mattern timestamp. Unfortunately, the algorithm as described in [17] does nothing to select good clusters. Rather it uses pre-determined, contiguous clusters. The contribution of this paper is to extend that algorithm to enable the dynamic selection of clusters, and to show experimentally that this improves the stability of the timestamp algorithm, even when using a rather poor dynamic clustering algorithm.

There are various other approaches that have been taken to the problem of reducing the size of vector timestamps. Fowler and Zwaenepoel [3] create direct-dependency vectors. While these vectors can be substantially smaller than Fidge/Mattern timestamps, precedence testing requires a search through the vector space, which is in the worst case linear in the number of messages. Jard and Jourdan [7] generalize the Fowler and Zwaenepoel method, but have the same worst-case time bound. Singhal and Kshemkalyani [15] take the approach of transmitting just the information that has changed in the vector between successive communications. While not directly applicable in our centralized context, it is possible to use a differential technique between events within the partial-order data structure. However, when we evaluated such an approach we were unable to realize more than a factor of three in space saving. Ward [16] has an approach based on Ore timestamps [13], though it is only applicable to low-dimension computations. There is also some amount of work on dynamic transitive closure (e.g. [8]), though the problem being addressed is more general than is our problem, and the results are not as good for our application.

3 Self-Organizing Cluster Timestamps

We now describe our algorithm in two steps. We start by extending the two-level cluster timestamp to incorporate self-organizing clusters. We then merge this with the arbitrary-depth hierarchical cluster algorithm to produce self-organizing hierarchical cluster timestamps.

3.1 Self-Organizing Two-Level Clusters

Our extension to the two-level cluster timestamp to incorporate self-organizing clusters required several changes. The core change is that clusters are no longer pre-determined, contiguous entities. Rather, each process is defined as initially belonging to a cluster of size one. When timestamping an event that would be

```
timestamp(e) {
  e.FM = Fidge/Mattern(e);
  if (e.type == RECEIVE && e.cluster().inverseMap[e.partner().process] == -1
          && !mergable(e, e.partner()) {
    e.clusterTimestamp = e.FM;
    GCR[e.process] = e.clusterReceive = e.eventNumber;
  } else {
    if (e.type == RECEIVE && e.cluster().inverseMap[e.partner().process] == -1)
      mergeClusters(e, e.partner());
    for (j = 0 ; j < e.cluster().size ; ++j)
      e.clusterTimestamp[j] = e.FM[e.cluster().map[j]];
    e.clusterReceive = GCR[e.process];
  }
  Clean up Fidge/Mattern timestamps;
}
```

Fig. 2. Two-Level Self-Organizing Cluster Timestamps

a cluster receive (that is, the event is a receive event with a partner transmit event that occurred in a different cluster), the algorithm first determines if the two clusters are mergable. If they are, then they are merged, and the event is given the shorter cluster timestamp. Only if they are not mergable is the event left with its Fidge/Mattern timestamp. Insofar as the dynamic clustering algorithm captures communication locality patterns, future communication is also in-cluster, and will likewise receive the shorter cluster timestamp.

The determination of the mergability of clusters is an orthogonal function in our timestamping algorithm, and not the core subject of this paper. However, there is an implicit constraint on acceptable clustering methods. Specifically, the method must be relatively dynamic. It cannot look at more than a few events within a couple of clusters before deciding that the clusters are mergable. The reason is that after an event is timestamped, its timestamp will never change. Therefore, if clusters are to merge, the sooner the merger occurs, the less false cluster-receive events there will be (that is, events that are timestamped as cluster receives, since the merger had not occurred at the time the event was timestamped.). This substantially constrains the possible clustering methods, since most clustering algorithms must look at the entire data set, or at least a randomly selected sample of it, and require multiple iterations rather than clustering in one pass.

For our experimental analysis we used a simple greedy heuristic of merging two clusters whenever processes in the clusters communicate, subject to an upper bound on cluster size. This is probably a worst-case dynamic clustering algorithm, and so insofar as it provides good results, better clustering methods will only improve the results.

The timestamp-creation algorithm is shown in Fig.2. Before we describe the algorithm, a few notation points will help the reader understand the code. Each event is an object in the event class which contains, among other things, the event number (e.eventNumber), process identifier (e.process), a reference to the event's greatest preceding cluster receive within the process (e.clusterReceive), a pointer to the FidgeMattern timestamp (e.FM), if it exists, and a pointer to the cluster timestamp (e.clusterTimestamp). The event class also contains two important methods: cluster() and partner(). The cluster() method returns a reference to the cluster that the process belonged to at the time the event was timestamped. We note at this point that we have designed our algorithm such that the cluster information for any cluster will be maintained as it was at the time an event was timestamped even after cluster mergers. We will explain this point in more detail when we describe the cluster merging algorithm of Fig.3. The partner() method returns a reference to the event's partner event, if any.

Cluster information is maintained in an array of clusters, one for each process in the initial instance. The information that a cluster maintains is its current size, a mapping from cluster timestamp vector locations to process identifiers (this is, a simple array indicating which process each vector element corresponds to), and an inverse mapping from process identifiers to cluster timestamp vector locations (this is, a hash array, since most processes will not have entries in a cluster timestamp). Information pertaining to the merging of clusters is maintained by an array of lists of merge points, which indicate at what event a given cluster merged with another cluster, and what cluster it merged with. As with the original cluster timestamp algorithm, this algorithm maintains a vector of all greatest cluster-receive events in all processes, referred to as GCR, initialized to 0. Process identifiers range from 0 to $N-1$, as do cluster identifiers. Events are numbered starting at 1.

The algorithm first computes the Fidge/Mattern timestamp. For events that are non-mergable cluster receives, it simply uses the Fidge/Mattern timestamp, and adjusts the vector of greatest cluster receives to identify this new event. The determination of mergability, as we have noted, is an orthogonal issue.

For events that are mergable cluster receives, we merge the clusters. This is done as shown in Fig.3. Note that the process for merging consists of appending to the mapping and inverse mapping vectors of the merged-into cluster those values from the merged-from cluster (lines 3 and 4). This ensures that all events in the merged-into cluster that were timestamped prior to the merger will still see the same mapping and inverse mapping data as was present at the time the events were timestamps. In addition, the processes in the merged-from cluster must for future events be identified as belonging to the merged-into cluster (lines 5 to 8).

After merging clusters, if required, the timestamp algorithm sets the cluster timestamp to the projection of the Fidge/Mattern timestamp over the cluster processes. This is sufficient to determine precedence within the cluster. To enable precedence determination beyond the cluster, it also records the greatest preceding cluster receive in the event's process. Finally, the algorithm deletes Fidge/Mattern timestamps that are no longer needed. The computation cost of

```
mergeClusters(e,f) {
   for (i = 0 ; i < f.cluster().size ; ++i) {
      e.cluster().map[e.cluster().size + i] = f.cluster().map[i];
      e.cluster().inverseMap[f.cluster().map[i]] = e.cluster()size + i;
      clusterMerges[f.cluster().map[i]][CMP[f.cluster().map[i]]].event =
         maxEvent[f.cluster().map[i]] + 1;
      clusterMerges[f.cluster().map[i]][CMP[f.cluster().map[i]]].cluster = e.process;
      ++CMP[f.cluster().map[i]];
   }
   e.cluster().size += f.cluster().size;
}
```

Fig. 3. Two-Level Cluster Merging

```
 1: precedes(e,f) {
 2:    if (f.cluster().inverseMap[e.process] != -1)
 3:       return (e.clusterTimestamp[e.cluster().inverseMap[e.process]] <
 4:                f.clusterTimestamp[f.cluster().inverseMap[e.process]]);
 5:    else {
 6:       for (j = 0 ; j < f.cluster().size ; ++j) {
 7:          g = f.clusterTimestamp[j] - (f.process == f.cluster().map[j] ? 0 : 1);
 8:          r = event(f.cluster().map[j],g).clusterReceive;
 9:          if (e.clusterTimestamp[e.cluster().inverseMap[e.process]] <
10:             event(f.cluster().map[j],r).FM[e.process]);
11:             return true;
12:       }
13:    }
14:    return false;
15: }
```

Fig. 4. Two-Level Self-Organizing Timestamp Precedence Test

this algorithm is $O(N)$, where N is the number of processes. This cost arises because of the need to compute the Fidge/Mattern timestamp in line 2.

The precedence-test algorithm for the two-level self-organizing cluster timestamp is shown in Fig.4. The algorithm first determines if event f's cluster contains the process that contains event e (line 2). This is a simple matter of checking f's cluster's inverse mapping for e's process identifier. A -1 is returned from the hash if the value is not found. If f's cluster contains e, the algorithm applies the Fidge/Mattern test, using the inverse mapping function to select the correct elements of the vectors (lines 3 and 4). If the events are in different clusters, then the algorithm needs to determine if there is a cluster receive causally prior to f that is a successor of e. The test computes this by determining f's set of greatest preceding cluster receives (lines 7 and 8) and checking each in turn to see if e is prior to it (lines 9 and 10). The computation cost of this test is constant for events within the same cluster and O(f.cluster().size) between clusters.

```
 1: timestamp(e) {
 2:    e.FM = Fidge/Mattern(e);
 3:    k = 1;
 4:    while (e.type == RECEIVE && e.cluster(k).inverseMap[e.partner().process] ==
-1
 5:              && !mergable(k, e, e.partner())) {
 6:       e.clusterReceive[k] = GCR[k][e.process] = e.eventNumber;
 7:       ++k;
 8:    }
 9:    if (e.type == RECEIVE && e.cluster(k).inverseMap[e.partner().process] == -1)
10:       mergeClusters(k, e, f);
11:    e.clusterReceive[k] = GCR[k][e.process];
11:    for (j = 0 ; j < e.cluster(k).size ; ++j)
12:       e.clusterTimestamp[j] = e.FM[e.cluster(k).map[j]];
13:    Clean up Fidge/Mattern timestamps;
14: }
```

Fig. 5. Hierarchical Dynamic Cluster Timestamping

3.2 Hierarchical Self-Organizing Clusters

We now merge the two-level self-organizing algorithm with the hierarchical cluster algorithm of [17]. The resulting algorithm is presented in Fig.5. Clusters are now organized in a hierarchy and assigned a level, starting at 0, which is the innermost cluster. At the highest level is a cluster that comprises the entire computation. The event class now has a cluster(k) method which returns a reference to the level-k cluster that the process belonged to at the time the event was timestamped. The cluster class itself remains unaltered, though the array of clusters now becomes an array of levels of clusters. The cluster-merging data structures likewise have an additional parameter for cluster level, but are otherwise unaltered.

As with the two-level algorithm, the starting point is computing the Fidge/ Mattern timestamp of the event (line 2). The **while** loop of lines 4 to 8 then identifies either the cluster in which both events reside, or two clusters, at level-k in the hierarchy, that should be merged. In the event of the latter, we perform essentially the same cluster merge operation as for the two-level algorithm, but adjust the cluster method calls to include the value k, so as to merge the correct clusters. Note also that the algorithm can no longer maintain just one GCRvector, but must maintain such a vector for every cluster level, per the algorithm of [17]. Other than these points, the algorithm is identical to the two-level algorithm of Fig.2. The computation cost of this timestamp-creation algorithm is $O(N)$ as it requires the calculation of the Fidge/Mattern timestamp for each event.

The precedence-test algorithm, shown in Fig.6 is essentially identical to that of hierarchical cluster timestamps. The primary differences are the same as those that occurred in the two-level algorithm, except the cluster method calls now require a cluster level. The computation cost of this test is iden-

```
precedes(e,f) {
   if (f.cluster(1).inverseMap[e.process] == -1)
      return (e.clusterTimestamp[e.cluster(1).inverseMap[e.process]] <
               f.clusterTimestamp[f.cluster(1).inverseMap[e.process]]);
   k = 1;
   currentTimestamp = f.clusterTimestamp;
   while (f.cluster(k).inverseMap[e.process] == -1 {
      newTimestamp = 0;
      for (j = 0 ; j < f.cluster(k).size ; ++j) {
         g = currentTimestamp[j] - (f.process == f.cluster(k).map[j] ? 0 : 1);
         r = event(f.cluster(k).map[j],g).clusterReceive[k];
         newTimestamp = max(event(f.cluster(k).map[j],r).,newTimestamp);
      }
      ++k;
      currentTimestamp = newTimestamp;
   }
   for (j = 0 ; j < f.cluster(k).size ; ++j) {
      g = currentTimestamp[j] - (f.process == f.cluster(k).map[j] ? 0 : 1);
      r = event(f.cluster(k).map[j],g).clusterReceive[k];
      if (e.clusterTimestamp[e.cluster(1).inverseMap[e.process]] <
          event(f.cluster(k).map[j],r).clusterTimestamp[e.cluster(k+1)
                                                                   .inverseMap[e.process]]);
         return true;
   }
   return false;
}
```

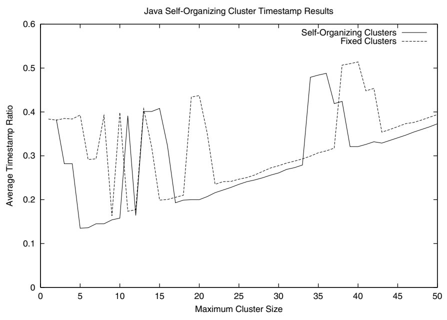
Fig. 6. Hierarchical Cluster Timestamp Precedence Test

tical to that of the original hierarchical cluster timestamp algorithm, being O(f.cluster(k-2).size f.cluster(k-1).size) in general, where the events are in the same level-k cluster. It is O(f.cluster(1).size) if the events are in the same level-1 cluster, and constant if they are in the same level-0 cluster.

4 Experimental Results

We have evaluated our algorithms over several dozen distributed computations covering a variety of different environments, including Java [5], PVM [4] and DCE [2], with up to 300 processes. The PVM programs tended to be SPMD style parallel computations. As such, they frequently exhibited close neighbour communication and scatter-gather patterns. The Java programs were web-like applications, including various web-server executions. The DCE programs were sample business application code.

For our experiments we compared the space requirements for three algorithms: Fidge/Mattern timestamps, pre-determined contiguous cluster timestamps, and self-organizing cluster timestamps. The clustering method used



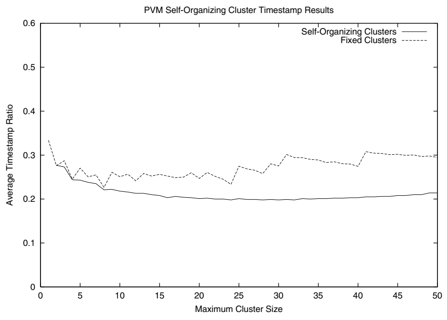


Fig. 7. Ratio of Cluster to Fidge/Mattern Timestamp Sizes

for the self-organizing clusters was to merge clusters on first communication, up to a maximum cluster size. The form of the experiments was to vary the cluster size (maximum cluster size in the case of the self-organizing clusters) from 1 to 50 and observe the ratio of the average cluster timestamp size to the Fidge/Mattern timestamp size. We presumed that the observation tool knew the number of processes in the computation, and encoded the Fidge/Mattern timestamp as a vector of that size.

Sample results for Java and PVM respectively are shown in Fig.7. We can see from these results that the primary objective of this work has been largely achieved. The self-organizing cluster timestamps typically require less space than do the pre-determined cluster timestamps. Of far greater significance, however, is the fact that the self-organizing clusters are far more stable in their space-consumption requirement than are the pre-determined clusters. Note that the Java example demonstrates a size ratio of between 12 and 15 percent of Fidge/Mattern over a range of maximum cluster size from 5 to 10. By contrast, the pre-determined cluster algorithm experienced a size ratio of between 15 and 40 percent of Fidge/Mattern over the same range of cluster size. While there are still spikes in the Java results, they are less frequent, and more easily avoided by our observation tool. Likewise, in the PVM case, the size requirement is both lower, and the range of cluster size in which average cluster timestamp size is within a small factor of that minimum is much larger.

This effect of more consistent and slightly better space-consumption was observed over almost all experiments that we ran. In no instance did our self-organizing cluster timestamps achieve worse results than pre-determined clusters. In the cases where the results were no better, it was readily observed that the pre-determined clusters happened to be a good selection. This typically happened in PVM programs where the communication was extremely regular and coincided with the pre-determined clusters. It should be noted that had the communication been regular but counter to the pre-determined clusters, then the pre-determined cluster approach would have produced very poor results. The self-organizing clusters do not suffer from this problem. Indeed, this is precisely what they overcome.

The full raw result information for our experiments are available on our web site at `http://www.shoshin.uwaterloo.ca/~pasward/ClusterTimestamp`.

5 Conclusions and Future Work

We have presented an algorithm for self-organizing hierarchical cluster timestamps. These timestamps capture the communication locality that is required for cluster timestamps to be effective. We have presented experimental results that demonstrate that our algorithm is more stable with cluster size and provides timestamps whose average size is consistently superior to a pre-determined cluster approach, even with a poor clustering algorithm.

There are several areas of future work that we are actively exploring. First, we are looking at alternate, more sophisticated, dynamic clustering algorithms. Specifically, we are developing a clustering algorithm that observes several communication events before deciding on the suitability of a cluster merger. The tradeoff is an increase in the number of false cluster-receive events. Second, and in a similar vein, we are extending this work to enable dynamically reconfigurable clusters. This is needed when communication patterns are not stable in long running computations or to correct mistakes made by the dynamic clustering algorithm. Third, we are looking at a technique to allow for overlapping clusters at the same level in the hierarchy. This may be a significant benefit when neighbouring processes communicate. Specifically, while it increases the cluster vector size slightly, it can reduce the precedence-test time by enabling in-cluster testing.

Fourth, we plan to more thoroughly explore the tradeoffs in cluster size and arrangement for the hierarchical algorithm with three or more cluster levels. This is a complex tradeoff between average timestamp size, timestamp-creation time, precedence-test time, and greatest-predecessor-set computation time.

Finally, we are investigating the integration of this timestamp technique with the POET system. The primary issue is that the POET system is based on fixed-size FidgeMattern timestamps, with their distributed-precedence-testing capability. The implications of variable-size timestamps requiring centralized-timestamp testing are unclear.

References

1. Colin Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991. 46
2. Open Software Foundation. *Introduction to OSF/DCE*. Prentice-Hall, Englewood Cliffs, New Jersey, 1993. 53
3. Jerry Fowler and Willy Zwaenepoel. Causal distributed breakpoints. In *Proceedings of the 10th IEEE International Conference on Distributed Computing Systems*, pages 134–141. IEEE Computer Society Press, 1990. 48
4. Al Geist, Adam Begulin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachusetts, 1994. 53
5. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996. Available at http://java.sun.com/docs/books/jls/. 53
6. IBM Corporation. IBM distributed debugger for workstations. Online documentation available at: `http://www-4.ibm.com/software/webservers/appserv/doc/-v35/ae/infocenter/olt/index.html`. 46
7. Claude Jard and Guy-Vincent Jourdan. Dependency tracking and filtering in distributed computations. Technical Report 851, IRISA, Campus de Beaulieu – 35042 Rennes Cedex – France, August 1994. 48
8. Valerie King and Garry Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing*, pages 492–498. ACM, 1999. 48
9. Deiter Kranzlmüller, Siegfried Grabner, R. Schall, and Jens Volkert. ATEMPT — A Tool for Event ManiPulaTion. Technical report, Institute for Computer Science, Johannes Kepler University Linz, May 1995. 46
10. Dieter Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, GUP Linz, Linz, Austria, 2000. 46
11. Thomas Kunz, James P. Black, David J. Taylor, and Twan Basten. POET: Target-system independent visualisations of complex distributed-application executions. *The Computer Journal*, 40(8):499–512, 1997. 46
12. Leslie Lamport. Time, clocks and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, 1978. 46
13. Oystein Ore. *Theory of Graphs*, volume 38. Amer. Math. Soc. Colloq. Publ., Providence, R.I., 1962. 48
14. Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994. 46
15. M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43:47–52, August 1992. 48
16. Paul A.S. Ward. A framework algorithm for dynamic, centralized dimension-bounded timestamps. In *Proceedings of the 2000 CAS Conference*, November 2000. 48
17. Paul A.S. Ward and David J. Taylor. A hierarchical cluster algorithm for dynamic, centralized timestamps. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems*. IEEE Computer Society Press, 2001. 47, 48, 52