A Dynamic Cluster-timestamp Creation Algorithm for Distributed-System Management

by

Tao Huang

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2005

ABSTRACT

The distributed-system observation tool is a key component in distributed-systems management. The partial-order data structure is the key data structure used in the distributed-system observation tool. It typically uses vector timestamps to answer efficiently event-precedence queries. However, all current vector-timestamp algorithms either require the size of the vector timestamp to be equal to the total number of processes in the distributed system or require unacceptable space consumption to construct a graph in order to determine event precedence by searching this graph. Ward created a cluster timestamp algorithm which could reduce the size of the vector timestamp, but is sensitive to the cluster size. In this dissertation, we present our static and dynamic clustering algorithms, and experimental results which demonstrate that our algorithms not only can significantly reduce vector timestamp size beyond Ward's algorithm, but also achieve insensitivity to cluster size.

ACKNOWLEDGEMENTS

I want to thank my advisor Dr. Paul A.S.Ward for his wisdom, enthusiasm, patience and all-around outstanding supervision during the somewhat longer than normal duration of my degree. Furthermore, without his support and guidance, from start to finish, this thesis would not have been a reality. I would like to extend my gratitude to Dr. Kostas Kontogiannis and Dr. Rudolph E. Seviora for being on my exmination committee and providing valuable comments regarding this thesis. Finally, I like to thank my family and my friend Joan Orrett for their patience, support and encouragement.

CONTENTS

FIGURES

1 INTRODUCTION

A distributed system is a collection of two or more individual computers with data communicated between those computers via some forms of communication network. The distributed system appears to the users of the system as a single computer [22].

The advantages of distributed systems are the following. Microprocessors offer a better price/performance than mainframes, which makes distributed systems more economical. A distributed system can provide much more powerful computation ability, which can speed up processing. Redundancy is available within distributed systems, which means a single machine failure will not necessarily cause the whole system to fail. The computing power of distributed systems can be increased in small increments.

With the wide availability of low-cost computers, pervasive network connectivity, and the above advantages offered by distributed systems, most large organizations around the world are utilizing distributed systems. However, distributed systems are becoming more and more complex. In order to optimally control and manage these distributed resources, improve the efficiency of analyzing and resolving partial failure, performance inefficiencies, resource arrangement and security control, distributed-systems-management tools and methodologies are needed by system administrators and other related professionals. Several distributed-systems-management tools have been developed for the above purposes, such as POET [16], Object-Level Trace [1] and the MAD environment [14, 15].

Distributed-systems-management tools are combination of the observation, interpretation and control tools for distributed systems. Such tools observe the computation of the distributed system, gather the relevant data, store them in a querable data structure, present the data to the user, and according to the user's instruction, implement manage-

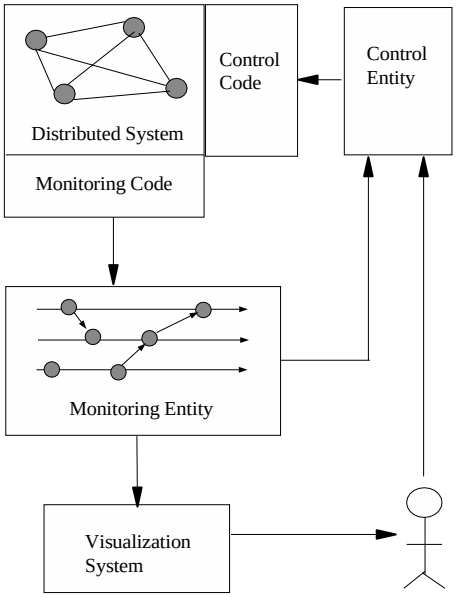


Figure 1.1: Distributed-System Observation and Control

ment actions in the distributed computation, which include debugging [3], fault analysis [10], program understanding [17], computation visualization [2], and dependable distributed computing [11].

Such distributed-systems-management tools can be broadly shown as having by an architecture as in Figure 1.1 [26]. The observation and control components constituting the architecture are defined as follows.

The monitoring code is instrumentation attached to the distributed system. It catches the relevant data from different processes in the system according to the requirements of the tools. It presents that data to the monitoring entity.

The functionality of the monitoring entity is to collect the information provided by the monitoring code which keeps capturing all data of interest within the computation of the distributed system. On the other hand, this entity organizes that data into a queryable form acting as a data structure to store the data, enabling it to be accessed and queried by

various sub systems, such as the control entity and visualization system.

The visualization system graphically presents the distributed computation to a human, most likely a system administrator. This component can visualize the specific distributed computation according to different requirements and purposes.

The control entity queries the monitoring entity, or it takes instruction from a human, or both. Then it sends instruction to the control code which has access to the system to control the computation.

In general, the monitoring code, monitoring entity, visualization system, control entity and control code are individable components which constitute a complete distributed-system management tool.

1.1 MOTIVATION

As described above, distributed systems are becoming more and more complex, which means thousands of processes might exist concurrently, and within those processes, thousands of messages could be sent and received. In other words, a large amount of communication happens in modern distributed systems. In order to maintain and manage such huge systems accurately, the distributed-system observation and control tool must be scalable. Three problems in distributed-system management tools need to be solved in order to achieve the desired scalability [26]. First of all, the data gathering ability needs to be scalable. As described, the monitoring entity acts as a single repository, and becomes a bottleneck in data gathering. It could also be a single point of failure. Second, the monitoring entity maintains a data structure which stores the gathered data. This data structure should be able to grow along with the data gathering and keep data in an efficiently querable form. Most distributed-systems-management tools apply a partial-order model to build up the data structure. However, currently none of them have suitable mechanisms

to scale the data structure. The third problem is that scalability of information-abstraction mechanisms is required by those management tools. Abstraction reduces the complexity of a system by clustering similar low-level entities together so that a large number of separate entities can be conceptually treated as a single high-level composite. It is especially unrealistic to have large-scale visualization without proper abstraction.

This thesis focuses on issues within the second problem. As described, the monitoring entity maintains a data structure to store data and organizes that data into a querable form. Such a data structure is queried by the visualization engine and control entity, and common queries are precedence queries. Such data structure can be encoded as a directed acyclic graph which maintains the transitive closure of the partial-order relation among events. In this case, each pair of ordered events within the graph is connected by an edge. Thus, precedence can be decided within constant time. However, the space consumption is unacceptably high, which makes such a method infeasible.

The Fidge/Mattern timestamp mechanism [5] was broadly applied in distributed system observation and control tools. However, this approach consumes too much real memory. The consumption of memory increases with the increment of the number of processes and the number of events within each process. For example, a thousand processes with a thousand events in each process consume about 4GB of real memory, which could easily make the system thrash.

$$1000\text{processes} \times 1000\text{events} \times 1000\text{integers} \times 4\text{bytes/per integer} \approx 4GB$$

Such an approach is limited, as it cannot scale well with the number of processes and events.

Ward's Ph.D. dissertation [26] proposed a clustering-timestamp mechanism in order to reduce the total timestamp size over the Fidge/Mattern approach. The basic idea of

his theory is to group processes into clusters according to the communication pattern within the distributed computation. Events in each process that receives communication from outside the cluster are required to use a Fidge/Mattern timestamp, while events that only communicate within the cluster are given a cluster-timestamp whose size is equal to the size of cluster. Such a mechanism has the potential to greatly reduce the memory consumption. Ward applied one method in implementing self-organizing hierarchical cluster strategy: merge-on-1^{st}-communication. However there is no single cluster size or range of cluster size that is appropriate for all computations. Ward provided no proof that such a single cluster size exists.

1.2 CONTRIBUTION

This thesis addresses the need for a scalable data structure. Building on Ward's cluster timetamp approach, it provides the following significant contributions to scientific and engineering knowledge.

1. The creation of a static clustering algorithm that is suitable for all computations and greatly reduces the timestamps size. This static algorithm allowed us to demonstrate that there do exist cluster-size ranges over which timestamp size reduction is stable;

2. The creation of a dynamic cluster-timestamp creation algorithm and an optimized dynamic cluster-timestamp creation algorithm;

3. Theoretical cost and experimental analysis of the dynamic cluster-timestamp creation algorithm.

The static clustering algorithm applies a hierarchical clustering method. By doing so, small groups of processes are clustered together based on their similarity. After

progressively merging, the size of the clusters grows until it reaches a maximum cluster size limit. Through this algorithm, the cluster-timestamp creation algorithm could avoid sensitivity to the maximum cluster size enabling it to work well with a large range of computations.

Our dynamic cluster-timestamp creation algorithm is based on the static clustering algorithm. With this algorithm, when the distributed system is executing, after each collection of certain number of events, we perform the static clustering algorithm, creating new clusters for those events. We then calculate the total timestamp size. We compare this result with the result of leaving those events in the previous cluster. After comparison, the cluster choice which generates the least timestamp size will be kept for further clustering computation. Our optimized dynamic cluster-timestamp creation algorithm is based on the dynamic clustering algorithm, it avoids the unnecessary reclustering procedure, thus speeding up the whole processing. Such mechanisms dynamicly generate the cluster-timestamp within the distributed computation. It not only inherently achieves the advantages of the static clustering algorithm but also obtains better results in reducing the timestamp size.

Our cost analysis shows the time complexity for our dynamic cluster-timestamp creation algorithm is O(N^2) , where N is equal to the total number of processes in the distributed system.

1.3 ORGANIZATION

The remainder of this thesis is organized as the follows. In the first part is backgroud and related work. In this chapter, Fidge/Mattern timestamps will be introduced, together with a description of the problem of space consumption of Fidge/Mattern timestamps. The clustering timestamp algorithm created by Ward will be reviewed here, as well as other

related work which has been developed to address this problem of efficient reduction of space consumption.

In the second part, we describe our algorithms. The dynamic-cluster-timestamp-creation algorithm and the corresponding precedence test algorithm will be illustrated and justified. Also the cost analysis of the algorithm will show their efficiency in applying this approach to create cluster timestamps.

In the third part we provide simulation results and relevant analysis. These results provide evidence in support of our algorithms.

Finally, we conclude and discuss future work.

2 BACKGROUND AND RELATED WORK

In order to precisely understand the computation activities in a distributed system it is very important to determine the causal relationship between events happening in that system. Events occurring in a different order can produce fundamentally different results. Therefore, the methodology and practice which can effectively and accurately determine the precedence between events plays a crucial role in distributed-system observation, such as analyzing, debugging, visualizing and monitoring the operation of a distributed system.

In a distributed system there are four types of events:

- Unary events, which do not involve any communication. Examples are events C and D in Figure 2.1;
- Synchronous events, which interact with one another by synchronous communication. We model synchronous events as a single logical event that occurs simultaneously in different processes. An example of a synchronous event is event E in Figure 2.1;
- Send events, that correspond to the asynchronous send operation in distributed systems. An example is event A in Figure 2.1 ;
- Receive events, that correspond to the asynchronous receive operation in distributed systems. An example is event B in Figure 2.1.

Events within a distributed-system computation are not, in general, totally ordered. They are, rather, partially ordered. The causality relationship between events is one of

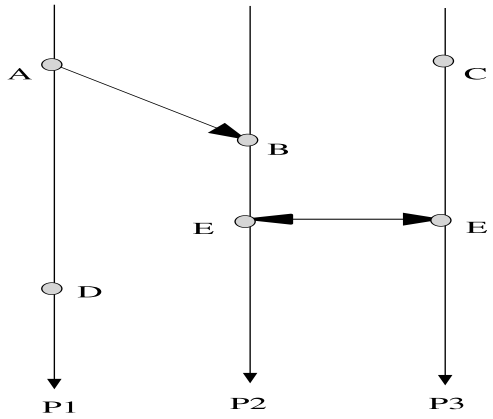


Figure 2.1: Events Type

two types: **happened-before** and **concurrent**. The **happened-before** relationship, first defined by Lamport [18], is denoted as:

$$A \rightarrow B$$

where A and B are events.

A pair of events, A and B, is in the **happened-before** relation if:

- A and B occur in same process, and A is observed to happen before B happen; Or
- if an asynchronous communication occurred, A is the send event and B is the corresponding receive event ; Or
- there exists an event C, and $A \rightarrow C$ and $C \rightarrow B$.

The **concurrent** relationship is denoted as:

$$A \| B$$

where A and B are events. A pair of events, A and B, are in this relationship if neither $A \rightarrow B$ nor $B \rightarrow A$. This is also denoted as $A \nrightarrow B$.

A simple and straightforward method to encode these relationships is a directed acyclic graph. In such a graph, any two ordered events are connected via an edge. Precedence can thus be decided within constant time. However, the space-consumption overhead is unacceptably high, which makes this approach unrealistic.

Systems we are aware of store the transitive reduction of the partial order. This is space-efficient but requires a search operation on the graph to determine precedence. A common way to compensate for this deficiency is to add additional information to each encoded event. The most general form this information takes is logical timestamps. Logical timestamps are integers or vector of integers that are associated with events to enable the efficient determination of precedence between events.

From Summers' point of view [23], the optimal timestamp algorithm has the following attributes:

1. Whenever an event happens in a distributed computation, the algorithm is able to calculate its corresponding timestamp and assign it to the event, and these actions do not require waiting until all events have occurred.

2. Since timestamps are calculated, manipulated, passed and stored, the size of timestamps should be as small as possible.

3. Timestamps should enable the precedence test be correct and efficient.

4. The information passed between traces in a distributed system required to compute timestamps should be appended to messages within that system. Otherwise, extra synchronization is caused by the passing of additional messages, which might disorder the actual synchronization.

In practical distributed-system observation, the records of the computation are collected into the observation tool. Only in the tool are the timestamps calculated, by simulating the original pattern of the distributed computation. This avoids most of the overhead which would be added to the messages that are passed between processes.

Thus far, many logical timestamp algorithms have been developed. In the following subsections, we will describe some of these algorithms. However, in our description of these algorithms (specifically, the Fidge/Mattern algorithm, Lamport timestamp, Fowler/Zwaenepoel algorithm, and Jard/Jourdan algorithm), they will be introduced in an online format since it is simpler. This means events are described as being timestamped when they happen and timestamp data is appended to each message passed among traces in the distributed system. The Fidge/Mattern timestamp algorithm and Ward's cluster timestamp algorithm will be introduced in detail, since this thesis builds on them. Other approaches to the problem of efficient timestamping will be described briefly.

2.1 FIDGE/MATTERN TIMESTAMP ALGORITHM

Fidge and Mattern independently developed the logical vector clock [4, 5, 19]. The vector clock is an array of integers with size N, which is the number of processes in the distributed system, with this array indexed by process ID. Here we define T_i to be a current vector clock of process i and FT_e to be the Fidge/Mattern vector timestamp of an event e, with the k^{th} element in such a vector clock represented as $T[k]$. Likewise, an event j occuring in process i is represented as e_i^j. The vector timestamp appended to a message that an event sends out during communication is represented as T_{send}. Likewise, the vector timestamp appended to a message that an event receives is denoted as $T_{receive}$.

The Fidge/Mattern algorithm is then designed as follows:

1. Initialize each vector clock with all elements equal to zero $T = (0, 0, 0, ..., 0)$;

2. Before each event, the element $T[k]$ of its vector clock is increased by 1, where k is the process or processes (if the event is a synchronous event) in which the event occurred;

3. If an event e is a send event with current vector clock T. The Fidge/Mattern vector timestamp FT_e for event e is $FT_e = T$. The vector timestamp appended to the message it sends out is T_{send} which is identical to T;

4. If an event e is a receive event, it receives a vector timestamp T_{send} from the message sent out by the corresponding send event which occurs in process k. Also, this receive event occurs when its current vector clock is T. The algorithm calculates this event's Fidge/Mattern timestamp as follows:

$$T_{receive} = T_{send}$$

$$T_{receive}[k] = T_{send}[k] + 1$$

$$FT_e = max(T_{receive}, T)$$

   where $max()$ is the element-wise maximum of the two vectors. Furthermore, the current local clock for this process which event e occurred on is updated by $T = FT_e$

5. If an event e is a unary event, then $FT_e = T$;

6. If an event e is a synchronous event, it receives multiple vector clocks, T^i, T^j, T^k, etc., from its home processes. Then FT_e = max(T^i, T^j, T^k, ...). Furthermore, the current local clock for e's home processes in which event e occurred is updated to

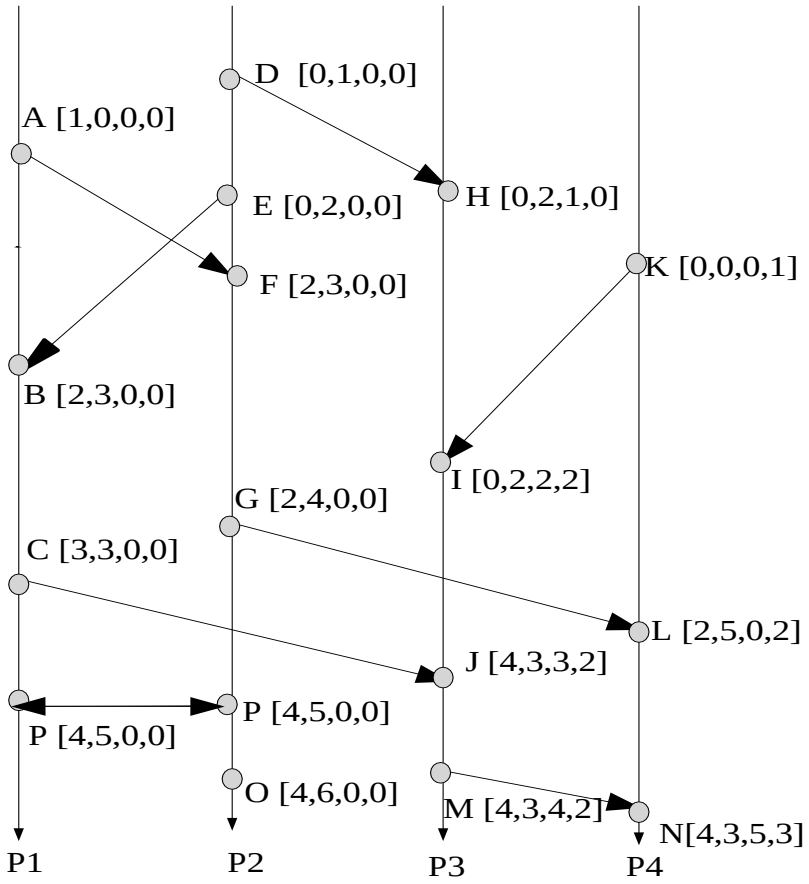


Figure 2.2: Events labeled with Fidge/Mattern timestamps

$T = FT_e$.

Figure 2.2 is an example showing how Fidge/Mattern vector timestamps are created and assigned to each event. To further explain the Fidge/Mattern vector timestamp algorithm, the processing for events A, J and O will be described in detail.

As we can see, event A is the first event in process 1. According to Rules 1 and 2 in the algorithm, the vector clock for process 1 at the time event A occurs is $T_1 = [1, 0, 0, 0]$. Since event A is a send event, then according to Rule 3, the Fidge/Mattern timestamp for event A is: $FT_A = T_1 = [1, 0, 0, 0]$.

Another example is how to process the vector timestamp for event J. According to Rule 2, the current vector clock for process 3 at the time event J occurs is $T_3 = [0, 2, 3, 2]$. It receives a message from event C, with $T_{send} = T_1 = [3, 3, 0, 0]$ (Rule 3). Therefore it receives a vector clock $T_{receive} = [4, 3, 0, 0]$ according to Rule 4 in the algorithm. Thus, the Fidge/Mattern timestamp for this event J is $FT_J = max(T_3, T_{receive}) = max([0, 2, 3, 2], [4, 3, 0, 0]) = [4, 3, 3, 2]$.

The vector clock for process 2 at the time event O occurs is $T_2 = [4, 6, 0, 0]$ according to Rule 2. Since event O does not involove any communication, its Fidge/Mattern timestamp is $FT_O = T_O = [4, 6, 0, 0]$ (Rule 5).

2.1.1 PRECEDENCE TEST IN FIDGE/MATTERN ALGORITHM

The precedence test between any two events with Fidge/Mattern timestamps is:

$$e^j \rightarrow e^n \Longleftrightarrow \exists p \in \emptyset(e^j) \;\; FT_j[p] < FT_n[p]$$

where e^j and e^n are events, $\emptyset(e^j)$ denotes the set of home processes for event e^j (since synchronous events have more than one home process).

As the above equivalence shows, in order to determine the relationship between two events e^j and e^n, we can apply the following tests to them: if $\exists p \in \emptyset(e^j) FT_j[p] < FT_n[p]$, then $e^j \rightarrow e^n$. If $\exists p \in \emptyset(e^n) FT_n[p] < FT_j[p]$, then $e^n \rightarrow e^j$. Otherwise $e^j \| e^n$. As such, usually contant-time tests are needed to determine the relationship[1]. Thus the precedence test between a pair of events can be done in constant time.

For example, in the Figure 2.2 the relationship between events F and L is $F \rightarrow L$, since $FT_F[2] < FT_L[2]$. The relationship between events B and J is $B \rightarrow J$, since $FT_B[1] < FT_J[1]$. However the relationship between events G and C is $G \| C$, since

[1] in most cases, the number of home processes for a synchronous event is limited to 2

neither $FT_G[2] < FT_C[2]$ nor $FT_C[1] < FT_G[1]$.

The above description of the Fidge/Mattern timestamp algorithm shows that these timestamps can be calculated and assigned to events by applying the algorithm in a distributed system. The precedence test can be done in a very efficient way as well. However, this approach has a scalability problem. The size of the vector timestamps is a bottleneck for scalability of the data structure in a distributed-system-observation tool.

To illustrate that this issue is caused by the Fidge/Mattern timestamp approach, imagine a thousand processes in a distributed system instead of the 4 processes of Figure 2.2. Similarily assume there are one thousand events occurring in each process instead of only 3 or 4 events, as occur in each process in Figure 2.2. Thus, there are one-million events in such a distributed system. If this system applies the Fidge/Mattern timestamp algorithm to calculate and assign vector timestamps to each event, then each event in this system has a vector timestamp with 1000 integers. Since that timestamp information has to be stored within the data structure of the observation tool for further precedence testing or other purposes, in total we need about $4GB$ of memory to store that data.

$$(1000 \text{ processes} \times 1000 \text{ events} \times 1000 \text{ integers} \times 4 \text{ bytes/integer}) \approx 4GB$$

Thus, this system suffers an unacceptably high memory consumption. This implies that the data structure will spill into virtual memory and the virtual memory system will thrash.

Potentially an improved Fidge/Mattern vector timestamp can be implemented as an associative array with only non-zero integers. In practice this approach does not save space, since the Fidge/Mattern timestamp encodes transitive causality. Thus, most elements in the vector timestamps of most events occuring in the distributed computation are not zero [26].

Ward created a cluster-timestamp algorithm based on the Fidge/Mattern approach to reduce the size of the vector timestamp. This algorithm will be illustrated in the following section.

2.2 WARD'S CLUSTER TIMESTAMP ALGORITHM

Ward proposed a cluster timestamp algorithm in order to reduce the memory consumption of vector timestamps based on the observation that most processes do not communicate with many other processes [24, 25, 26]. In another words, there are no communication events directly connecting most traces with most other traces.

Before we describe the cluster timestamp algorithm further, we need to introduce a crucial notion in this algorithm, the **Cluster-Receive** event. Ward defined it as follows: an event is a cluster-receive if and only if it is a receive event with a partner send event on a process in a different cluster or a synchronous event whose home processes occur in different clusters.

The basic idea for this algorithm is to group processes into different clusters, where only cluster-receive events maintain a Fidge/Mattern timestamp. The size of vector timestamp of all other events within the cluster will be shortened to the number of processes in the cluster. By reducing the number of cluster-receive events, the average size of vector timestamp for each event in the distributed computation will be less than the size of the Fidge/Mattern vector timestamp. The cluster-timestamp creation algorithm is therefore a combination of the timestamp algorithm and the clustering strategy.

In Ward's thesis, he devised two timestamp algorithms, the two-level algorithm and the hierarchical algorithm. In the two-level timestamp algorithm, the Fidge/Mattern timestamps are calculated for each event. If the event is a cluster-receive event, the Fidge/Mattern vector timestamp will be kept. Otherwise, the event will be assigned

a cluster timestamp which is the projection of Fidge/Mattern timestamp over the processes in the cluster. Then the Fidge/Mattern vector timestamp will be erased and the corresponding occupied memory will be freed at such time as it is no longer needed to compute further Fidge/Mattern timestamps.

The hierarchical timestamp algorithm extends the two-level algorithm to an arbitrary hierarchy of levels. The core idea for this approach is for each level of the hierarchy to maintain a set of cluster-receive events that have timestamps only to the next level in the hierarchy. Therefore, a tradeoff is allowed between the size of the cluster, the number of cluster-receive events and the size of the cluster-receive timestamp. In the two-level algorithm, there is only a tradeoff between the first two of these three. Unlike the two-level algorithm, where an event only belongs to one cluster, in the hierarchical algorithm an event belongs to a sequence of clusters, extending out to a universal cluster that contains all processes in the distributed computation. In the hierarchical approach every event is assigned a cluster timestamp. The cluster timestamp is the projection of the Fidge/Mattern timestamp over the cluster processes of the cluster at the level in which the event no longer forms a cluster-receive event. For unary and send events, this is the first level cluster. For a receive event, the cluster level is that level at which such a receive event and its parter belong to same cluster. In other words, at this level these two clusters which this receive event and its parter previously belong to, respectively, are merged. This decision of mergeability is subject to the clustering strategy.

Ward also developed a complete precedence-test algorithm for the above two timestamp algorithms. He proved that the precedence test can be done within $O(C)$ time complexity, where C is the cluster size (that is, the number of processes in the cluster).

Since the cluster-timestamp creation algorithm is a combination of the timestamp algorithm and the clustering strategy, the clustering strategy plays a critical role in the cluster-timestamp-creation algorithm. A good clustering strategy is crucial to the quality

of the timestamp algorithm, since it could significantly reduce the number of cluster-receive events, and therefore reduce the average size of timestamps for events occurring in the distributed-system computation.

Ward developed fixed-size and semi-dynamic clustering strategies. The fixed-size clustering strategy has clusters that are predetermined. The clusters cannot be changed during the execution of the distributed computation. Ward determined that the fixed-size technique was problematic. The problem is that the clustering strategy is unrelated to the computation pattern in the distributed system. If the clustering is bad there is no way to change it. Therefore he focused on the alternative possibility, the semi-dynamic clustering strategy.

The semi-dynamic cluster strategy does not determine clusters before the execution of the distributed-system computation. Rather, the clusters are determined by the run-time behavior of the distributed computation.

The semi-dynamic cluster strategy Ward implemented is called the self-organizing approach. In this approach, each trace is put into a single cluster with size one initially. Clusters are merged based on the number of communications occurring between the clusters. This merging action is under constraint of a maximum cluster size.

The action of merging clusters should happen as soon as possible, since after an event is timestamped, either the timestamp will never change or a computation cost needs to be paid to change it. Therefore, if the vector timestamp for some events never change, some of them might have a bigger timestamp than they need under a better clustering. Under this consideration, Ward implemented the merge-on-first-communication strategy in the self-organizing algorithm. In this strategy, each individual process will be put into a cluster with size one initially. Then, as long as first communication happens between clusters, they will be merged into a bigger cluster, subject to a maximum cluster size.

According to the simulation results for the fixed-cluster and self-organizing cluster-

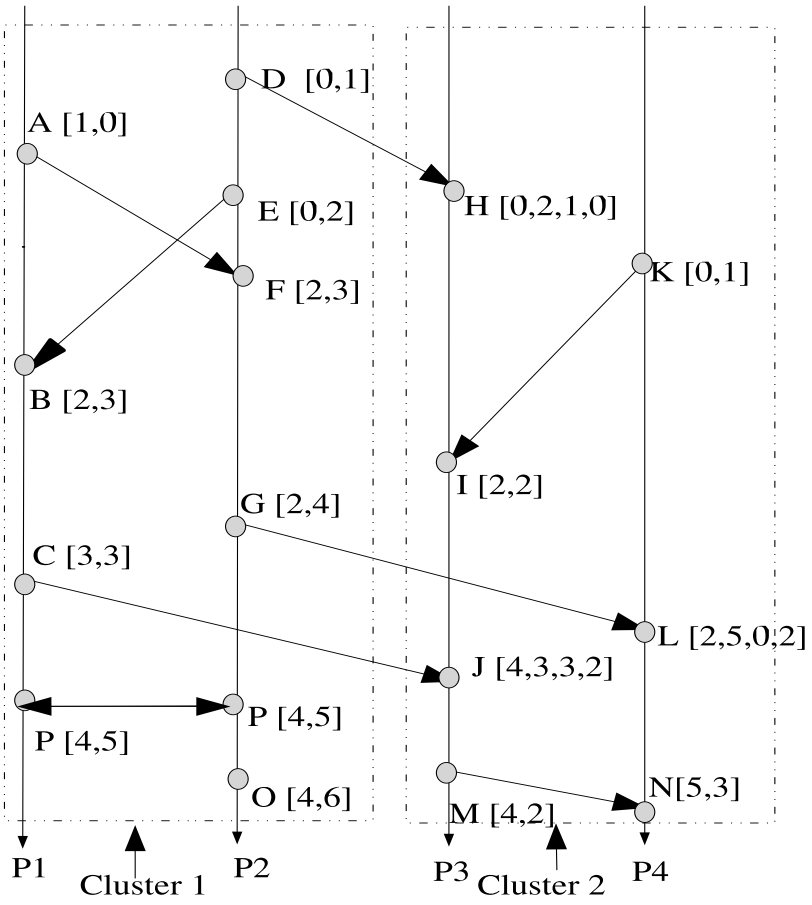


Figure 2.3: a fully timestamped system with two clusters

ing approaches, both of them achieve a significant space-consumption reduction. The results also show that the self-organizing cluster algorithm, with the merge-on-first-communication strategy is more stable with cluster size for all simulation environments than the static one. Also, the average size of vector timestamp generated by the merge-on-first-communication strategy is consistently superior to the static-clustering strategy.

However, the simulation results also show that both the fixed-size clustering and the self-organizing approaches are very sensitive to the cluster size. There is no single cluster size or range of cluster size appropriate for all distributed computations.

Figure 2.3 shows a system with 2 clusters. All events in this system are timestamped with the 2-level cluster timestamp algorithm. There are 17 events occurring in 4 processes. Three of them are cluster-receive events. Therefore, the average timestamp size for each event in this system is 2.35. By contrast, the timestamp size for each event is 4 if events are timestamped with the Fidge/Mattern algorithm.

Note that in Figure 2.3, if processes 1 and 4 were put into Cluster 1 and processes 2 and 3 were put into Cluster 2. There will be 8 cluster-receive events. In this case the average timestamp size for each event is 2.91. It is obvious that the former clustering strategy is better since it can save more space. Therefore, a good clustering strategy plays a crucial role in order to reduce the timestamp size for each event in the distributed-systems-observation tool.

2.3 OTHER APPROACHES FOR SPACE REDUCTION

In this section, some related alternative approaches to timestamp events in distributed systems and efficiently determine the precedence relationship among them will be briefly summarized. In order to make these descriptions simpler, timestamping synchronous events will be omitted.

2.3.1 LAMPORT TIMESTAMP

Lamport presented his idea to timestamp events occurring in a distributed system with a single integer [18]. The Lamport timestamp algorithm is described as follows:

1. Initialize each local clock to be $T = 1$;

2. Increment the local clock by 1 between events;

3. If an event e is a send event or an event which does not involove any communication, the Lamport timestamp for this event is $LT_e = T$, where T is the current local clock for the process on which event e occurred. The clock appended to the message it sends out is $T_{send} = T$;

4. If an event e is a receive event, it receives the clock T_{send} from the received message. $T_{receive}$ is calculated by $T_{receive} = T_{send} + 1$. It also has its own current local clock T. Overall, the Lamport timestamp for event e is generated as follows:

$$LT_e = max(T_{receive}, T)$$

Further, the local clock for the process on which event e occurred is updated by $T = LT_e$.

Figure 2.4 shows an example of Lamport timestamp.

As we can see, the creation of a Lamport timestamp is constant time and the space consumption is O(1). Therefore this timestamp algorithm is very efficient. However, the biggest limitation of this algorithm is that it can not completely determine precedence since it imposes a total order on the partial order. For example, in Figure 2.4, while the timestamp of event G is smaller than that of event J, $G||J$.

2.3.2 FOWLER/ZWAENEPOEL TIMESTAMPS

The Fowler/Zwaenepoel timestamp [6] augments up a transitive-reduction graph with a set of edges. The graph connects event e with its greatest events in each trace which have direct communication with its trace. However, those greatest events might not be the greatest predecessors for event e, since this timestamp does not capture the transitive dependency.

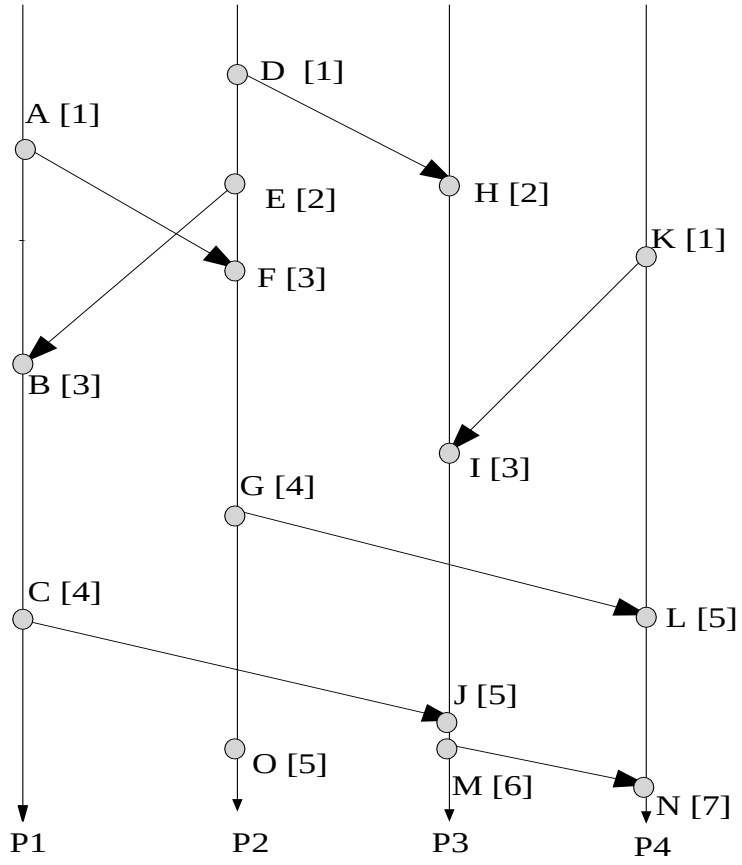


Figure 2.4: Events labeled with Lamport timestamps

The Fowler/Zwaenepoel timestamp algorithm can be described as the follows:

1. Initialize each vector clock with all elements equal to zero $T = (0, 0, 0, ..., 0)$;

2. Before each event, the element $T[k]$ of its vector clock is increased by 1, where k is the process in which this event occurred;

3. If event e is a send event, the current vector clock of the process on which the event e occurred is T. $T_{send} = T[k]$ where k is the process in which this event occurred. The Fowler/Zwaenepoel vector timestamp FZ_e for event e is $FZ_e = T$;

4. If event e is a receive event, it receives T_{send} from the send event. Then $T_{receive} = T_{send}$. The Fowler/Zwaenepoel timestamp for the receive event e is:

$$\forall k \neq PID(sendevent)\ \ FZ_e[k] = T[k]$$

$$\forall k = PID(sendevent)\ \ FZ_e[k] = max(T_{receive}, T[K])$$

   Further, the local clock for the process on which event e occurred is updated by $T = LT_e$.

5. If event e does not involve any communication, $FZ_e = T$.

Figure 2.5 shows an example of how the Fowler/Zwaenepoel timestamp algorithm creates vector timestamps and assigns them to each event.

The Fowler/Zwaenepoel algorithm can be implemented by using an associative array since the size of the array is proportional to the number of traces that directly communicate with the trace in which the event being timestamped happens. Therefore, the size of vector timestamps of Fowler/Zwaenepoel can be greatly reduced compared to the Fidge/Mattern timestamp. However, since this algorithm does not capture transitivity, the precedence test might require a search through the vector space, which can be linear in the number of communications within the distributed system in the worst case. For example, in Figure 2.5 $FZ_A[1] > FZ_L[1]$, however $A \rightarrow L$. In order to determine $A \rightarrow L$, communications between A and$F$, F and G, G and L are required.

2.3.3 JARD/JOURDAN TIMESTAMPS

Jard and Jourdan presented the idea of observability and pseudo-direct dependence in a distributed system [12]. Observability means that some events in the distributed system are worth observing but others are not. However, even those unobservable events can

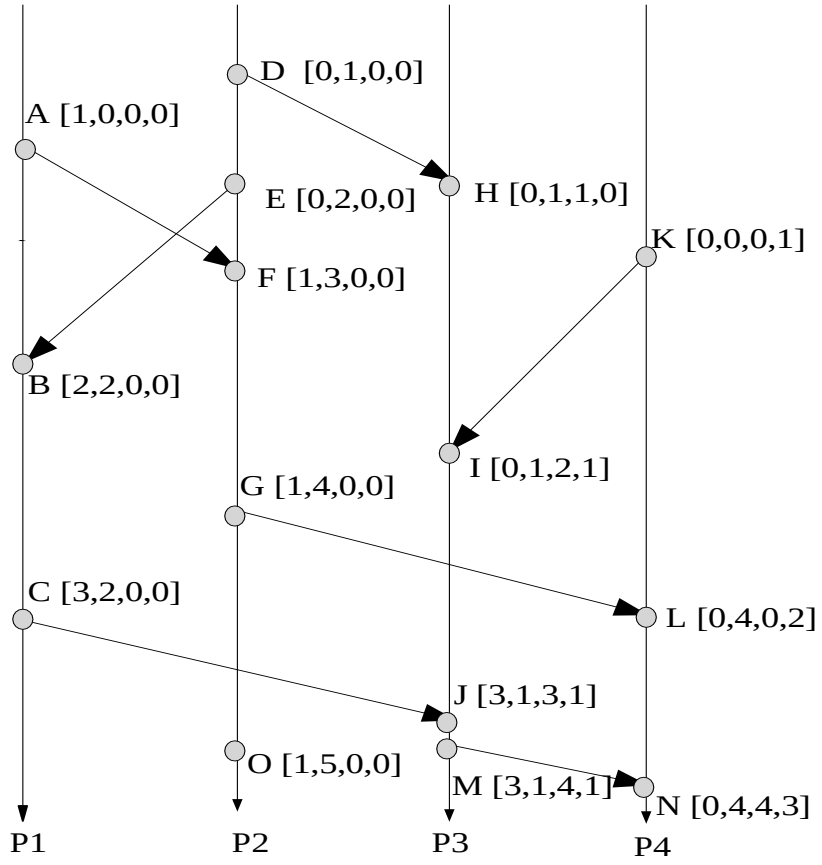


Figure 2.5: Events labeled with Fowler/Zswenepoel timestamps

cause a transitive dependency that needs to be captured. Therefore, pseudo-direct dependency is introduced in the Jard/Jourdan timestamp algorithm. The notion of pseudo-direct dependency is as follows. Event A is pseudo-dependent on event B (which can be denoted as $B \ll A$) if there are no observable events on the path connecting event A and event B.

The Jard/Jourdan timestamp algorithm is thus an extension of the Fowler/Zwaenepoel timestamp algorithm. It can be described as follows:

1. Initialize each vector clock with all elements equal to zero $T = (0, 0, 0...0)$;

2. Before each event (which is not the first event in its process), its current vector clock inherits from the vector timestamp of its greatest predecessor which occurs in its process. If the greatest predecessor of this event is an observable event, then $T[k]$ is incremented by 1, where k is its process ID, and $\forall p \neq k\ T[p] = 0$. Otherwise, if the greatest predecessor of this event is an unobservable event, then the current vector clock stays unchanged.

3. If an event e is a send event or an event without involving any communication, then $JT_e = T$. The vector clock appended to the message it sends out is $T_{send} = T$.

4. If an event e is a receive event, it receives vector clock T_{send} from its corresponding send event. If its send event is an observable event, then $T_{receive}[k] = T_{send}[k] + 1$, where k is the ID of the process in which its corresponding send event occurred, $\forall p \neq k\ T_{receive}[p] = 0$. Otherwise, if its send event is an unobservable event, then $T_{receive} = T_{send}$. According to the Rules 1 and 2, event e has its own local vector clock T. The Jard/Jourdan vector timestamps of this receive event e is $JT_e = max(T, T_{receive})$.

Figure 2.6 shows an example of how the Jard/Jourdan timestamp algorithm creates vector timestamps and assign them to each event. In the figure, those events displayed with a rectangle are unobservable events. Others are observable events.

As with the Fowler/Zwaenepoel timestamp, the Jard/Jourdan timestamp is better represented by an associative array for the same reason, so the size of vector timestamp can be smaller than that of the Fidge/Mattern timestamp. Since the Jard/Jourdan timestamp algorithm does not capture the transitive relationship between events, therefore the greatest observable events might not be the greatest predecessors of the event. Thus the precedence test in general needs to search through all events in the worst case, as is the case with the Fowler/Zwaenepoel timestamp algorithm.

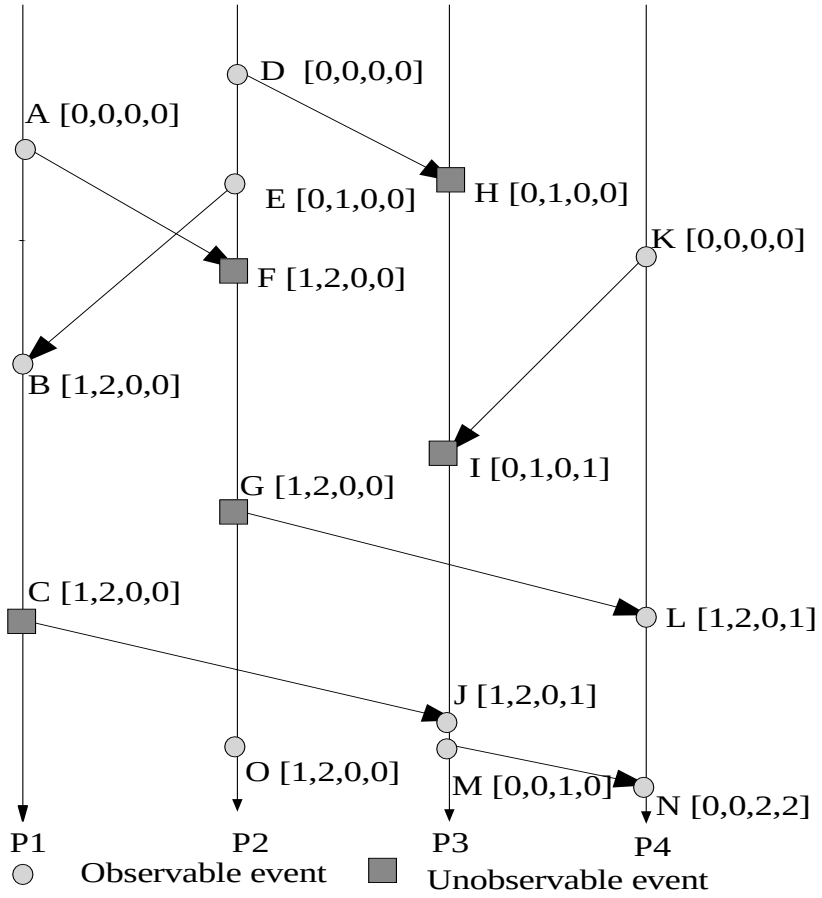


Figure 2.6: Events labeled with Jard/Jourdan timestamps

2.3.4 THE SUMMERS CLUSTER TIMESTAMP ALGORITHM

Summer proposed a two-phase cluster timestamp algorithm in his thesis [23]. This algorithm was designed based on the idea that events within a cluster can be causally dependent on outside-cluster events through receive events from transmissions occurring outside the cluster. Those receive events are called cluster-receive events, whose corresponding send event occurs outside the cluster. After identifying cluster-receive events, the vector timestamp of all events within the cluster, which is a Fidge/Mattern vector timestamp, can be reduced to a vector timestamp with a size of double the number of

processes in the cluster. Thus, Summers cluster timestamp algorithm can reasonably reduce the storage space.

The Summers algorithm requires two phases. In the first phase, all cluster-receive events are identified and timestamped with Fidge/Mattern timestamps, as well as being recorded for further use. In the second phase, the algorithm produces a two-part timestamp with size of double the number of processes in the cluster for events within the cluster. The first half of this vector timestamp is the projection of the Fidge/Mattern timestamp over the processes in the cluster for cluster-receive events. The second part of the timestamp specifies the preceding cluster receives.

If both events are in same cluster, their precedence can be determined within constant time using the Fidge/Mattern algorithm. Otherwise, Summers provides a mechanism to convert the cluster timestamp to a Fidge/Mattern timestamp. The time complexity of this conversion is linear in the product of the cluster size and the number of processes. After conversion, the precedence test can be done within constant time.

The precedence test between two events, either of which are not in the cluster, is thus time consuming since it requires Fidge/Mattern timestamp regeneration. Also, the Summers algorithm is a static timestamp algorithm, which limits its applicability.

2.3.5 SYNCHRONOUS COMPUATTION TIMESTAMP ALGORITHM

Garg and Skawratananond proposed a fundamentally different timestamp algorithm for synchronous system from the Fidge/Mattern method [7]. They consider that a computation based on synchronous messages is logically equivalent to a computation in which all messages are instantaneous. Therefore, the timestamp diagrams for synchronous computations occurring in distributed system can be denoted with vertical message arrows which can construct a communication topology.

Based on the above consideration, they developed an algorithm to capture the order

relationship with vectors of size less than or equal to the size of the vertex cover of the communication topology of the distributed system. Thus, the vector timestamp in this algorithm does not utilize one component per trace. As a result, it efficiently reduces the size of vector timestamp. Since the size of such vector timestamps is fixed and the content of this vector never changes during the execution of the algorithm, the precedence determination is straightforward and efficient by applying this timestamp algorithm.

However, Garg and Skawratananond's algorithm only applies to the synchronous computation. Furthermore, according to this algorithm, the communication topology must be known a priori. Thus, this methodology can only be applied in a static way practically. Moreover, in this algorithm, the unary events occurring in the distributed system have double the size of vector timestamp over those synchronous events. Worse, the timestamp for such unary events cannot be fixed until its following synchronous event occurs in the same trace.

2.3.6 THE SINGHAL/KSHEMKALYANI'S ALGORITHM

Singhal and Kshemkalyani proposed a different method to reduce the space consumption of timestamping events in distributed system [20, 21]. Their technique is based on the observation that only a few elements of the vector timestamp are likely to change between successive events at a trace. This is even more obvious for a distributed system with a large number of processes.

In this algorithm, each process i needs to keep two additional vectors, $LU_i[1, 2, ...N]$ and $LS_i[1, 2, ...N]$. $LU_i[1, 2, ...N]$ denotes "Last updated". Element $LU_i[j]$ indicates the value of $T_i[i]$ when process i last updated element $T_i[j]$. $LU_i[j]$ only needs to be updated when an incoming message causes process i to update entry $T_i[j]$. $LS_i[1, 2, ...N]$ denotes "Last sent". Element $LS_i[j]$ indicates the value of $T_i[i]$ when process i last sent a message to process j.

Since the last communication from trace i to trace j, only those entries $T_i[m]$ changed for which $LS_i[m] < LU_i[m]$. Thus, when a process i sends a message to a process j, i piggybacks only those elements $T_i[m]$ of its vector timestamp for which $LS_i[m] < LU_i[m]$. In another words, a process i only needs to send process j a set of tuples $\{(m, T_i[m]) | LS_i[m] < LU_i[m]\}$ instead of a message with a vector of N elements, where N is the number of processes in system. Therefore, this algorithm substantially lessens the communication bandwidth and memory consumption of passing messages.

However, Singhal and Kshemkalyani's technique still keeps a Fidge/Mattern vector timestamp for each event in the distributed system. As we discussed before, this is very space-consuming and might cause the whole system thrash. Therefore, this technique does not solve the scalability problem of the data structure in distributed-system-observation tools.

3 DYNAMIC CLUSTER-TIMESTAMP CREATION ALGORITHM

In this chapter we introduce a dynamic cluster-timestamp creation algorithm, which is a combination of clustering strategy and timestamp algorithm. It is an extension of Ward's cluster-timestamp algorithm. Ward never proved that there exists a cluster size or a range of cluster size which is suitable for all ditributed computations. We start by demonstrating by means of static clustering that cluster timestamps are feasible. That is, we show that such a range of cluster size exists. We then extend Ward's algoirthm with the following significant changes :

1. We make it a fully-dynamic clustering algorithm, which we then optimize. Our algorithm dynamicly creates clusters according to the ongoing distributed computation. They operate clustering based on a subset of events. Processes are permitted to move between clusters as needed by applying these algorithms. These dynamic cluster-creation algorithms are appropriate for all distributed computations;

2. We create the corresponding cluster timestamp creation algorithm which manipulates cluster timestamps according to the cluster pattern generated by the clustering strategy.

3. We create the corresponding precedence test algorithm which determines the precedence between two events.

We divide this chapter into the following 5 parts. First, we introduce the static cluster algorithm. In this algorithm, we develop a mechanism to calculate the similarity between

clusters, which is key for clustering processes. The cost analysis of the algorithm will also be described here, as it becomes a component of our dynamic algorithm. Second we introduce a dynamic clustering strategy which applies the static cluster algorithm to each "n-events", followed by the corresponding time-complexity analysis. In Section 3.3, we describe an optimized dynamic-clustering strategy: after collection of each "n-events", we perform the static clustering algorithm, creating new clusters (that is we apply the dynamic strategy we introduced in Section 3.2). We then calculate the total timestamp size. We compare this result with that which is achieved by leaving those events into previous clusters. After the comparison, the better clustering choice will be kept for further clustering computation. Again, the computation cost for this optimized dynamic clustering strategy is analyzed. In Section 3.4, the corresponding timestamp creation algorithm will be introduced and its time-complexity analysis is given. In Section 3.5, the corresponding precedence-test algorithm is described.

3.1 STATIC CLUSTERING ALGORITHM

In this section, we introduce a static clustering strategy. First of all, we explain our choice of clustering algorithm. Then, we describe how we measure the similarity between different clusters. Further more, we describe how to apply the hierarchical cluster approach to grouping processes in the distributed computation. Finally, we analyze the computation cost for this static clustering algorithm.

The K-medoid and hierarchical-cluster methods are described in "Finding Groups in Data: An Introduction to Cluster Analysis" [13]. The idea of the K-medoids algorithm is to randomly choose some objects as medoids. Then, according to the distance between other objects and those medoids, other objects are grouped to the closest medoid to construct different clusters. This is subject to a maximum cluster size limitation. The idea of

the hierarchical clustering method is to assign each object into a single cluster initially. The two clusters with the most similarity are merged. This is repeated recursively until clusters reache the maximum cluster size limit.

The reason we considered the k-medoid is because this algorithm is more efficient than the hierarchical method. However, in experimenting with the k-medoid approach we found that there are no obvious centroids of processes in distributed systems. Since most distributed systems do not apply collections of the master-slave model, it is problematic that any process in distributed system can represent the centre of a cluster of processes. Further, the results from our experiments showed that the k-medoid approach was poor since it selects the number of clusters to be generated, rather than grouping the size of the desired clusters. As a result many processes are grouped into a single cluster with the remaining clusters left sparse. The cluster timestamp based on such clustering will not achieve significant space reduction against the Fidge/Mattern vector timestamp, as most events in distributed computation would have a timestamp that was just a little smaller than the Fidge/Mattern one.

We therefore evaluated the hierarchical clustering approach. Since there is no special process representing the centre of the processes within a distributed system, the hierarchical method, which progressively merges clusters with highest similarity into bigger clusters, seems a much more appropriate solution.

We now explain the strategy we employed to measure the similarity between clusters for the hierarchical method. As described, the biggest concern of our clustering strategy is to minimize the number of cluster-receive events, subject to a maximum cluster size. Therefore, the key element to decide is which pair of clusters need to be merged. Let us recall the definition of cluster-receive event here: a cluster-receive event is an event whose corresponding send event is outside the cluster, or an event whose home processes are in different clusters (if this event is a synchronous event). Thus, a cluster-

receive event means a communication occurrence between two clusters. It seems like the more communication between two clusters, the higher the similarity between those two clusters. However, such a conclusion has a significant flaw. The problem with this idea is that as clusters increase in size, they are inherently likely to have more communication with other clusters. Therefore, we applied a modified approach to measure the similarity between two clusters, which is to normalize the number of communications based on the combined size of the pair of clusters. The pair of clusters with the highest normalized communication count will be merged subject to the fixed cluster size limit.

The case of synchronous events is somewhat special since the synchronous communication involves both send and receive. Therefore if there is a synchronous communication occurrence between two clusters, it will be counted as two communication occurrences rather than a single communication occurrence. This is because the benefit of merging two clusters with a synchronous communication between them is to remove two cluster-receive events.

The input of this static-clustering algorithm is all events occurred in a distributed computation, the output is a vector which maintains a clustering pattern having processes into different clusters.

The following algorithm describes our static-clustering approach.

1: **repeat**
2: $CC_{max} \leftarrow 0$
3: ${C_1}^m \leftarrow 0$
4: ${C_2}^m \leftarrow 0$
5: **for** all $c_i \; \epsilon$ clusters **do**
6: **for** all $c_j \neq c_i \; \epsilon$ clusters **do**
7: **if** $((|c_i| + |c_j|) > maxCS)$ **then**
8: continue

9: **else**
10: $CC_{ij} \leftarrow$ communicationCount(c_i, c_j)
11: $CC \leftarrow CC_{ij} \div (|c_i| + |c_j|)$
12: **if** $(CC > CC_{max})$ **then**
13: $CC_{max} \leftarrow CC$
14: ${C_1}^m \leftarrow c_i$
15: ${C_2}^m \leftarrow c_j$
16: **end if**
17: **end if**
18: **end for**
19: **end for**
20: remove ${C_1}^m$ and ${C_2}^m$ from clusters
21: $c_3 \leftarrow {C_1}^m \cup {C_2}^m$
22: $clusters \leftarrow clusters \cup c_3$
23: **until** $CC_{max} = 0$

We now describe the above algorithm step-by-step in detail. From line 2 to line 4, we initialize CC_{max} which represents the communication count between two clusters and create two empty clusters: ${C_1}^m$ and ${C_2}^m$, which represent those two clusters which will be merged.

The variable $clusters$ contains the current cluster list. Initially $clusters$ contains only single-size clusters. The size of $clusters$ at this stage is equal to the number of processes in the distributed system. Lines 5 and 6 form the nested loop over all clusters within the cluster list. Lines 7 and 8 make sure that any cluster pair whose merger would exceed the maximum permitted cluster size, $maxCS$, is prevented. Line 10 calculates the communication count CC_{ij} between distinct cluster pair, c_i and c_j. Line 11 normalizes the communication count CC_{ij} to the combined size of the pair of clusters, c_i and c_j. Line

12 then determines if this normalized communication count CC_{ij} exceeds the current maximum communication count CC_{max}. If it does, then from line 13 to line 15, the normalized communication count CC_{ij} is set to be the current maximum communication count CC_{max} and the current pair, c_i and c_j, is set to be the potential pair to be merged.

After all pairwise comparisons have been performed (from line 5, the first **for** loop, to line 19), the pair of clusters(${C_1}^m$ and ${C_2}^m$) with the most similarity is found. Therefore, line 20 remove that cluster pair ${C_1}^m$ and ${C_2}^m$ from the cluster list. From line 21 to line 22 the newly merged cluster c_3 which contains ${C_1}^m$ and ${C_2}^m$ is added into the cluster list.

Finally, this algorithm terminates when no 1 pair of clusters can be merged due to the restriction of maximum permitted cluster size, or the communication count between two mergable clusters (satisfing the maximum permitted cluster-size limit) is 0.

The point of this algorithm is that no matter how poor a cluster might be, as long as the new cluster list could reduce the communication count between clusters, and thus reduce the number of cluster-receive events, the pattern of such clusters is better than if it were not constructed. Therefore, the cluster-timestamp algorithm which creates a vector timestamp for each event in the distributed system based on such a cluster strategy could achieve significant reduction in memory consumption.

Our experiments proved that this static cluster strategy succeeds in reducing memory consumption in comparison with the Fidge/Mattern vector timestamp. Another advantage of this strategy is that not only could it apply well to all computation environments, but also it is not sensitive to maximum cluster size at all. This attribute, achieved by this strategy, overcomes a significant shortcoming of Ward's cluster-timestamp algorithm. The experiment results for this algorithm will be presented in Chapter 4.

Now we give the cost analysis for this approach. As we described, initially, the cluster list contains N single-size clusters, where N is the number of processes in the distributed

system. The worst case the time complexity for this algorithm is $O(N^3)$. Since each iteration of the repeat loop (Line 1) will gradually reduce the size of cluster list by one, so the number of iterations of the inside for loops will be reduced too. Here we also need to mention the time complexity for function $communicaitonCount()$, since the maximum cluster size should be restricted within a small range, such as 15 to 50 (that is because in order to reduce the total timestamp size, the cluster timestamps need to be kept small). Therefore, the computation cost for calculating the number of communication between two clusters is constant.

3.2 DYNAMIC CLUSTERING ALGORITHM

In the above section, a static clustering algorithm was presented. The cluster-timestamp algorithm using this clustering strategy is capable of producing timestamp size reduction over Fidge/Mattern timestamps. However, since the clustering strategy is static, the overall timestamp algorithm paired with this strategy must be static.

Generally, a static timestamp algorithm is not sufficient to meet the requirements of distributed-systems-management tools. Since the observation tool is expected to keep operating while the distributed computation is executing, a dynamic timestamp algorithm is required.

Ward's Ph.D. dissertation [26] proposed a semi-dynamic cluster strategy: merge-on-1^{st}-communication. In this strategy, each individual process will be assigned into a single-size cluster initially. When first communication happens between two clusters, these two clusters will be merged, subject to the maximum cluster-size limit. While this strategy is capable of achieving significant space reduction over the Fidge/Mattern timestamp, Ward did not prove there exists a cluster size, or a range of cluster size, that is appropriate for all computations when applying this dynamic clustering strategy. Also,

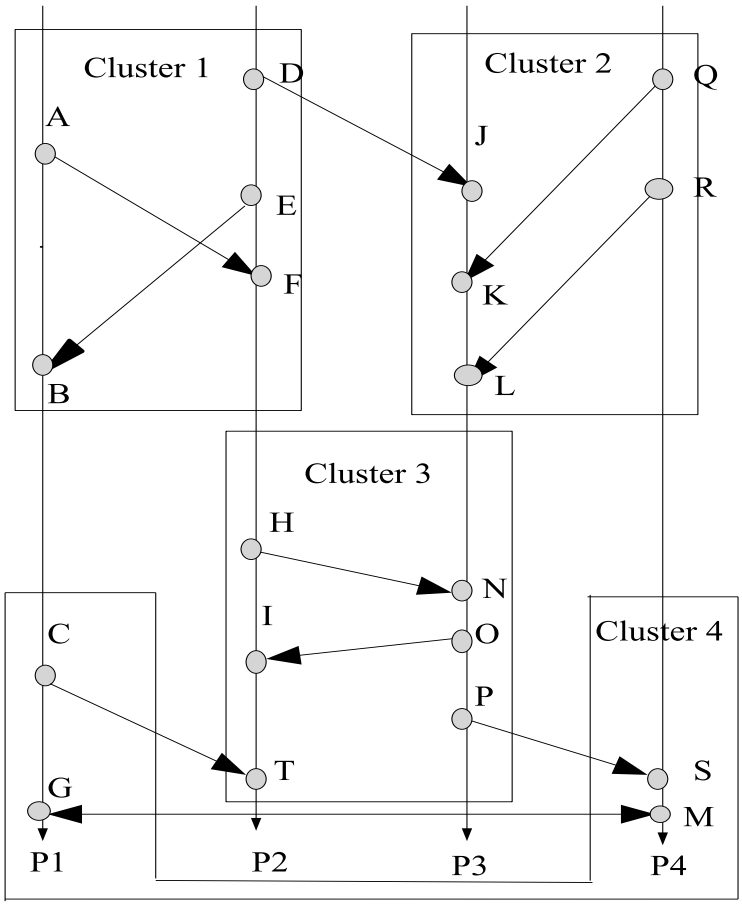


Figure 3.1: Sample for Dynamic Clustering Approach

even if there exists an appropriate cluster size for certain computation environments, it is not known a priori.

We developed a dynamic clustering strategy based on the static clustering algorithm introduced in the previous section. The idea for this dynamic clustering approach is to collect a number of events and then perform the static clustering algorithm. The distributed systems observation tool repeats this operation until the end of the distributed computation. The significant difference between this dynamic clustering approach and previous clustering approaches is that the same process might belong to different clusters

over the course of the computation. As Figure 3.1 shows, process 1 belongs to Cluster 1 and Cluster 4 at different periods. Also the reader should note that the "begin events" of each cluster, such as events H, N, C and S in Figure 3.1, are regarded as cluster-receive events in order to simplify the precedence test.

The input for this dynamic clustering algorithm is all events occurred in distributed system. The output is a vector which maintains clustering pattern for each "n" events. The clustering patterns contains processes into different clusters.

The corresponding dynamic timestamp-creation operation will be described later. We describe the dynamic clustering algorithm as follows:

1: **repeat**
2: $storeEvent(event)$
3: $count++$
4: **if** $count\%n == 0$ **then**
5: **repeat**
6: $CC_{max} \leftarrow 0$
7: ${C_1}^m \leftarrow 0$
8: ${C_2}^m \leftarrow 0$
9: **for** all $c_i \; \epsilon$ clusters **do**
10: **for** all $c_j \neq c_i \; \epsilon$ clusters **do**
11: **if** $((|c_i| + |c_j|) > maxCS)$ **then**
12: continue
13: **else**
14: $CC_{ij} \leftarrow$ communicationCount(c_i, c_j)
15: $CC \leftarrow CC_{ij} \div (|c_i| + |c_j|)$
16: **if** $(CC > CC_{max})$ **then**
17: $CC_{max} \leftarrow CC$

18: $C_1{}^m \leftarrow c_i$

19: $C_2{}^m \leftarrow c_j$

20: **end if**

21: **end if**

22: **end for**

23: **end for**

24: remove $C_1{}^m$ and $C_2{}^m$ from clusters

25: $c_3 \leftarrow C_1{}^m \cup C_2{}^m$

26: $clusters \leftarrow clusters \cup c_3$

27: **until** $CC_{max} = 0$

28: $clearEvent()$

29: **end if**

30: **until** no more new events

From the above, we can see that this algorithm keeps repeating until there are no further events occurring within the distributed system. In line 2 we store each new event occurring in the distributed computation. Line 3 calculates the number of events which have been stored. When there are n events available (line 4), we cluster these n events into clusters (lines 5 to 27) by applying the static clustering algorithm. After that, we clear the stored events (line 28). This procedure repeats until there are no more new events occurring in the distributed system.

The static clustering algorithm we described in previous section is to cluster all processes within distributed system. In these processes, some have communication with other processes, some do not. So the time complexity for static clustering algorithm is $O(N^3)$ where N is the number of all processes in distributed system.

In this dynamic clustering algorithm, C is the number of processes containing events within that set of n events. We collect certain amount of events (which is defined as

n events) before performing the clustering procedure. It is obvious that $C \leqq n$ since different events could occur in the same process. We need to make n to be an appropriate number. If this number is too big, C could be big. Then the total number of steps in the clustering operation will be large. This will cause poor performance of this dynamic clustering algorithm. However if this number is too small, all these n events might be included just in one cluster, so the hierarchical clustering strategy is useless because n is too small. In such a case it might produce poor clusters. Also, if those n events are distributed into different processes, many of them will be begin events requring a Fidge/Mattern timestamp.

The time complexity of this dynamic algorithm for each event is $O(C^3/n)$. Since C is smaller than n, $O(C^3/n)$ can be reduced to $O(C^2)$. Since $C = kN$ where N is the total number of processes in the distributed system and k is a constant, then $O(C^2) = O(N^2)$.

3.3 OPTIMIZED DYNAMIC CLUSTERING ALGORITHM

In Section 3.2 we described the dynamic clustering algorithm and analyzed the time complexity of this algorithm. The reader should note that in this dynamic clustering algorithm a new clustering pattern is generated for each n events. As a result, the computation cost is high ($O(N^2)$). Also when the clustering changes, there are an extra C begin events generated by the new clustering pattern. However, after we get the initial clustering pattern, it may not be necessary to perform reclustering for each n events. The reclustering strategy can only result in improved clustering when the number of cluster-receive events (created by leaving these n events in the current clustering patterns) exceeds the number of begin events (created by reclustering). Based on this thought, we created an optimized dynamic clustering algorithm which can avoid unnecessary reclustering. Therefore this

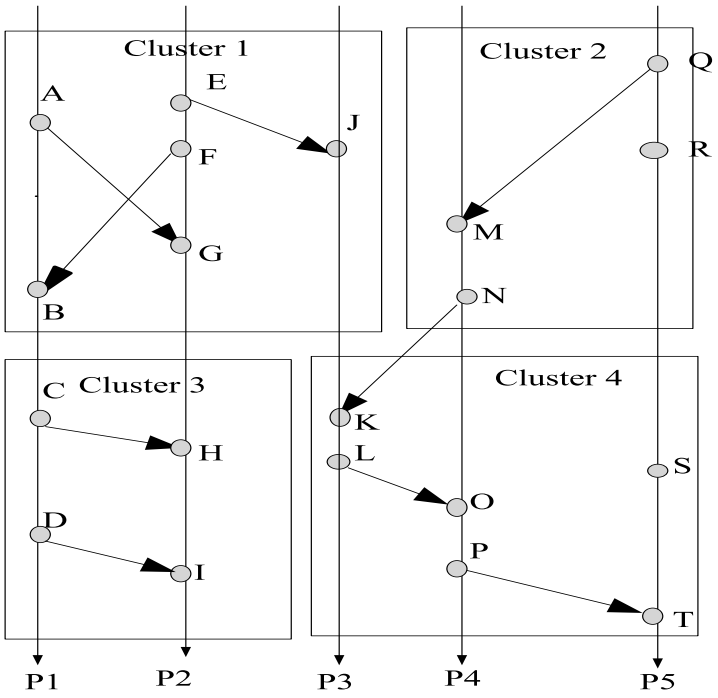


Figure 3.2: Cluster Pattern Sample for Dynamic Clustering Approach

optimized approach can speed up the whole clustering process.

The basic idea for this optimized dynamic clustering algorithm is as follows. First, after a significant number of events have been collected, they are clustered and an initial clustering pattern is generated. After that, for each following new event that occurs in the distributed system, if it is not a cluster-receive event according to current clustering pattern, no change is needed. Otherwise, if it is a cluster-receive event, from now on, each new event will be stored until half n events have been collected. Then we calculate the difference between the number of cluster-receive events (created by leaving these half n events in the previous clustering pattern) and the number of begin events (created by reclustering). If the difference does not exceed a threshold for change, we leave these half n events in the current clusters. Otherwise, if the difference exceeds the threshold for

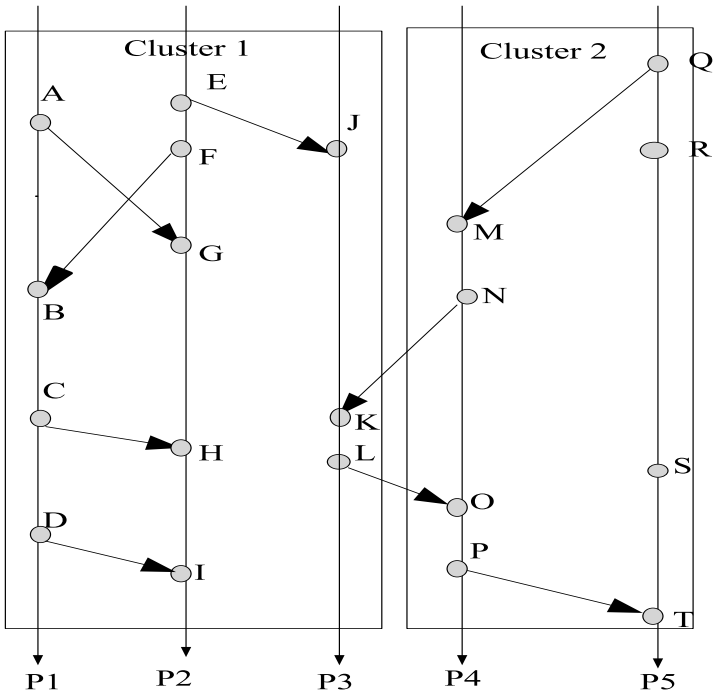


Figure 3.3: Clustering Pattern Sample for Optimized Dynamic Clustering Approach

change, we continue collecting events until n events have been collected. Then we do the same check again. If the difference exceeds the threshold for change, these n events will be reclustered. We then calculate the total timestamp size based on these clusters, and compare this result with the result generated by leaving these events into previous clusters instead of grouping them into new clusters. After comparison, the cluster choice which generates the least timestamp size will be kept for further clustering computation. If the difference does not exceeds the threshold, then the reclustering will not be necessary. These n events will be left in the previous cluster pattern.

Figure 3.2 shows the clustering pattern generated by the dynamic clustering algorithm described in Section 3.2 (In this sample $n = 10$). In this cluster pattern, there are 10 begin events. Figure 3.3 shows the clustering pattern generated by the optimized

dynamic clustering algorithm. In this clustering pattern, there are 5 begin events and 2 cluster-receive events.

The input for this optimized dynamic clustering algorithm is all events occurred in distributed system. The output is a vector which maintains a clustering pattern for each "n" events. The clustering patterns contains processes into different clusters.

This optimized dynamic clustering algorithm is described as follows:

```
1: repeat
2:    count++
3:    if doneFirstCluster == False then
4:       storeEvent(event, &eventList, &processList)
5:       if getEnoughEvent(eventList) == True then
6:          clusterEvents(eventList)
7:          timestamp(eventList)
8:          empty(&eventList, &processList)
9:          doneFirstCluster = True
10:      end if
11:   else
12:      if startStoreEvent == False then
13:         if isCr(event) == True then
14:            startStoreEvent = True
15:            countList = 0
16:            procCount = 0
17:            bECount = 0
18:            crCount = 0
19:         else
20:            timestamp(event) //gets cluster timestamp
```

21: **end if**
22: **end if**
23: **if** $startStoreEvent == True$ **then**
24: $countList++$
25: $procCount++$ if not seen this process containing this event in processList
26: $bECount++$ if not seen this process containing this event in current clustering pattern
27: **if** $isCR(event) == True$ **then**
28: $crCount++$
29: **end if**
30: $storeEvent(event, \&eventList, \&processList)$
31: **end if**
32: **if** $startStoreEvent == True$ and $countList >= n/2$ and $crCount \leq procCount$ **then**
33: $timestamp(eventList)$
34: $empty(\&eventList, \&processList)$
35: $startStoreEvent == False$
36: **else if** $startStoreEvent == True$ and $countList == n$ and $procCount \leq crCount$ **then**
37: $sumTS\_1$ = $(coutList - crCount - bECount) * maxCS$ + $(crCount + bECount) * N$
38: $sumTS\_2$ = $reCluster(eventList)$
39: **if** $sumTS\_1 > sumTS\_2$ **then**
40: $clusterCopy(0)$
41: **end if**
42: $timestamp(eventList)$

43: $empty(\&eventList, \&processList)$
44: $startStoreEvent = False$
45: **end if**
46: **end if**
47: **until** No more new events

Now we describe the above algorithm step by step. As we can see from Line 1, this algorithm keeps running when there are new events occurring in the distributed computation. Line 2 counts the total number of events. The scope between line 3 and line 10 shows how the algorithm creates the initial clustering patterns. Line 3 tells if the inital clustering procedure is done. If it is not, then in Line 4 the event is stored into an event list. The process containing this event is stored into a process list. The function $getEnoughEvent()$ in Line 5 tells if a certain suitable amount of events have been collected for the initial clustering. If this function returns true, then in Line 6 the function $cluster(eventList)$ applies the static clustering algorithm to these events in the $eventList$, creating the initial clustering pattern. According to this clustering pattern, these events in $eventList$ are timestamped in Line 7. This $timestamp()$ function will be described in Section 3.4. Then, these events in $eventList$ and these processes in $processList$ are clear up by calling function $empty(\&eventList, \&processList)$ in Line 8. In Line 9, $doneFirstCluster$ is assigned to be $True$, which means we have done the initial clustering procedure.

From Line 11 to Line 47, the algorithm deals with events following those used in the initial clustering. The scope from Line 12 to Line 22 determines if a reclustering operation should be trigered. Line 12 determines if a variable $startStoreEvent$ is $False$. If it is, then Line 13 determines if this $event$ is a cluster-receive event according to current clustering pattern. If it is, then a potential reclustering operation is triggered. The variable $startStoreEvent$ is assigned to be $True$ in line 14. A variable $countList$, which

is used to count the number of events in the $eventList$, is initialized to be 0 in line 15. A variable $procCount$ which is used to count the number of process in $processList$, is initialized to be 0 in Line 16. In line 17, a variable $bECount$ used to count the number of processes which are in the $processList$ but not in the current clustering pattern. A variable $crCount$, which is used to count the number of cluster-receive events, is initialied to be 0 in line 18. Otherwise, if this event is not a cluster-receive event according to current clustering pattern, this event is cluster-timestamped in line 20.

Lines 23 to 31 show that after a reclustering operation is trigered the event is stored into the $eventList$ and the number of events in the $eventList$ is counted. Also $procCount$ is incremented by 1 in line 25 if the process containing this event is not in the $processList$. Similarly $bECount$ is incremented by 1 in line 26 if the process which contains this event is not in current clustering pattern. Lines 27 to 29 show that $crCount$ is incremented by 1 if this $event$ is a cluster-receive event. The $eventList$ and $processList$ are updated in line 30.

Line 32 determines if there are more than half of "n events" in the $eventList$ and if these events are worth being reclustered (if $crCount$ is bigger than $procCount$; $crCount$ and $procCount$ represent the number of cluster-receive events (created by leaving events in the current clustering patterns) and the number of begin events (created by reclustering these events) respectively). If these events in the $eventList$ are not worth being reclustered, all events in the $eventList$ are cluster timestamped according to current clustering pattern in line 33. Line 34 empties all events in the $eventList$ and all processes in the $processList$. The variable $startStoreEvent$ is assigned to be $False$ in line 35, which means the algorithm gives up the potential reclustering operation.

Line 36 determines if there are n events in the $eventList$ and if these events are potentially worth being reclustered (again by comparing the difference between $crCount$ and $procCount$). If the potential reclutering operation is necessary, in line 37 the algo-

rithm calculates the total timestamp size by leaving these n events in the current clusters. Note that N is the total number of processes in the distributed system and $maxCS$ is the maximum cluster size. This total timestamp size is assigned to the variable $sumTS\_1$. In line 38 the function $reCluster(eventList)$ calculates the total timestamp size after reclustering these n events in new clusters (applying the static clustering mechanism described in Section 3.1). Then in lines 39 to 41 the cluster pattern which can generate the least total timestamp size will be kept for further computation. These n events are cluster timestamped according to prefered cluster pattern in line 42. Line 43 calls $empty(\&eventList, \&processList)$ to empty all events in the $eventList$ and all processes in the $processList$. The variable $startStoreEvent$ is assigned to be $False$ in line 44, which means the algorithm has finished the reclustering operation.

The time complexity of this optimized dynamic algorithm for each event is $O(N^2)$ for worst case, which is identical to the time complexity of dynamic algorithm. In the best case, the time complexity will be $O(N^2)$ for each event which is clustered initially. For the remaining events, the time complexity is $O(clusterSize)$ for any non-cluster-receive event and $O(N)$ for any cluster-receive event.

3.3.1 DESCRIPTION OF $isCr(event)$ FUNCTION

This $isCr(event)$ is a very important function in our optimized dynamic clustering algorithm since it determines when the reclustering operation should be trigered. After the initial clustering, there might be some new events occurring in some new processes which are not included in the initial clustering pattern. This function also determines if we should put these new processes into the existing clusters or put them into new clusters.

In this subsection, we describe the algorithm of the function $isCr(event)$. Before we describe the algorithm of $isCr(event)$, a few notation points will help readers understand the following pseudo-code. The variable $maxCS$ is referred to as the maximum cluster

size limit. We define $eTrace()$ and $pTrace()$ methods in class $event$ as returning the trace ID where this event occurs and its corresponding partner trace ID, respectively. We also define $unary()$ method in class $event$ as returning $True$ if this $event$ is a unary event or $False$ otherwise, and we define $transmit()$ method in class $event$ as returning $True$ if this $event$ is a send event or $False$ otherwise. The $clusters[]$ array is defined as a list containing clusters. We define the method $trace2clusterID(eTrace)$ as returning the ID of the cluster to which process $eTrace$ belongs. The function $exist(clusterId)$ is defined as returning $True$ if there is a cluster with this clusterId in the clusters list, and $False$ otherwise. There are some methods that belong to class $cluster$, such as $size()$ which is defined as returning the number of processes in this cluster, $add(trace)$ which is defined as appending process $trace$ to this cluster. A function $lastCId()$ is defined as returning the clusterId of last cluster in cluster list. If the size of this cluster reaches the maximum cluster size, a new empty cluster will be created and inserted into this cluster list, its clusterId is returned.

The input of this algorithm is an event. The vector $clusters[]$ is a global variable maintaining the current clustering pattern. After operating this function, the output is $True$ if $event$ is a cluster-receive event or $False$ otherwise. Also, the vector $clusters[]$ is updated if $event$ occurred in a process which is not included in the current clustering pattern.

This $isCr(event)$ function is described as follows:

1: $isCr(event)\{$
2: **if** $event.unary()$ or $event.transmit()$ **then**
3: return $False$
4: **else**
5: $eTrace = event.eTrace()$
6: $pTrace = event.pTrace()$

7: **if** $trace2clusterID(eTrace) == trace2clusterID(pTrace)$ and $exist(trace2clusterID(eTrace))$ **then**
8: return $False$
9: **else if** $trace2clusterID(eTrace) \neq trace2clusterID(pTrace)$ and $exist(trace2clusterID(eTrace))$ **then**
10: **if** $exist(trace2clusterID(pTrace))$ **then**
11: return $True$
12: **else**
13: **if** $clusters\big[trace2clusterID(eTrace)\big].size() < masCS$ **then**
14: $clusters\big[trace2clusterID(eTrace)\big].add(pTrace)$
15: return $False$
16: **else**
17: clusters$\big[lastCId()\big]$ .add($pTrace$)
18: return $True$
19: **end if**
20: **end if**
21: **else if** $exist(trace2clusterID(eTrace)) == False$ **then**
22: **if** $clusters\big[trace2clusterID(pTrace)\big].size() < maxCS$ and $exist(trace2clusterID(pTrace))$ **then**
23: $clusters\big[trace2clusterID(pTrace)\big].add(eTrace)$
24: return $False$
25: **else if** $clusters\big[trace2clusterID(pTrace)\big].size() \geq maxCS$ and $exist(trace2clusterID(pTrace))$ **then**
26: $clusters\big[lastCId()\big].add(eTrace)$
27: return $True$
28: **else if** $exist(trace2clusterID(pTrace)) == False$ **then**

29: $clusters\big[lastCId()\big].add(eTrace)$

30: $clusters\big[lastCId()\big].add(pTrace)$

31: $trace2clusterID(eTrace) == trace2clusterID(pTrace)$? return $False$: return $True$

32: **end if**

33: **end if**

34: **end if**

35: }

Now, we describe the above algorithm step by step. As we can see from line 2, if $event$ is a unary event or a send event, $False$ is returned. The scope between line 4 and line 33 shows how the algorithm determines if this $event$ is a cluster-receive event or a cluster event when this $event$ is not a unary event or a send event. Line 5 and 6 assign this $event$'s process id and its partner event's process id to $eTrace$ and $pTrace$ respectively. Line 7 determines if $eTrace$ and $pTrace$ belong to same cluster and $eTrace$ belongs to an existing cluster. If it is true, which means this $event$ is a cluster event, $False$ is returned in line 8.

The scope between line 9 and 20 shows how the algorithm deals with the scenario that $eTrace$ belongs to an existing cluster, to which $pTrace$ does not belong. Line 10 determines if $pTrace$ belongs to an existing cluster. If it does, this $event$ is a cluster-receive event, and $True$ is returned in line 11. Otherwise, if $pTrace$ does not belong to any existing cluster, if the size of the cluster which contains $eTrace$ is smaller than $maxCS$, $pTrace$ is inserted into this cluster in line 14, and $False$ is returned in line 15. If the size of the cluster which contains $eTrace$ reaches the maximum cluster size limit, $pTrace$ is inserted into the last cluster in the cluster list in line 17, $False$ is returned in line 18, since $eTrace$ and $pTrace$ are in different clusters.

The scope between line 21 and 33 shows how this function deals with the scenario

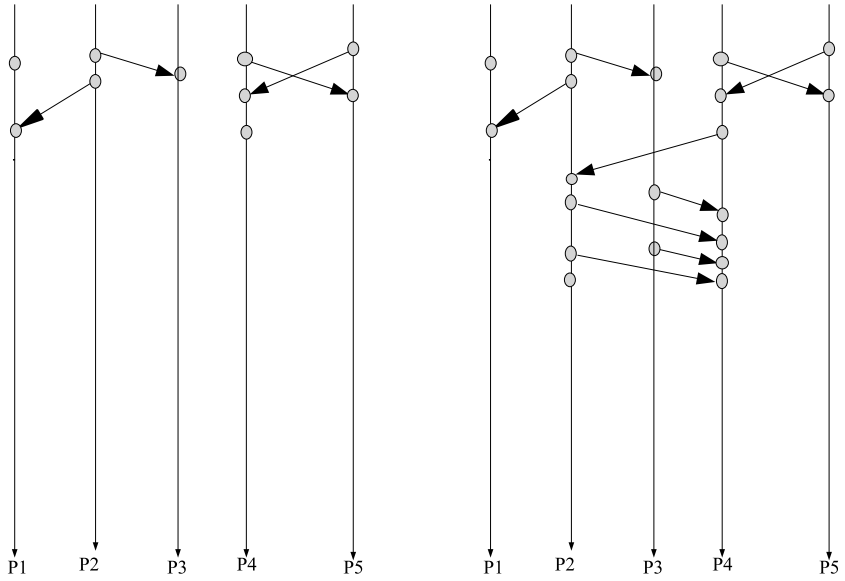
that $eTrace$ does not belong to any existing cluster. Line 22 determines if $pTrace$ belongs to an existing cluster whose size is smaller than $maxCS$. If it does, $eTrace$ is inserted into the same cluster which contains $pTrace$ in line 23, and $False$ is returned in line 24, since both $eTrace$ and $pTrace$ now belong to the same cluster. Otherwise, if the size of the cluster which contains $pTrace$ reaches the maximum cluster size limit, $eTrace$ is inserted into the last cluster in the cluster list in line 26, and $False$ is returned in line 27. Finally, if both $eTrace$ and $pTrace$ do not belong to any existing cluster, from line 29 to 30 $eTrace$ and $pTrace$ are inserted into the last cluster in the cluster list. Then, line 31 determines if they belong to same cluster. If they do, $False$ is returned. Otherwise, $True$ is returned.

3.3.2 Clustering Pattern for Sample Distributed-system Computation

In this subsection, we present figures to show how our algorithm works. Figure 3.4 shows a sample of distributed-system computation. Subfigures a, b and c in sequence present the distributed-compuation pattern when we process events in group of 10 ($n = 10$).

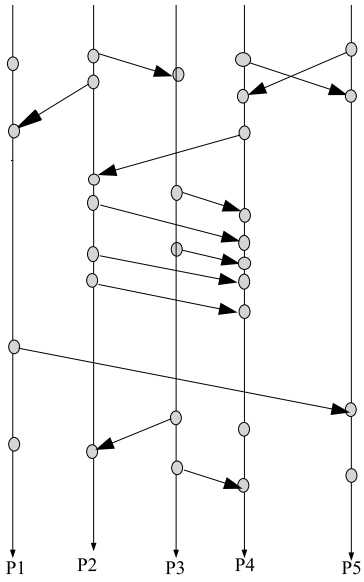
Figure 3.5 shows the clustering pattern after applying our optimized dynamic clustering algorithm to this sample distributed-system compuation.

Another idea about how to organize this clustering pattern regarding this sample distributed-system computation is presented in Figure 3.6. From this figure we can see that this clustering pattern further reduces memory consumption, since there are 9 cluster-receive events and begin events according to the clustering pattern in Figure 3.6, but there are total 10 cluster-receive and begin events according to the clustering pattern in Figure 3.5. However this clustering pattern may cost more computation cost to construct. Also it might make precedence test more complicated. We do not explore this



(a) Stage 1 of Sample Distributed-system Compuation

(b) Stage 2 of Sample Distributed-system Compuation



(c) Stage 3 of Sample Distributed-system Compuation

Figure 3.4: Sample Distributed-system Computation

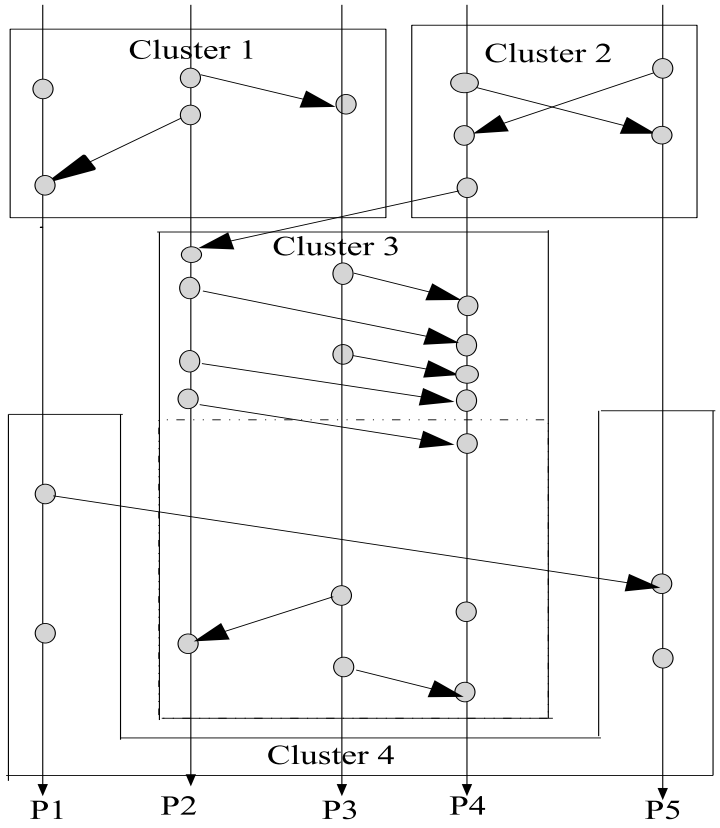


Figure 3.5: Clustering Pattern for Sample Distributed-system Computation

idea further in this thesis.

3.4 DYNAMIC CLUSTER TIMESTAMPING

The dynamic cluster timestamping algorithm assigns Fidge/Mattern timestamp to cluster receive events and begin events. The remaining events are assigned cluster timestamps by the projecting Fidge/Mattern vector timestamp over the processes in the cluster.

The timestamp-creation algorithm is shown in the following pseudo-code. Before we describe the algorithm, a few notation points will help readers to understand this algo-

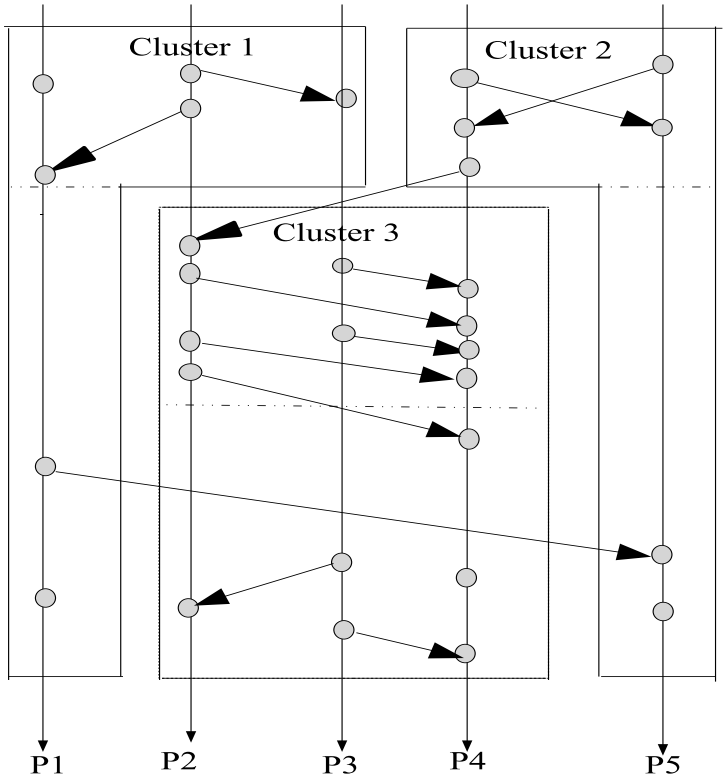


Figure 3.6: Clustering Pattern for Sample Distributed-system Computation

rithm. The Fidge/Mattern timestamp of an event e is referred to as FM_e. The timestamp vector for event e is referred to as T_e, while RB_e is the reference to event e's greatest-predecessor cluster receive or begin event, whichever is more recent. This algorithm as a whole maintains a vector of all greatest cluster-receive events and/or begin events in all processes. $GCRB$ refers to this vector, which is initialized to 0. All processes in the distributed computation are enumerated from 0 to $N-1$ and all clusters are enumerated from 0 to $C-1$. We then defined $clusterID(e)$ as a method returning the ID of the cluster to which event e belongs. An event belongs to only one cluster, which means that the clusters do not overlap. The method $clusterSize(e)$ returns the size of the cluster to which event e belongs. The method $PC(e)$ returns a vector of process IDs. The cluster

to which event e belongs contains these processes. The method to return the ID of the process/processes (for sychronous events) on which event e occurs is referred to as $P(e)$. The method $partner(e)$ returns the partner event of event e. $B(e)$ is a method returning a boolean value to tell if event e is a begin event or not. Finally, the notation $f <: e$ is used to specify that event f is an immediate predecessor of event e. That is , there is no intermediate event that precedes e and that is preceded by f.

The input for this algorithm is an event e without being assigned any timestamps. Some global variables:

1. All events f where $f <: e$ (which denotes that event f is an immediate predecessor of event e). These events have cluster timestamp and Fidge/Mattern timestamp. The Fidge/Mattern timestamp will be used to calculated the Fidge/Mattern timestamp for event e;
2. All events g where $g <: f$ (which denotes that event g is an immediate predecessor of event f). These events should only have cluster timestamp;
3. A vector $GCRB[]$ which maintains the greatest cluster-receive or begin events in each process;
4. A vector $BeginEvents[]$ which contains all begin events. It is used in function $B(e)$ to tell if event e is a begin event;
5. A vector maintaining the clustering pattern is used in function $clusterID(e)$ to determine which cluster event e belongs to.

The output for this algorithm is

1. For all events f where $f <: e$ (which denotes that event f is an immediate predecessor of event e) their Fidge/Mattern timestamp are deleted. The corresponding occupied memory is freed;

2. The event e with Fidge/Mattern timestamp if e is a cluster-receive or begin event. Or event e with Fidge/Mattern timestamp and cluster timestamp if e is not a cluster-receive or begin event (the Fidge/Mattern timestamp will be deleted later);

3. The event e is inserted into vector $GCRB[]$ if this event e is a cluster-receive event or begin event.

1: timestamp(e){
2: $FM_e \leftarrow$ fidgemattern(e);
3: **if** ($\exists_f f <: e$ and $clusterID(e) \neq clusterID(f)$ and $clusterID(e) \neq clusterID(partner(e))$) or ($B(e) == TRUE$) **then**
4: $T_e \leftarrow FM_e$;
5: GCRB$\big[P(e)\big] \leftarrow FM_e\big[P(e)\big]$;
6: **else**
7: **for** $i \leftarrow 0; i < clusterSize(e); i++$ **do**
8: $T_e\big[i\big] \leftarrow FM_e\big[PC(e)[i]\big]$;
9: **end for**
10: $RB_e \leftarrow GCRB\big[P(e)\big]$;
11: **end if**
12: **for** $\forall_f f <: e$ **do**
13: **if** e is last covering event of f and $B(f) \neq TRUE$ **then**
14: **if** $\forall_g (g <: f \Longrightarrow clusterID(g) == clusterID(f))$ **then**
15: delete FM_f;
16: **end if**
17: **end if**
18: **end for**
19: }

Now we step-by-step describe the above algorithm. The algorithm first computes the Fidge/Mattern timestamp (Line 2) by using the method described in Section 2.1 in Chapter 2. For events that are cluster receives or begin events (Line 3), it simply uses the Fidge/Mattern value as the timestamp (Line 4), and adjust the vector of greatest cluster receives and begin events to identify this new event (Line 5). Otherwise, for events that are neither cluster receives nor begin events, they are assigned a cluster timestamp by projecting the Fidge/Mattern timestamp over the cluster processes (Line 7 to 9). This is sufficient to determine precedence within the cluster. Finally, from Lines 12 to 18, for any event that immediately precedes event e for which e is the last possible covering event, if it is not cluster receive or begin event, its Fidge/Mattern timestamp is deleted since it is no longer needed.

The time complexity for this algorithm is $O(N)$, where N is the number of processes. This computation cost is caused by the need to compute the Fidge/Mattern timestamps in line 2.

3.5 DYNAMIC CLUSTER-TIMESTAMP PRECEDENCE TEST

In the above section, we described the dynamic cluster timestamping algorithm. Now, we introduce the cluster-timestamp precedence test algorithm in this section.

The basic idea behind this algorithm is for two events e and f, if they are in the same cluster or they do not belong to same cluster but they occurred on same process, or event f is a cluster receive or begin event, then the standard Fidge/Mattern precedence test applies, with parameters adjusted to map the processes to timestamp entries precisely. If this is not the case, then this precedence test returns true $if - and - only - if$ there exists a cluster receive event or begin event which precedes event f and is a successor to event e.

The dynamic cluster-timestamp precedence test algorithm $precedenceTest(evente, eventf)$ is shown in the following pseudo-code, it returns $TRUE$ if $e \rightarrow f$ and returns $FALSE$ otherwise. Before we describe the algorithm, a few notation points will help readers to understand this algorithm. The Fidge/Mattern timestamp of an event e is referred to as FM_e. The timestamp vector for event e is referred to as T_e. RB_e is the reference to event e's greatest-predecessor cluster receive or begin event which occurred in the same process as event e. All processes in the distributed computation are enumerated from 0 to $N-1$ and all clusters are enumerated from 0 to $C-1$. We then defined $clusterID(e)$ as a method returning the ID of the cluster to which event e belongs. An event belongs to only one cluster, which means that the clusters do not overlap. The method $clusterSize(e)$ returns size of the cluster to which event e belongs. The method to return the ID of the process on which event e occurs is referred to as $P(e)$. The method $PC(e)$ returns a vector of processes ID. The cluster to which event e belongs contains these processes. The method $I(PC(e), P(e))$ returns the index i, if $P(e)$ is within its cluster. Therefore $P(e) = PC(e)[i]$. We also define method $IPRE(e, pID)$ to return a reference to the event which is the greatest predecessor to event e in process pID.

The input of this algorithm are events e and f. The event f's correspondingly immediate predecessor and their greatest-predecessor cluster-receive or begin events are global variables. The output is $True$ if event e happened before event f and $False$ otherwise.

1: precedenceTest(event e, event f){
2: **if** $clusterID(e)$ == $clusterID(f)$ **then**
3: return $T_e\big[I(PC(e), P(e))\big] < T_f\big[I(PC(e), P(e))\big]$;
4: **else if** $P(e) == P(f)$ **then**
5: return $T_e\big[I(PC(e), P(e))\big] < T_f\big[I(PC(f), P(f))\big]$;
6: **else if** $f == RB_f$ **then**
7: return $T_e\big[I(PC(e), P(e))\big] < T_f\big[I(PC(e), P(e))\big]$;

```
 8: else
 9:    for i ← 0; i < clusterSize(f); i++ do
10:       event g ← IPRE(f, PC(f)[i]);
11:       event r ← RB_g;
12:       if T_e[I(PC(e), P(e))] < FM_r[P(e)] then
13:          return TRUE;
14:       end if
15:    end for
16: end if
17: return FALSE;
18: }
```

Now we walk through the above code line by line. In line 2, the algorithm determines if events e and f belong to same cluster. If they do, then we use the Fidge/Mattern timestamp test (line 3), adjusting the offset into the timestamp vector according to the position of the traces in the cluster. In line 4, we determine if events e and f are in different clusters but they belong to same process. If they are, we can apply Fidge/Mattern timestamp test to them, with parameters adjusted to map the processes to timestamp vector entries appropriately (line 5). We then determine if event f is a cluster receive or begin event. If it is, we also can apply the Fidge/Mattern timestamp test (line 6). In all these cases the precedence test can be done in constant time. Otherwise, if events e and f are in different clusters and they do not belong to same trace, and event f is neither a cluster receive nor a begin event (line 8), we use for a loop to go through all processes in the cluster which contains event f. In line 10, we determine event g which is the greatest predecessor to event f in process $PC(f)[i]$. We then determine event r which is the greatest cluster receive or begin event prior to event g in process $PC(f)[i]$ (line 11). In line 12, we apply the Fidge/Mattern test to determine precedence. As soon as we get the

result the algorithm ceases. The algorithm returns $FALSE$ if event e does not precede event f (line 17).

The computation cost of this precedence test depends on the relative locations and precedence relationships between the two events. If the two events are in same cluster, or they are in same process or if event f is a cluster receive or begin event, the time complexity is constant. If this is not the case, then the cost is $O(clusterSize(f))$ in the worst case, being on average less than half this if $e \rightarrow f$.

3.6 CONCLUSION

In this chapter, we introduced the static, dynamic and optimized dynamic clustering algorithms. The time complexity of the static and dynamic clustering algorithms is O(N^3) where N is the number of processes in the distributed system. The time complexity of this optimized dynamic clustering algorithm for each event is O(N^2) for worst case. In the best case, the time complexity will be O(N^2) for each event which is clustered initially. For the remaining events, the time complexity is O($clusterSize$) for any non-cluster-receive event and O(N) for any cluster-receive event. Therefore the optimized dynamic clustering algorithm can speed up the clustering over the dynamic clustering approach. It is demonstrated in our experiments.

We also introduced the dynamic cluster timestamping algorithm and the precedence test algorithm. In this timestamping algorithm, an event in the distributed system can be timestamped within O(N) time complexity. In this precedence test algorithm, precedence between two events can be decided in constant time for best case or O($clusterSize(event)$) for worst case.

4 EXPERIMENTAL EVALUATION

In this chapter, we present the experimental evaluation of our clustering algorithm. We evaluated our algorithms over several dozen distributed computations covering a variety of different parallel, concurrent and distributed environments with up to 300 processes. We focus this evaluation on PVM [8], the Parallel Virtual Machine, and Java [9]. The PVM programs tended to be SPMD-style parallel computations. Therefore, they typicaly present close neighbour communication and scatter-gather patterns. The Java programs were web-like applications, including various web-server executions.

All of the following experiments concern timestamp size, rather than precedence-test computation time. Except for the Fidge/Mattern timestamp algorithm, all algorithms under comparison have a common parameter: the maximum cluster size. We therefore varied this parameter from 2 to 50 processes and observed the ratio of the number of cluster-receive events to total events and the ratio of average cluster-timestamp size to the Fidge/Mattern-timestamp size. We implemented the simulation following the POET approach which assumes that the observation tool encodes timestamps using a fixed-size vector. By default this is 300. The cluster timestamps were assumed to be encoded using a vector of size equal to the maximum cluster size. These assumptions are consistent with the current behavior of existing observation tools. We calculate the average cluster-timestamp size by the following equation:

$$((eventCount - crCount - beCount) * cluster\_size + (crCount + beCount) * 300) / eventCount$$

where $eventCount$ is the total number of events in the computation sample, $crCount$ is the the number of cluster-receive events, $beCount$ is the number of begin events, and

$cluster\_size$ is the maximum cluster size.

Our analysis is divided according to the clustering algorithm applied. First, we evaluated the static-clustering algorithm described in Section 3.1, comparing the evaluation result with the output of the fixed-size clustering algorithm. We do this comparison because both of them are static clustering strategies. We then evaluate the dynamic clustering algorithms described in Section 3.2 and 3.3, comparing the evaluation results with the experimental output of merge-on-first-communication and merge-on-5^{th}-communication. We put them together since all of them are dynamic clustering approaches. In the last section, we compare all algorithms described in Sections 3.1, 3.2 and 3.3 since the optimized dynamic algorithm derives from the dynamic algorithm and the dynamic algorithm derives from the static clustering algorithm. Readers should note that all algorithms we mentioned above are compared against the Fidge/Mattern algorithm as well.

4.1 STATIC CLUSTERING ALGORITHM

In this section we compare between the simulation result from the static clustering algorithm described in Section 3.1, and the fixed-size clustering algorithm. In the fixed-size clustering algorithm, for a cluster-size cs, the first cs processes received by the tool were placed in the first cluster, the next cs traces in the second cluster, and so forth.

PVM sample results for the fixed-size clustering algorithm and the static-clustering algorithm are shown in Figure 4.1. In this figure, we can see the ratio of the number of cluster-receive events to total-event count in PVM by applying the fixed-size clustering algorithm and the static clustering algorithm. It is obvious that the cluster-receive ratio is high when the cluster size is small. In the PVM computation environment, when we apply the fixed-size clustering algorithm, the highest ratio reaches about 28% and the lowest ratio is about 12%. When we applied the static-clustering algorithm, the highest

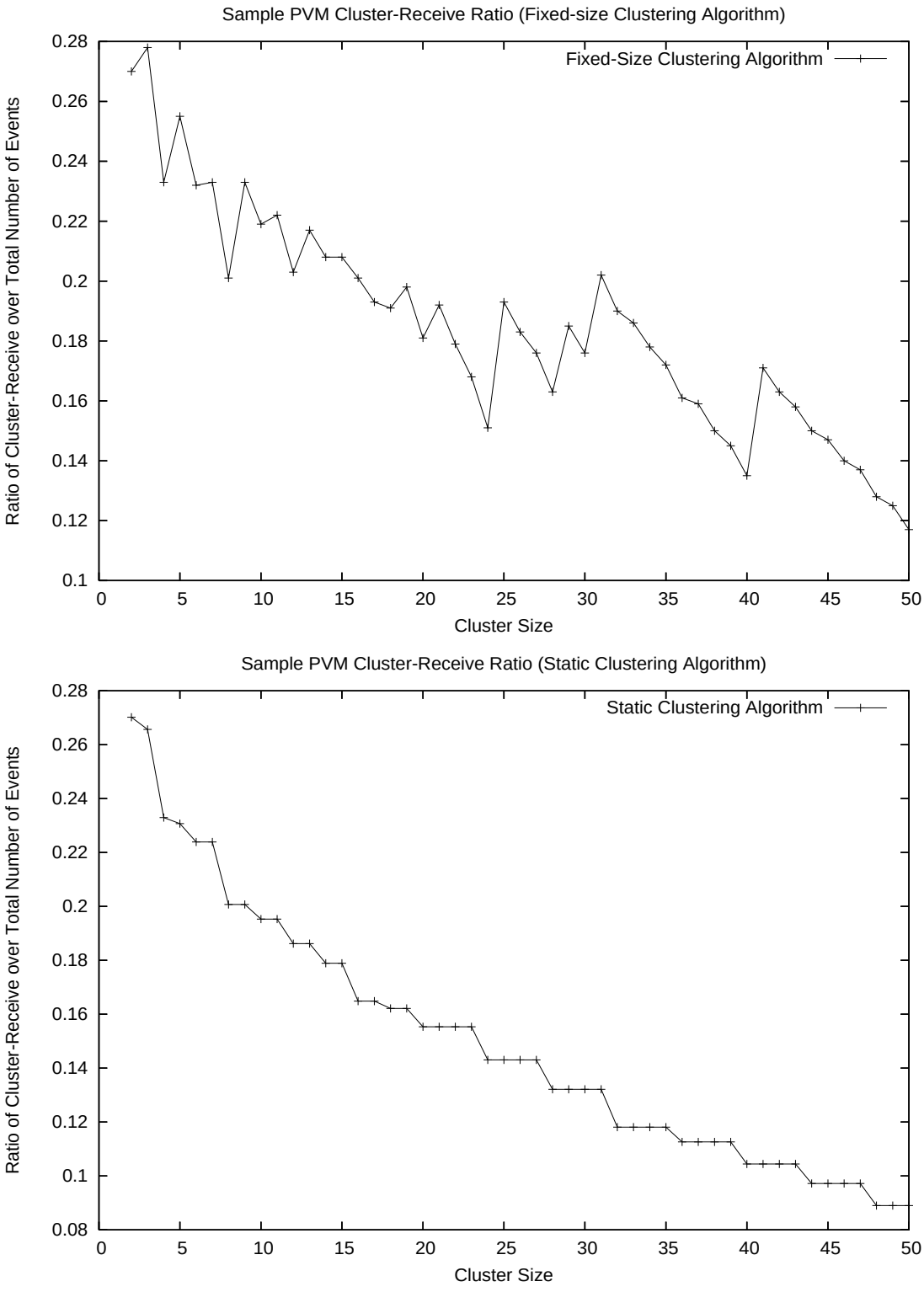


Figure 4.1: Sample Cluster-Receive Ratio for PVM Sample

and lowest ratio in PVM computation environment are about 27% and 9% respectively.

Figure 4.2 shows a couple of sample results for Java, showing the ratio of the number of cluster-receive events to total events count by applying the fixed-size clustering algorithm and the static-clustering algorithm. It is obvious that static clustering algorithm has better performance in cluster-receive ratio than fixed-size clustering algorithm does, in this figure we found much smoother decline in the cluster receive ratio. By applying the fixed-size clustering algorithm, the highest ratio reaches about 75% and the lowest is about 9%. By applying the static-clustering algorithm, the highest and lowest ratio are about 40% and 5%.

From the upper graphs of Figures 4.1 and 4.2, we note that the problem for the fixed-size clustering algorithm is the lack of consistency. For example, in Java sample, when the maximum cluster size increases from 18 to 19, the cluster-receive ratio increases from about 19.4% to 70%.

In order to make a more clear comparison of cluster-receive ratio between the fixed-size clustering algorithm and the static clustering algorithm, we aggregated the results shown on Figures 4.1 and 4.2, summarizing them in the following table. According to the data in the table, the static cluster algorithm reduced the average cluster-receive ratio in PVM computation environment by 3%. The static cluster algorithm works even better in the Java computation environment. It reduced the average cluster-receive ratio from 32.1% to 9.7%.

(PVM)	Minimum(%)	Average(%)	Maximum(%)	Variance
Fixed-Size	11.7	18.5	27.8	3.64
Static	8.9	15.0	27.0	4.7
(JAVA)	Minimum(%)	Average(%)	Maximum(%)	Variance
Fixed-Size	9.1	32.1	74.4	24.3
Static	5.2	9.7	40.4	5.48

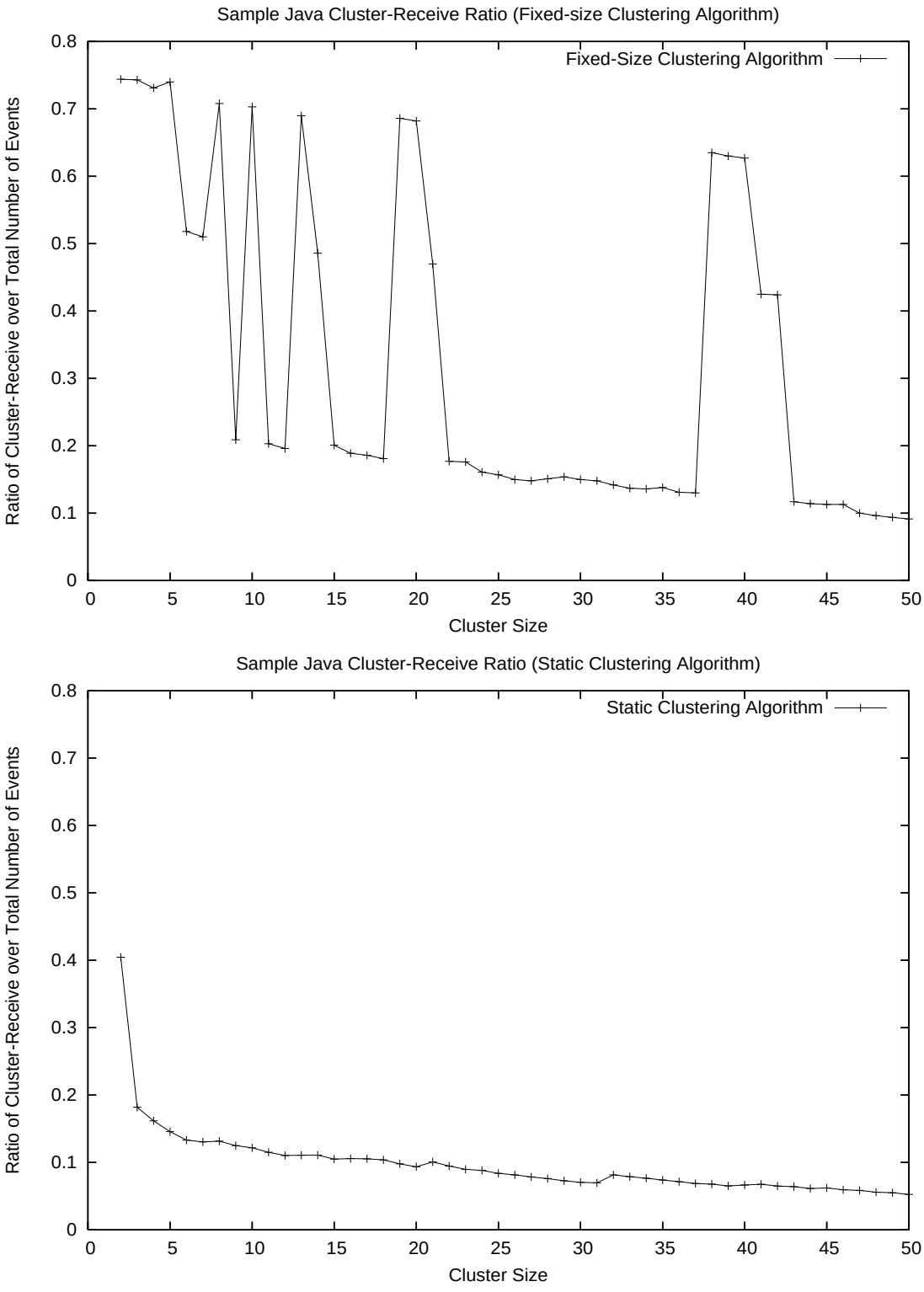


Figure 4.2: Sample Cluster-Receive Ratio for Java Sample

The value of Variance is calculated by the equation:

$$variance = \sqrt{\frac{\sum_{s=2}^{50}((i-a)*100)^2}{N}}$$

where i is the cluster-receive ratio corresponding to difference cluster size (varied from $2 \sim 50$), a is average cluster-receive ratio and N is 49 (from cluster size 2 to cluster size 50). We note that static clustering has dramatically reduced variance in the Java environment.

We now compare the approaches regarding the average timestamp size. A couple of sample results for the fixed-size clustering algorithm and static clustering algorithm for distinct PVM and Java computations are shown in Figure 4.3. The figure shows the ratio of the average cluster timestamp size to the Fidge/Mattern timestamp size.

A couple of comments must be made about these results. First of all, we note that the static algorithm produced the least space consumption in both samples. In the upper figure, we note that the ratio is under 25% for a maximum cluster size from 5 processes to 50 processes when applying the static clustering algorithm. The average ratio of the fixed-size cluster timestamp size to Fidge/Mattern timestamp size for this sample PVM computation is about 26%. However the average ratio is only 23% when we applied our static clustering algorithm. Thus our static algorithm reduced the average timestamp size by an additional 3% compared to the fixed-size cluster method, which matches the 3% difference in cluster-receive ratio.

For the sample Java computation the difference is even more significant. After summarizing the data shown on the lower figure, the average ratio of the fixed-size cluster timestamp size to Fidge/Mattern timestamp size for this computation is 39%. This ratio is reduced to 18% by applying the static clustering algorithm.

Second, the results shown on the Figure 4.3 suggests that there exists a cluster algo-

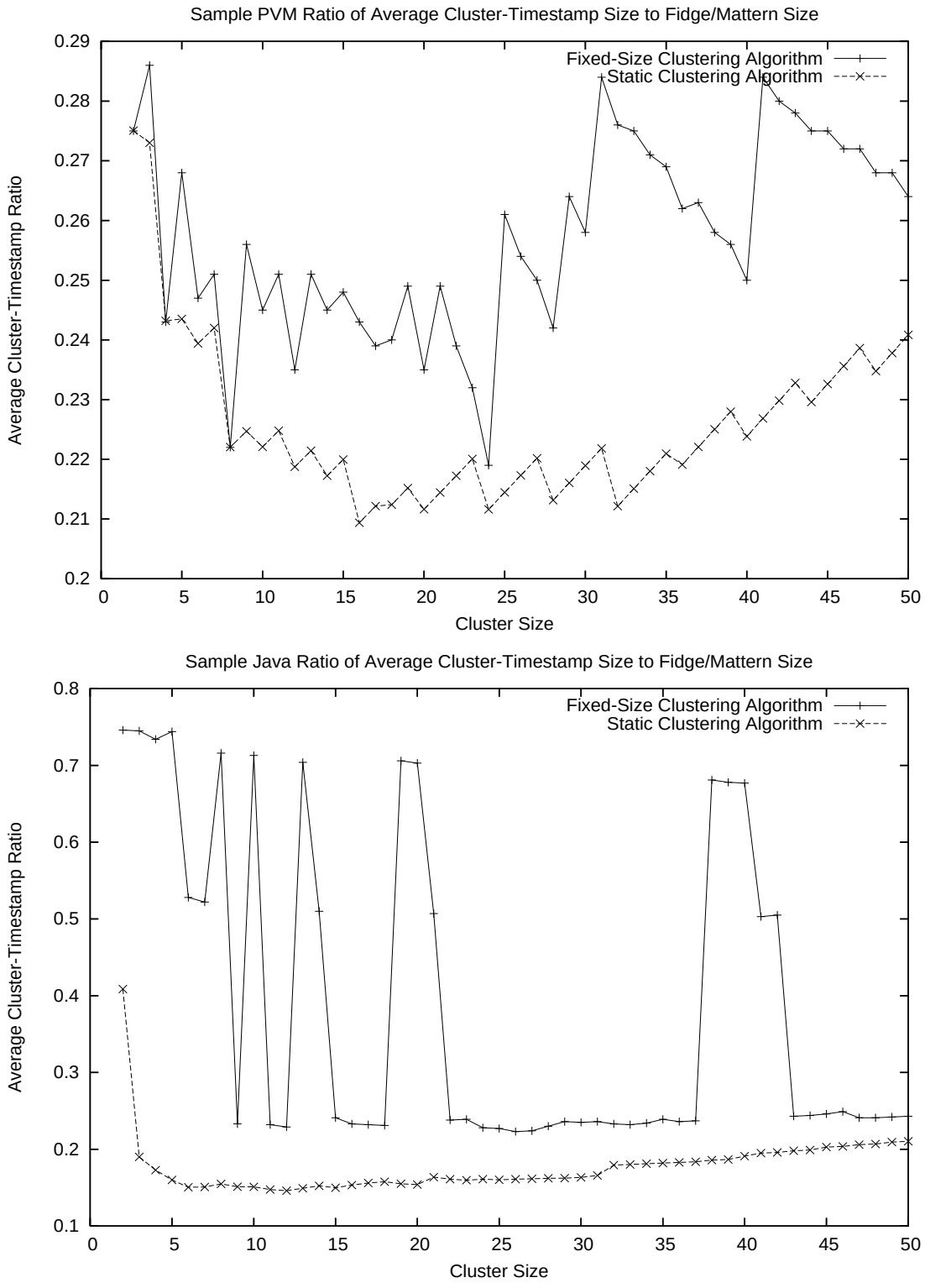


Figure 4.3: Sample Average Cluster Timestamp Size

rithm which is not significantly sensitive to the maximum cluster-size selected. Figure 4.3 shows that the static clustering algorithm we created produced a relatively smooth ratio curves. This smooth curve shown in this Figure is typical of all the Java results. These curves illustrate that the static clustering algorithm does not have the sensitivity to cluster size that the fixed-size clustering algorithm exhibits.

To provide more evidence in support of our claims, for all PVM and Java computations we studied, we found that the static clustering algorithm produced better results in reducing space consumption. Also we computed the range of maximum cluster size for which the timestamp size ratio was within 20% of the best timestamp size ratio achieved. After studying these computations, we determined that for all computations a cluster size of 13 or 14 resulted in a timestamp size ratio that was within 20% of the best achievable. No such range exists for the fixed-size clustering algorithm.

4.2 DYNAMIC CLUSTERING ALGORITHM

In this section, we evaluate our dynamic clustering algorithms. We compare the dynamic clustering algorithm described in Section 3.2, the optimized dynamic clustering algorithm described in Section 3.3, and the prior dynamic clustering approaches merge-on-1^{th}-communication and merge-on-5^{th}-communication.

Before we present the simulation results, the reader should recollect that the basic mechanism of the dynamic clustering algorithm is to recluster processes after collecting a certain number of events. After some study, we determined that 200 is a suitable number, which we therefore use in our simulation.

Sample results showing the ratio of the average timestamp size over Fidge/Mattern timestamp size using the dynamic clustering algorithm and optimized dynamic clustering algorithm, merge-on-first-communication, and merge-on-5^{th}-communication are

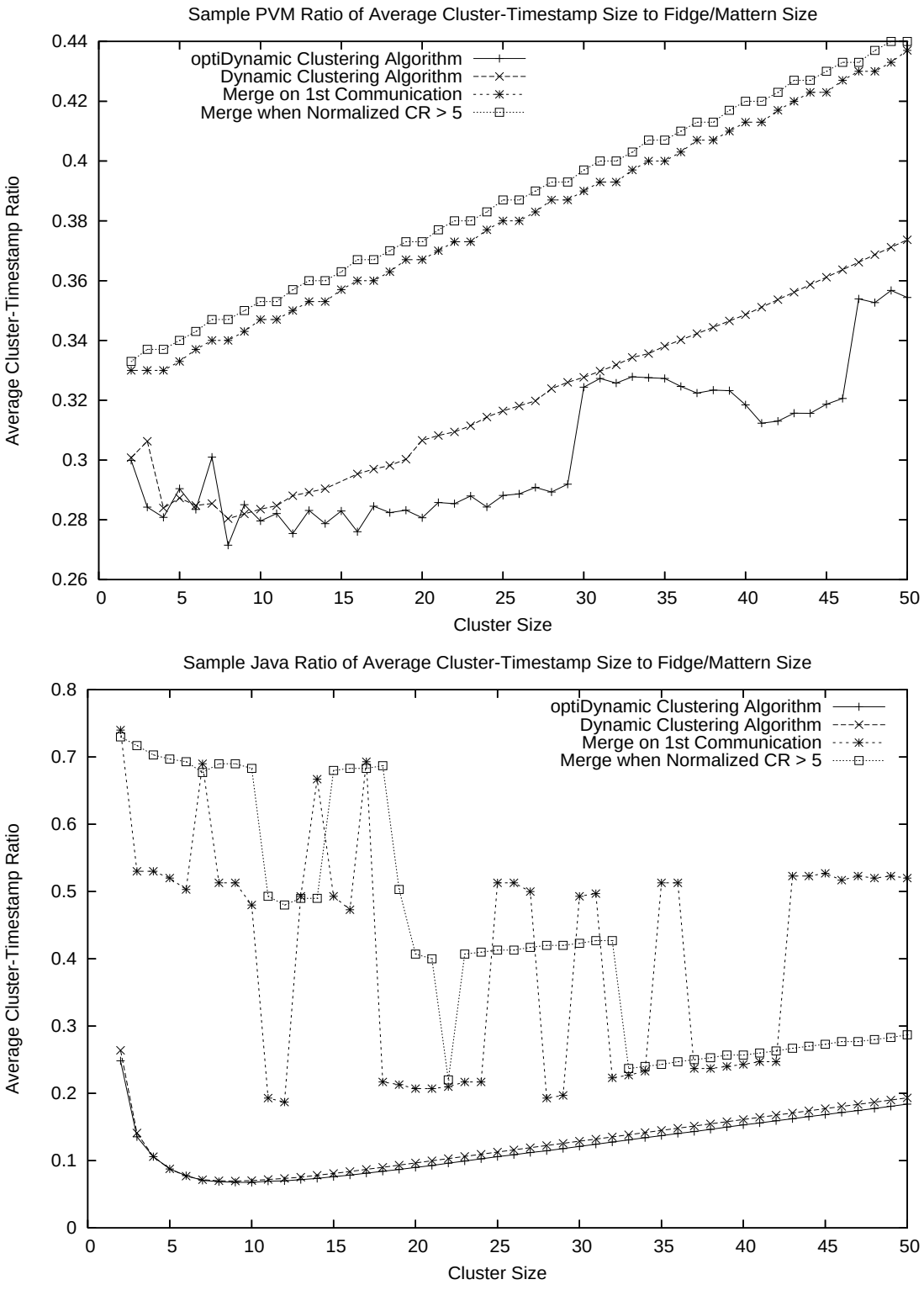


Figure 4.4: Sample Cluster Receive Ratio (Dynamic Clustering Algorithms)

presented in Figure 4.4. The sample PVM and Java computations presented in this figure are the same as those sample computations presented in Section 4.1.

In Figure 4.4 we note that our dynamic clustering algorithms achieve better results than the self-merge approaches. Also, the ratio curve produced by the optimized dynamic clustering algorithm is below the ratio curve generated by the dynamic clustering algorithm in most instances. This is reasonable since the mechanism of optimized dynamic clustering algorithm is designed based on the dynamic clustering algorithm, though their initial clustering mechanisms are different.

Both curves generated by self-merge clustering algorithms in upper graphic in Figure 4.4 are smooth and straight up all the time. This appears to be because the computation has a very large number of events, but the number of cluster-receive events becomes fixed once the maximum cluster size starts from 2 (for merge-on-1^{st}-communication) or reaches 3 (for merge-when-normalized CR$>$5). Also in the upper graph, we note that the optimized dynamic clustering algorithm produced ratio curve about 2% lower than the curve generated by the dynamic clustering algorithm between maximum cluster size $24 \sim 29$ and $39 \sim 47$. The reader should also note that it is flat between cluster-size $10 \sim 29$, with average cluster-timestamp ratio between this range of 28% and the variance in this range of 0.45(generated by variance equation applied in Section 4.1).

The lower graph in Figure 4.4 presents the simulation result for a sample Java computation which is same as in Section 4.1. We note these self-merge algorithms did not always create smooth curve as much as might be desired, and even sometimes when it does so, it creates smooth curve at a notably higher timestamp size. Our dynamic clustering strategies produced fundamentally different ratio curves from those curves generated by the self-merge strategies. Two major observations need be described. First, our dynamic clustering algorithms always smooth the curve. The second observation is that our dynamic strategies always produce less average timestamp size at any maximum cluster

size than the self-merge approaches did.

When performing experiments we noticed that when we apply the optimized dynamic-clustering algorithm to the PVM sample, when the maximum cluster size is between $8 \sim 50$, there is no re-clustering operation occurring. However, when we apply dynamic clustering algorithm to PVM there are 10 re-clustering operations. Similarly, when we apply optimized dynamic clustering algorithm to Java sample, there is about $51 \sim 58$ re-clustering operations (when the maximum cluster size is from 3 to 17) and $36 \sim 51$ re-clustering operations (when the maximum cluster size is from 18 to 50). However, when we applied dynamic clustering algorithm to Java sample, there are 130 re-clustering operations. Therefore, the optimized dynamic clustering algorithm can greatly speed up the clustering process.

4.3 OUR CLUSTERING ALGORITHMS

In this section, we compare our static clustering algorithm, dynamic clustering algorithm, and optimized dynamic algorithm which are described in Sections 3.1, 3.2 and 3.3, respectively.

Sample results using our clustering algorithms are presented in Figures 4.5 and 4.6. The sample computations represented in these figures are the same as those computations used in Sections 4.1 and 4.2. Several observations can be made about these results. First, the upper graph in Figure 4.5 shows the ratio of cluster-receive events over total events for the sample PVM computation. We note that the ratio from the static clustering algorithm is less than the one generated by the dynamic clustering algorithms. This might be because the dynamic clustering algorithm generated more cluster-receive events in this sample PVM computation which include cluster-receive events, begin events for each cluster, and those events not being clustered (those unclustered events happen at

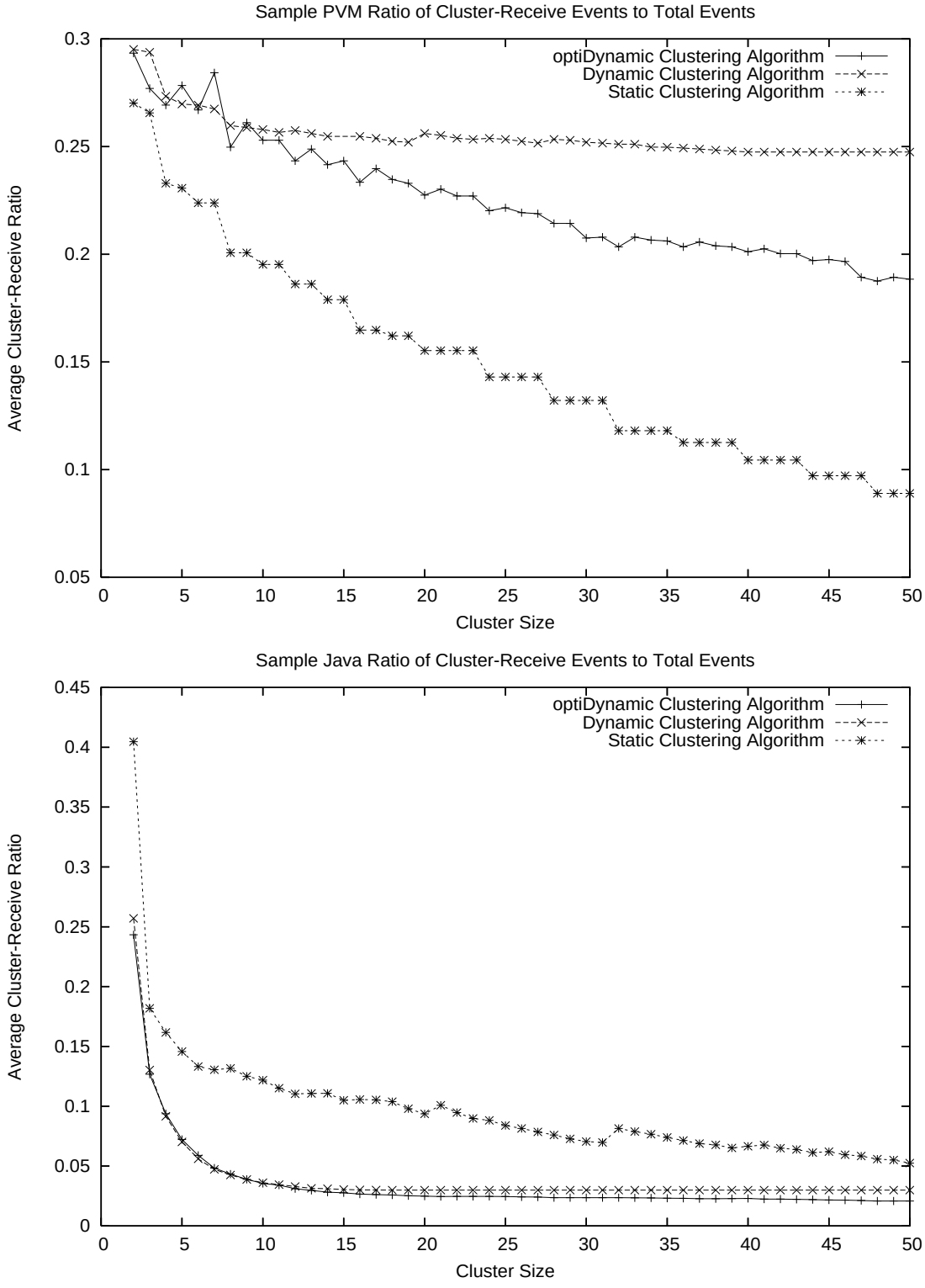


Figure 4.5: Sample Cluster Receive Ratio(Our Clustering Algorithms)

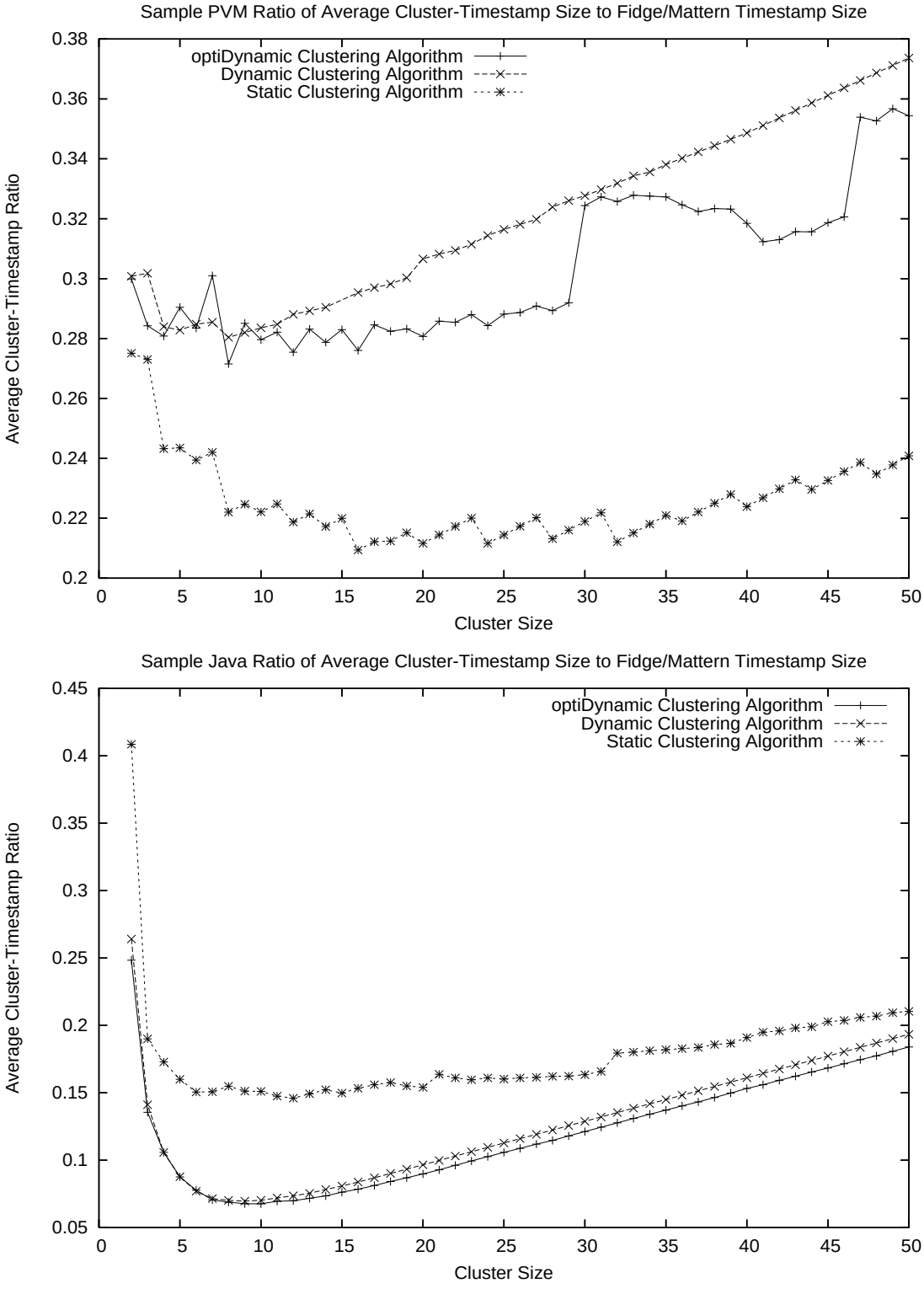


Figure 4.6: Sample Average Timestamp Size Ratio(Our Clustering Algorithms)

the end of the distributed computation where the number of these events is less than the number of so-called “n events” required for a reclustering). Since the static clustering algorithm generated less cluster-receive events than the dynamic cluster algorithm did in the sample PVM computation, the static clustering algorithm produced a smaller average timestamp size. This phenomenon is presented in the upper graph in Figure 4.6.

Second, we also note the phenomenon that happened in the sample PVM computation did not occur in the Java computation. As we see in the lower graph in Figure 4.5, the dynamic clustering algorithms generated less cluster-receive events than the static clustering approach did. Therefore, the dynamic clustering algorithms perform better in reducing the average timestamp size, as shown in the lower graph in Figure 4.6. The reason that the dynamic clustering algorithms perform better in Java is that the clustering of the computation shifts over course of the computation, and these reclustering operations can generate better clustering patterns, which produce less average timestamp size than the previous clustering pattern.

Third, we note that all our clustering algorithms produced relatively smooth average timestamp ratio curves, which demonstrates that our clustering approaches can work well and are not inherently sensitive to cluster size selection.

5 CONCLUSION AND FUTURE WORK

In this thesis, we have developed algorithms to enable a scalable partial-order data structure for distributed-system management. It provided the following contributions to scientific and engineering knowledge.

1. The creation of a static clustering algorithm that is suitable for all computations and greatly reduces the timestamps size. This static algorithm allowed us to demonstrate that there do exist cluster-size ranges over which timestamp size reduction is stable;
2. The creation of a dynamic cluster-timestamp creation algorithm and an optimized dynamic cluster-timestamp creation algorithm;
3. Theoretical cost and experimental analysis of the dynamic cluster-timestamp creation algorithm.

This thesis has presented static, dynamic, and optimized dynamic clustering approachs for cluster timestamp creation. In the static clustering algorithm, initially each process is dumped into each single cluster, then clusters are merged based on distributed-computation pattern subject to the maximum cluster size limit. The static clustering algorithm ususally is implemented when the distributed compuation is over. Unlike the static clustering algorithm, the dynamic one is implemented dynamically. The core idea of the dynamic approach is to implement the static clustering algorithm after collecting each **n** events until the distributed computation terminates. The optimized dynamic clustering algorithm is based on the dynamic clustering algorithm, but it is designed to avoid unnecessary reclustering. Therefore this approach speeds up the whole clustering

process. Unlike other vector timestamp algorithms, our algorithms achieve signicant reduction in memory consumption and insensibility to cluster size. We have demonstrated these advantages of our algorithms by presenting experimental results.

5.1 FUTURE WORK

The possible future work is to develop an alternate dynamic approach, in which processes will be permitted to migrate between clusters in the situation that it is obvious that the clustering originally selected is a poor one. The difference between this approach and our dynamic clustering algorithms is our algorithms iterate necessary reclustering operations against each “n” events to construct new clustering patterns, this alternate approach will ajust the original clustering pattern by moving processes into different clusters. The major challenge of this approach is how to adjust the cluster timestamps of those events that occurred on the migrating processes.

Specifically, as described in Section 3.3.2, there are other ways to organize the clustering pattern which might achieve better memory reduction.

Finally integration of our dynamic cluster creation algorithm into the POET system would validate the work in real systems. As we mentioned, the POET system applies the fixed-size Fidge/Mattern timestamps in order to achieve the distributed-system precedence testing capability. By replacing the Fidge/Mattern timestamps in POET with our cluster timestamps, we are able to gather data from significantly larger computations.

BIBLIOGRAPHY

[1] Websphere application server, object level trace. Technical report, IBM Corporation, 1998.

[2] System performance visualization tool user's guide. Technical Report 001, Intel Corporation, 1993.

[3] Peter C.Bates. Debugging heterogenous distributed systems using event-based models of behaviour. In *ACM SIGPLAN Notices*, volume 24, pages 11–22, 1989.

[4] Colin.J.Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference, Brisbane*, pages 56–66, 1988.

[5] Colin Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.

[6] 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.

[7] Vijay K. Garg and Chakarat Skawratananond. Timestamping messages in synchronous computations. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems*, pages 552–559. IEEE Computer Society Press, 2002.

[8] Al Geist, Adam Begulin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM : Parallel virtual machine. MIT Press Cambridge Massachusetts, 1994.

[9] James Gosling, Bill Joy, and Guy Steele. The java language specification. Addison-Wesley, 1996.

[10] Boris Gruschke. A new approach for event correlation based on dependency graphs. In *Proceedings of the 5th Workshop of the OpenView University Association*, OVUA, April 1998.

[11] Pankaj Jalote. *Fault Tolerance in Distributed System*. Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[12] Claude Jard and Guy-Vincent Jourdan. Dependency tracking and filtering in distributed computations. Technical report, IRISA, Campus de Beaulieu, 35042 Rennes Cedex-France, Aug,1994.

[13] Kaufman and Rousseeuw. Finding groups in data: An introduction to cluster analysis. John Wiley and Sons, 1990.

[14] Dieter Kranzlmuller, Siegfried Grabner, and Jens Volkert. Race condition detection with the MAD environment. In *Second Australasian Conference on Parallel and Real-Time Systems*, pages 160–166, 1995.

[15] Dieter Kranzlmuller, Siegfried Grabner, and Jens Volkert. Debugging with the MAD environment. *Journal of Parallel Computing*, 23(1-2):199–217, 1997.

[16] Thomas Kunz, James P. Black, David J. Taylor, and Twan A.Basten. Poet: Target-system independent visualisations of complex distributed-application executions. *The Computer Journal*, 40(8):499–512, 1997.

[17] Thomas Kunz and James P.Black. Using automatic process clustering for design recovery and distributed debugging. *Software Engineering*, 21(6):515–527, 1995.

[18] L. Lamport. Time,clocks and the ordering of events in a distributed system. In *Comms. ACM*, volume 21, pages 558–65, 1978.

[19] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V., 1988.

[20] Michel Raynal and Mukesh Singhal. Capturing causality in distributed systems. *IEEE Computer*, 29(2):49–56, 1996.

[21] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. In *Information Processing Letters*, volume 43, pages 47–52, August 1992.

[22] Andrew S.Tanenbaum. *Distributed Operating System*, volume 1, pages 1–33. 1995.

[23] James Alexander Summers. Precedence-preserving abstraction for distributed debugging. Master's thesis, University of Waterloo, 1992.

[24] P. Ward and D. Taylor. Self-organizing hierarchical cluster timestamps. In *EuroPar'01 Parallel Processing*, pages 46–56. Lecture Notes in Computer Science 2150, 2001.

[25] P. Ward and D. Taylor. A hierarchical cluster algorithm for dynamic, centralized timestamps. In *Proceedings of the 21st Conference on Distributed Computing Systems*, pages 585–593. IEEE Computer Society Press, April, 2001.

[26] Paul A.S Ward. *A Scalable Partial-Order Data Structure for Distributed-System Observation*. PhD thesis, University of Waterloo, 2001.