A Scalable Partial-Order Data Structure for Distributed-System Observation

by

Paul A.S. Ward

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2001

ABSTRACT

Distributed-system observation is foundational to understanding and controlling distributed computations. Existing tools for distributed-system observation are constrained in the size of computation that they can observe by three fundamental problems. They lack scalable information collection, scalable data-structures for storing and querying the information collected, and scalable information-abstraction schemes. This dissertation addresses the second of these problems.

Two core problems were identified in providing a scalable data structure. First, in spite of the existence of several distributed-system-observation tools, the requirements of such a structure were not well-defined. Rather, current tools appear to be built on the basis of events as the core data structure. Events were assigned logical timestamps, typically Fidge/Mattern, as needed to capture causality. Algorithms then took advantage of additional properties of these timestamps that are not explicit in the formal semantics. This dissertation defines the data-structure interface precisely, and goes some way toward reworking algorithms in terms of that interface.

The second problem is providing an efficient, scalable implementation for the defined data structure. The key issue in solving this is to provide a scalable precedence-test operation. Current tools use the Fidge/Mattern timestamp for this. While this provides a constant-time test, it requires space per event equal to the number of processes. As the number of processes increases, the space consumption becomes sufficient to affect the precedence-test time because of caching effects. It also becomes problematic when the timestamps need to be copied between processes or written to a file. Worse, existing theory suggested that the space-consumption requirement of Fidge/Mattern timestamps was optimal. In this dissertation we present two alternate timestamp algorithms that require substantially less space than does the Fidge/Mattern algorithm.

ACKNOWLEDGEMENTS

Notwithstanding the claim on page ii, a dissertation is the product of many individuals. First and foremost I thank my adviser, Dr. David Taylor, who has enabled me to pursue this research, both by providing excellent advice and the financial support that I needed for my growing family. Likewise, I wish to thank the various folk at IBM Canada, who provided both financial support and a motive for this research.

David Taylor is but one of a long succession of teachers who have influenced and encouraged me over the years, all of whom deserve my gratitude. I would like to thank the various faculty of both the University of Waterloo and the University of New Brunswick. Similarly, the teachers of Dalhousie Regional High School and King Edward VII Grammar School provided me with an excellent education; the foundation necessary to pursue advanced research.

While teachers enable learning, friends and family provide encouragement. Ellen Liu and David Evans deserve special mention, as do all the members of the Distributed Systems and Networks Group at the University of Waterloo. Over the years my parents have guided, encouraged and, as needed, pushed me in my educational endeavors. In this, my most recent pursuit, they have been joined by my wife, Shona, and children, Jonathan, Rebecca, James and Sarah.

Support is not motivation, and while IBM Canada provided an immediate motive for this specific research, they could not provide the inner drive needed to achieve a doctorate. This I attribute to my paternal grandfather and uncles who, from the time of my birth, it seems, have always encouraged me in creativity and excellence.

Finally, but by no means least, I wish to thank the members of my committee, Jay Black, Ken Salem, Kostas Kontogiannis and Richard LeBlanc, whose comments and criticisms have done much to improve this dissertation. Jay Black, in particular, has be invaluable, stepping in while David was on sabbatical.

For Shona

A wife of noble character is worth far more than rubies.

And for our children, Jonathan, Rebecca, James and Sarah

Wisdom is supreme; therefore get wisdom. Though it cost all you have, get understanding.
Esteem her, and she will exalt you; embrace her, and she will honor you.

CONTENTS

FIGURES

Tables

1 INTRODUCTION

Distributed-system management is defined as the dynamic observation of a distributed computation and the use of information gained by that observation to dynamically control the computation [99]. Distributed-system observation consists of collecting runtime data from executing computations and then presenting that data in a queryable form for computation-control purposes. The purpose of the control is varied and includes debugging [8, 21, 25, 35], testing [28, 29, 76, 176], computation visualization [37, 67, 71, 73, 89, 149, 151], computation steering [38, 58, 81, 118, 158, 159, 160], program understanding [72, 93, 94], fault management [57, 79], and dependable distributed computing [69, 119].

Tools for distributed-system management, such as POET [95, 147], Object-Level Trace [65] and the MAD environment [86, 87], can be broadly described as having the architecture shown in Figure 1.1. The various components are defined as follows.

The distributed system is the system under observation. Collecting together various researchers' views of what constitutes a distributed system, we arrive at the following general consensus. A distributed system is a system composed of loosely coupled machines that do not share system resources but rather are connected *via* some form of communication network. The communication channels are, relative to the processing capacity, low bandwidth and high latency [110]. Note that while network bandwidths are improving, latency remains high. Further, because of Moore's law [108], it is not clear that network bandwidths are improving relative to processor performance. Likewise, it is clear that wide-area latency will only get worse relative to processor performance because of the laws of physics [36].

Both the machines and the network may be faulty in various ways [15]. The failure of any machine or portion of the network in any particular way does not imply the failure of the whole system. Indeed, partial failure is one of the most difficult aspects to deal with in distributed systems [162].

A distributed computation is any computation that executes over such a distributed system. Again, the consensus view is that the distributed computation is composed of multiple processes communicating *via* message passing. The processes are physically distributed [97], their number may vary over time [131] and they are sequential (that is to say, the actions that occur within any process are totally ordered). The requirement for sequentiality is somewhat problematic at this stage because of the prevalence of multi-threading. A multi-threading model is fundamentally a shared-memory model, which is very different from a message-passing model. While some work has started to address this issue [142, 39], we do not deal with it in this dissertation. The communication may be synchronous or asynchronous, point-to-point (also referred to as unicast), multicast or broadcast. The logical network is a fully connected graph (*i.e.*, any process may communicate directly with any other process). In practice, most processes will communicate with just a few other processes, and the computation will have a distinct topology. There is no other knowledge of the underlying physical network [125]. The underlying physical network may

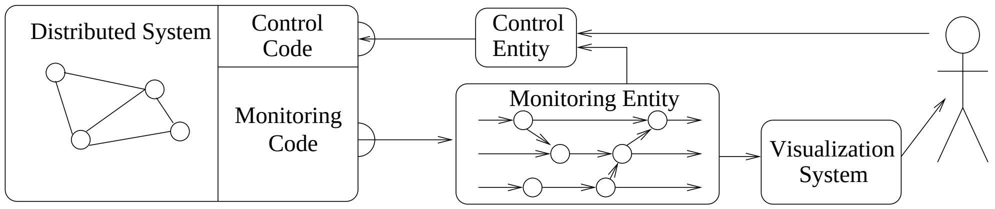


Figure 1.1: Distributed-System Observation and Control

not be (probably is not) fully connected. The topology of the computation may be quite different from that of the underlying physical network. This broad description is used as it encompasses the various more-specific descriptions that some researchers use and there is no generally agreed-upon restriction of this definition.

It should be recognized that the distributed system and associated computation are not single entities. This contrasts with the other components in a management tool (with the exception of the monitoring and control code, described below) which are single entities. Thus, the distributed computation is not expected to have a view of itself. Rather, the monitoring entity will collect that global view. Insofar as the computation requires a view of itself, it can be provided by feedback through the control code.

The distributed-system code must be instrumented in two distinct ways. First, monitoring code must be present to capture data of interest. The specific data collected will vary according to the tool's requirements but is fundamentally a subset of the local state of the processes of the computation. This data is forwarded to the monitoring entity. Second, if the computation is to be controlled in any way, the necessary control code must be linked to the system. For example, if the application is debugging, this control code might include the necessary instrumentation to allow the attachment of a sequential debugger.

The monitoring entity has two functions. First, it collects the data that is captured by the monitoring code from the various processes of the computation. Second, it must present that data in a queryable form. It thus acts as a central repository, providing a data structure for storing and querying the mathematical representation of the execution of the distributed computation.

If part of the purpose of the tool is computation visualization, it will have a visualization subsystem that presents the monitored data to a human. While the specific visualization presented will depend on the specific tool requirements, it ultimately is a representation of the distributed-computation execution. The visualization may be manipulable in various ways.

The control entity will likewise be very tool-specific. In general terms, it takes input either directly from the monitoring entity (*i.e.*, it queries the data structure representing the computation) or from a human (or some combination of the two). Its function is to control the computation (*via* the control code that it has access to) based on the observed behaviour of the computation.

By way of example, if we implement a distributed-breakpoint tool, a human will set a breakpoint condition and then commence execution of the computation. The monitoring code will relay the various events in the computation to the monitoring entity. It will build the data structure representing the execution. The control code will analyze the data structure to determine when the breakpoint occurs. It will then cause the computation to be halted. We will discuss specific techniques for distributed breakpoints in Section 3.5.

1.1 Motivation

We wish to build and maintain large distributed systems, where "large" should be understood to mean systems containing thousands of concurrently executing processes. We require that these systems be dependable and, in the event of partial failure, capable of identifying and isolating faults. We need to test and debug these systems. We may wish to steer the execution in certain directions. We probably need precise technical understanding of the runtime computation, especially if we need to do performance tuning. To enhance our understanding we may wish to have some visual presentation of the runtime computation. In summary, we need to be able to observe and control large distributed systems.

Tools for this purpose, as described above, lack the scalability necessary to observe and control large distributed computations. There are three fundamental problems that must be resolved to enable such tools to achieve the desired scalability. First, they require scalable data-gathering. As defined, the monitoring entity is a single repository, which both bounds its scalability and makes it an undesirable single point of failure. Second, any data structures that are used by the monitoring entity must scale with the data collected. The specific data structures used depend on the particular distributed-system model that is adopted by the tool. The model we, and most others in this field, adopt is a partial-order one. Unfortunately, existing data structures for encoding partial orders are either not dynamic or they do not scale with the number of processes. Third, these tools require scalable information-abstraction mechanisms. In particular, visualization is not feasible without appropriate abstraction, or at least will display such a small fraction of the computation as to be meaningless. Likewise, reasoning about the computation is unlikely to scale unless it is over a higher abstraction than raw data.

This dissertation addresses the second of these problems.

1.2 Contributions

This dissertation provides the following significant contributions to scientific and engineering knowledge.

1. The formalization of the operations on a partial-order data structure for distributed-system observation.

2. Proof that timestamp vector-size affects the performance of precedence-test execution time in distributed-system-observation tools.

3. Evidence that practical distributed computations do not in general have large dimension.

4. The creation of a centralized, dynamic timestamp algorithm that scales with partial-order dimension, not width.

5. The creation of a centralized, dynamic timestamp algorithm based on capturing communication locality.

In spite of the existence of several distributed-system-observation tools, the requirements of a partial-order data structure were not well-defined. To the best of our knowledge we are the first to formalize these requirements. The value of this formalization is the ability to build tools based on a partial-order abstract data type, rather than the current *ad hoc* approach based on events as a core data structure. Tools built on a partial-order abstract data type would be able to readily change the implementation of that data type, as needed. Current tools cannot easily change core design choices such as timestamp algorithm.

An abstract data type is, in and of itself, of no value if there do not exist good implementation algorithms for that data type. Current implementation mechanisms for observation tools use Fidge/Mattern vector timestamps (see Section 6.1.2). These timestamps require space per event equal to the number of processes involved in the computation. The space-consumption presents a scalability problem. Specifically, we have demonstrated that as the number of processes increases, the size of these timestamps detrimentally affects the performance of the associated precedence-test algorithm. The cause of this is dependent on the manner in which the tool uses the timestamps. In the case of POET [95, 147], some timestamps are cached, subject to available memory, while others are generated on-the-fly as needed for precedence testing. Clearly the number that can be be cached is dependent on the size of the timestamp. The result of this architecture is that computations requiring just a few hundred processes can spend much of their execution time either generating or copying timestamps. An alternate approach, building the partial-order data structure in memory, does not require timestamp copying, but then becomes dependent on virtual memory as the data-structure size grows beyond the size of main memory. If timestamps were smaller than the number of processes, then more of the data structure could fit in main memory.

We therefore endeavored to create alternate timestamp algorithms that would scale with the number of processes. Our requirement was that growth of the timestamp-vector size was less than linear in the number of processes, while not substantially impairing the precedence-test execution time. We developed two such scalable timestamps.

The first starts with the Ore timestamp (see Section 6.1.5), which is only suitable for static partial-order encoding. In theory these timestamps can require space per event equal to the number of processes, as do Fidge/Mattern timestamps. We therefore initially demonstrated that real-world computations would not generally require Ore timestamps of such a size. Indeed, we showed that partial orders induced by parallel and distributed computations with up to 300 processes can typically be encoded using these timestamps with 10 or fewer entries [164]. Given this possibility proof, we then endeavored to create a dynamic variant. We did so by developing a technique to create pseudo-linear extensions dynamically. This was done by incrementally computing critical pairs of the partial order and then building extensions that reversed those pairs. Indices for the extensions were developed by simulating the infinite divisibility of real numbers [165, 166].

The second technique starts with the Fidge/Mattern timestamp and encodes it in a space-efficient manner. This was achieved by clustering processes. Events that only communicate within the cluster only require timestamps of size equal to the number of processes in the cluster. Events that receive communication from outside the cluster require full Fidge/Mattern timestamps. This approach was then extended to make the clustering hierarchical, which allowed further space-saving and eliminated the requirement for a Fidge/Mattern timestamp [168]. This work was further extended by clustering processes dynamically by communication pattern [169]. By developing good clustering algorithms we were able to achieve more than an order-of-magnitude space reduction while still enabling efficient precedence determination.

In addition to these contributions, various minor contributions have been made. First, we have noted several problems related to existing information-abstraction mechanisms that can be largely attributed to poor formalization. The most serious problem in this regard is that convex abstract events are not well-defined, and that attempts to do so result either in cyclic precedence between abstract events or a creation-order dependency.

Second, we have provided the first correct abstract formalization of synchronous events that views them as single atomic events that occur simultaneously in multiple processes. We have used this formalization to clean up the descriptions of various timestamp algorithms. We have also generalized the concept of synchronous events from pairs of events to arbitrary numbers of events. It is therefore possible to directly model operations such as barriers.

Third, we have formalized the transformation of multicast and multi-receive events into sets of single-partner events. This is significant because it legitimizes the use of an existing, but unproven, practice.

1.3 Organization

This dissertation is organized into three parts, as follows. We first identify the requirements of a distributed-observation tool. To achieve this we define a mathematical abstraction of a distributed computation. We then research current requirements for such tools, and translate those requirements into operations on the mathematical abstraction. The set of such operations forms the specification for our partial-order data structure.

In Part II we describe current solutions for implementing such a partial-order data structure, and identify the limitations of such solutions. The problems identified can be summarized as a lack of scalable, dynamic precedence-test algorithms.

We address these problems in Part III by offering two possible dynamic, centralized timestamp algorithms. These algorithms have various advantages and drawbacks with respect to each other and with respect to Fidge/Mattern timestamps. However, we believe that the space-saving they offer over Fidge/Mattern timestamps is such that they will scale more effectively as the number of processes increases.

PART I

REQUIREMENTS

2 Formal Model

In this chapter we define the formal model of a distributed computation. There are (at least) three possible techniques that might be employed here: Petri nets, a state-based approach, and an event-based approach. While Petri nets have been used to model distributed programs [113], they are not well-suited to our application. State-based approaches, on the other hand, are not so easily dismissed. They are the preferred choice for sequential debuggers, such as dbx or gdb, and are frequently employed for the formal specification and verification of distributed algorithms [97]. The method used is to represent the computation as a triple $< S, A, \Sigma >$ where S is a set of states, A a set of actions, and Σ a set of behaviours of the form

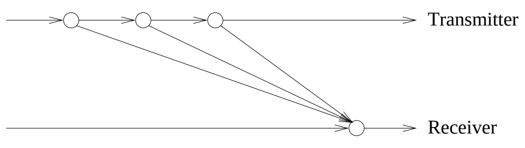
$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_1} s_2 \ldots$$

where $s_i \in S$ and $\alpha_i \in A$. Some form of logic, typically temporal logic, is then used to reason about Σ. The problem with this approach is that, while it may be suitable for reasoning about distributed algorithms, it is not well adapted to observing a distributed computation. There are several aspects that make it poorly suited to our application. Perhaps the most significant limitation is that, while communication and concurrency are probably the most significant aspects of interest in a distributed-observation context, they must be inferred in the state-based approach rather than being central. Furthermore, because of non-determinacy and partial failure, there is no well-defined global state in a distributed system at any given instant. In the case of sequential-system observation there is a single, well-defined state at any given time and actions deterministically move the system from one such well-defined state to another. It is this state that the user or control entity of the observation tool explores. In a distributed computation there is not a single deterministic movement by actions from one state to another. Instead, at any given instant the system could be in any one of a large number of states.

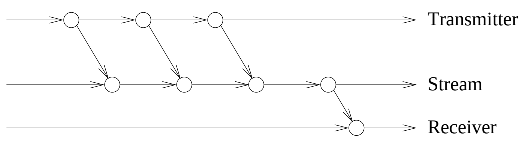
The third technique, and the one we, and most others in this field, adopt, is the event-based approach, originally developed by Lamport [96]. Rather than focusing on the state, it focuses on the *events* (or actions, to use the terminology of the state-based approach) which cause the state transitions. Information about events can be collected efficiently without regard to the current "state" of the system. Causal links between events, and thus between local states, can then be established.

The method taken is to abstract the sequential processes as sequences of four types of events: *transmit*, *receive*, *unary*, and *synchronous*. These events are considered to be atomic. Further, they form the primitive events of the computation.

Transmit and receive events directly correspond to transmit and receive operations in the underlying distributed computation. Every transmit event has zero or more corresponding receive events in different processes. This models attempted transmission, where the message was not received, unicast (or point-to-point) transmission, and multicast transmission. An unsuccessful transmission is effectively equivalent to a unary event in terms of the mathematics we will adopt



(a) Multi-Receive Operation Direct



(b) Multi-Receive Operation with Stream

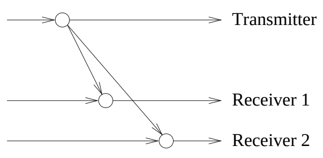
Figure 2.1: Modeling Multi-Receive Operations

(see Section 2.2).

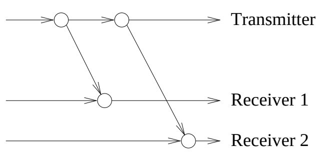
Every receive event has one or more corresponding transmit events. If there is only a single corresponding transmit event then the underlying operation is a simple unicast or multicast transmission. If there are multiple corresponding transmit events then the underlying operation is akin to a transmitter writing several blocks of data to a stream which the receiver reads in one action. Note that this can also be modeled by abstracting the stream as a process. The transmit events would be to the stream process, rather than to the receiver. The receiver would have a single receive event corresponding to a transmit event from the stream process to the receiver. This is illustrated in Figure 2.1.

We note also that a similar transformation may be performed with multicast and broadcast transmissions. Figure 2.2(a) shows the initial multicast. For such a multicast there is an obvious transformation, shown in Figure 2.2(b), which is wrong. The reason it is wrong is that the transmit operation is split into two events, denying its atomicity. Further, the first of the transmit events precedes both of the receive events, while the second transmit only precedes the receive of the second receiver. It is not at all clear that this is a valid transformation. An alternate transformation, shown in Figure 2.2(c), maintains a single transmit event in transmitter process, and introduces an intermediary as did the multi-receive. This transformation maintains the events and precedences of the transmitter and receiver processes.

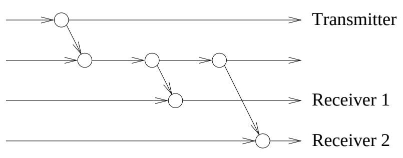
While these transformations are legitimate (we will formally define the transformations and prove their legitimacy in Section 2.3), we prefer to keep our model as general as possible. The rationale for this is that we wish to model the user's view of the system. We will only appeal to these transformations as needed for implementation-efficiency reasons and for consistency with existing observation tools.



(a) Multicast Direct



(b) Point-to-point Multicast



(c) Multicast with Intermediary

Figure 2.2: Modeling Multicast Operations

Continuing with the remaining two types of primitive events, a unary event is any event of interest that does not involve information transfer between processes. Its primary purpose is to allow any additional action or state information to be recorded that the user of the formal model desires.

Synchronous events correspond to synchronous operations in the underlying computation. We model synchronous events as single logical events that occur simultaneously in one or more processes. The reason it is one or more processes and not two or more is to model an attempted synchronous operation that failed. An unsuccessful synchronous event, like an unsuccessful transmit event, is effectively equivalent to a unary event. Synchronous events, like unary events, have no partner events. However, unlike unary events, they do provide information transfer between processes. We have discussed other methods of modeling synchronous events elsewhere [167].

We now briefly review some basic partial-order terminology before presenting the formal mathematical abstraction.

2.1 Partial-Order Terminology

The following partial-order terminology is due to Ore and Trotter [114, 154, 155, 156]. While we will not give detailed partial-order theory here, we will review a few of the important definitions and terminology relevant to this document.

Definition 1 (Partially-Ordered Set) *A partially-ordered set (or poset, or partial order) is a pair (X, P) where X is a finite set and P is a reflexive, anti-symmetric, and transitive binary relation on X.*

First note the restriction to finite sets. This is not strictly a requirement of all partial orders, but it is one that we impose as we are modeling distributed computations. As such, all sets we deal with are finite. This restriction simplifies various theorems and proofs. Second, since the set X is often implicit, it is frequently omitted, and the partial-order is simply referred to by the relation P. This invariably causes problems. We will therefore use the convention of subscripting the relation with the set over which it forms a partial order. Indeed, we will extend this convention to all relations over the set X. Third, note that P_X is reflexive. If P_X is instead irreflexive, then P_X forms a *strict partial-order*. The "happened before" relation, as defined by Lamport [96], is irreflexive and corresponds to the relation $\prec_p$ that we will define in Section 2.2. For simplicity all of the following definitions will assume that we are dealing with strict partial orders. Finally, the terms x_1 precedes x_2 and x_2 succeeds x_1 are used when $(x_1, x_2) \in P_X$.

Definition 2 (Subposet) *A subposet $(Y, P|_Y)$ is a poset whose set Y is a subset of X and whose relation $P|_Y$ is the restriction of P_X to that subset.*

Definition 3 (Chain) *P_X is a chain if-and-only-if*

$$\forall_{x_1,x_2 \in X; x_1 \neq x_2} \, (x_1, x_2) \in P_X \vee (x_2, x_1) \in P_X$$

A chain is any completely ordered poset. It is also referred to as a total order. The *height* of a chain P_X is the number of elements in its set X (that is, the cardinality of X). The *height* of a poset P_X is the height of the tallest chain that is a subposet of P_X. The *length* of a poset is one less than its height. The value of this definition is that it corresponds to the length of the longest path in the directed graph that corresponds to the partial order.

Definition 4 (Antichain) *P_X is an antichain if-and-only-if*

$$\forall_{x_1,x_2 \in X}\ (x_1, x_2) \notin P_X \wedge (x_2, x_1) \notin P_X$$

An antichain is any completely unordered poset. The *width* of an antichain P_X is the number of elements in its set X (that is, the cardinality of X). The *width* of a poset P_X is the width of the widest antichain that is a subposet of P_X. In the context of a distributed computation, the width must be less than or equal to the number of processes.

Definition 5 (Minimal Element) *x_1 is a minimal element of poset P_X if-and-only-if*

$$\forall_{x_2 \in X}\ (x_2, x_1) \notin P_X$$

A minimal element of a poset is any element such that no other element precedes it.

Definition 6 (Maximal Element) *x_1 is a maximal element of poset P_X if-and-only-if*

$$\forall_{x_2 \in X}\ (x_1, x_2) \notin P_X$$

A maximal element of a poset is any element such that no other element succeeds it.

Definition 7 (Extension) *An extension, (X, Q_X), of a partial order (X, P_X) is any partial order that satisfies*

$$\forall_{x_1,x_2 \in X}\ (x_1, x_2) \in P_X \Rightarrow (x_1, x_2) \in Q_X$$

If Q_X is a total order, then the extension is called a *linear extension* or *linearization* of the partial order. If (Y, R_Y) is an extension of the subposet $(Y, P|_Y)$ of (X, P_X), then (Y, R_Y) is said to be a *subextension* of (X, P_X).

Definition 8 (Realizer) *Given a poset (X, P_X) and a set $L = \{(X, L^i_X) \mid 0 \le i < N\}$ of N linear extensions of the poset, L forms a realizer of P_X if-and-only-if*

$$P_X = \bigcap_i L^i_X$$

A realizer of a partial order is any set of linear extensions whose intersection forms the partial order. The *dimension* of a partial order is the cardinality of the smallest possible realizer.

Definition 9 (Critical pair) *(x, y) is a critical pair of the partial order (X, P) if-and-only-if $(x, y) \notin P_X$, $(y, x) \notin P_X$, and $(X, P \cup \{(x, y)\})$ is a partial order.*

A critical pair of a partial order, also known as a non-forced pair, is any pair not in the partial order, whose addition to the partial-order relation would result in a relation that is also a partial order. Equivalently:

$$\begin{array}{rcrl} (x_1, x_2) \in \mathrm{CP}_X & \iff & (x_1, x_2) \notin P_X \wedge (x_2, x_1) \notin P_X & \wedge \\ & & \forall_{x_3 \in X} \left(((x_3, x_1) \in P_X \Rightarrow (x_3, x_2) \in P_X) \right. & \wedge \\ & & \left. ((x_2, x_3) \in P_X \Rightarrow (x_1, x_3) \in P_X) \right) & \end{array} \tag{2.1}$$

where CP_X is the set of all critical pairs of the partial order (X, P_X). The significance of critical pairs, as regards dimension, is in the following theorem [155]:

Theorem 1 (Dimension) *The dimension of a partial order is the cardinality of the smallest possible set of subextensions that reverses all of the critical pairs of the partial order.*

A critical pair (x_1, x_2) is said to be reversed by a set of subextensions if one of the subextensions in the set contains (x_2, x_1). Note that the subextensions need not be linear.

Finally, we say that "x_1 is covered by x_2" or "x_2 covers x_1" if there is no intermediate element in the partial order between x_1 and x_2. Formally:

Definition 10 ($<: \subseteq P_X \times P_X$)

$$x_1 <: x_2 \iff (x_1, x_2) \in P_X \wedge \nexists_{x_3} ((x_1, x_3) \in P_X \wedge (x_3, x_2) \in P_X)$$

In the context of distributed debugging, if $x_1 <: x_2$ we may also say that x_1 is an *immediate predecessor* of x_2 or x_2 is an *immediate successor* of x_1. If multicast and broadcast operations do not exist (*i.e.*, all communication is point-to-point) and synchronous events are limited to synchronous pairs, then any event will have at most two immediate successors or predecessors.

2.2 MATHEMATICAL ABSTRACTION

We now present the formal mathematical abstraction of a distributed computation. First we define the finite sets $\mathcal{T}$, $\mathcal{R}$, $\mathcal{U}$, and $\mathcal{S}$ as being the sets of *transmit*, *receive*, *unary*, and *synchronous* events respectively. These sets are pairwise disjoint. Collectively, they form the set of events for the whole computation:

$$\mathcal{E} = \mathcal{T} \cup \mathcal{R} \cup \mathcal{U} \cup \mathcal{S} \tag{2.2}$$

For convenience of terminology we will use the lower case letters t, r, u, and s (possibly sub- or superscripted) to refer to specific transmit, receive, unary, and synchronous events respectively. We will use e and f (possibly sub- or superscripted) to refer to specific events of unknown type.

Next, we define $\mathcal{P}$ as the set of processes that form the distributed computation. Each sequential process is the set of primitive events that compose it together with the relation that totally orders those events within that process. Thus

$$\forall_{p \in \mathcal{P}}\, p = (E_p, \prec_p) \tag{2.3}$$

where $E_p \subseteq \mathcal{E}$ is the set of events of the process and $\prec_p$ totally orders those events. Note that $(E_p, \prec_p)$ is a chain. We will use p and q (possibly sub- or superscripted) to refer to specific processes. Thus E_q will refer to the event set of process q and $\prec_q$ its ordering relation. When we wish to refer to specific events in process p we will subscript the event identifier with the process identifier. Thus for an event of unknown type in process p we will refer to e_p. Likewise a transmit, receive, unary, and synchronous event in process p would be t_p, r_p, u_p, and s_p respectively.

The relation $\prec_p$ totally orders the events of the process p according to the order of execution in the process. That is, if event e_p^i occurs in the process p before event e_p^j then $e_p^i \prec_p e_p^j$. As such, it is convenient to simply number the events in the process, starting at 1, and then use natural-number comparison as the ordering relation. We will superscript event identifiers with their associated natural number. We will refer to this natural number as the position of the event within the process or, more simply, as its position. Thus, we may define the ordering relation as

$$e_p^i \prec_p e_p^j \iff i < j \tag{2.4}$$

It should also be noted that an event is uniquely specified by its process combined with its position within that process. We will therefore consider <process, position> to be the event identifier. Note that a synchronous event will, in general, have multiple identifiers, as it occurs in multiple processes.

Two further points must be dealt with regarding the relationship between processes and events. First, every primitive event occurs in one or more processes:

$$\forall_{e \in \mathcal{E}}\, \exists_{p \in \mathcal{P}}\, e \in p \tag{2.5}$$

Note that since only synchronous events occur in multiple processes we may also assert that:

$$\forall_{p,q \in \mathcal{P}; p \neq q} \forall_e\, e \in (E_p \cap E_q) \Rightarrow e \in \mathcal{S} \tag{2.6}$$

The flip side of this is that there must be a mapping from every event to the processes in which it occurs. We therefore define two mapping functions, ϕ and φ that map events to processes and to positions within processes respectively.

Definition 11 ($\phi : \mathcal{E} \longrightarrow 2^{\mathcal{P}}$)

$$\phi(e) = \{p \mid e \in E_p\}$$

Definition 12 ($\varphi : \mathcal{E} \longrightarrow 2^{\mathcal{P} \times \mathbb{N}}$)

$$\varphi(e) = \{(p, n) \mid e \in E_p \textit{ at position } n\}$$

Note that for non-synchronous events, these functions will map events to a single process and for synchronous events there will be at most one position per process. Likewise, if the only synchronous events allowed are synchronous pairs, then for synchronous events the functions will map to exactly two processes.

Second, it is sometimes useful to define the reflexive equivalent of the $\prec_p$ relation. It is defined as:

$$e_p^i \preceq_p e_p^j \iff e_p^i \prec_p e_p^j \vee e_p^i = e_p^j \tag{2.7}$$

Thus far we have only related events to individual processes. We now incorporate the effect of communication between processes. To do this, we define the Γ relation.

Definition 13 (Gamma Relation: $\Gamma \subseteq \mathcal{T} \times \mathcal{R}$) $(t, r) \in \Gamma$ *if and only if* r *is a receive event corresponding to the transmit event* t*. Note that* $\forall_r\, r \in \mathcal{R} \Rightarrow \exists_t\, (t, r) \in \Gamma$

Every receive event will have at least one pair in the gamma relation, as it must have at least one corresponding transmit event. The corresponding condition does not hold for transmit events, as there is no guarantee that a transmit event will ever have a corresponding receive event. Further, even if the transmit event does have a receive event, we want to be able to model the incomplete state of the computation where the transmit has occurred but the receive has not. We also define the functions ρ and τ to determine the set of receive events corresponding to a transmit event and the set of transmit events corresponding to a receive event, respectively:

Definition 14 ($\rho : \mathcal{T} \longrightarrow 2^{\mathcal{R}}$)

$$\rho(t) = \{r \in \mathcal{R} \mid (t, r) \in \Gamma\}$$

Definition 15 ($\tau : \mathcal{R} \longrightarrow 2^{\mathcal{T}}$)

$$\tau(r) = \{t \in \mathcal{T} \mid (t, r) \in \Gamma\}$$

As per our previous observation, $|\rho(t)| \geq 0$ while $|\tau(r)| \geq 1$. For convenience in future proofs we will extend the domains of these two functions to all events, recognizing that an event that is not a transmit has no corresponding receive events and an event that is not a receive has no corresponding transmit events. Thus, $\rho(e) = \emptyset$ when $e \notin \mathcal{T}$ and $\tau(e) = \emptyset$ when $e \notin \mathcal{R}$.

We now define two partial-order relations, irreflexive and reflexive precedence, across the set of events for the whole computation.

Definition 16 (Irreflexive Precedence: $\prec_{\mathcal{E}} \subseteq \mathcal{E} \times \mathcal{E}$) *Irreflexive precedence is the transitive closure of the union of* Γ *and* $\prec_p$ *over all processes.*

$$\prec_{\mathcal{E}} = \textit{Transitive Closure}\left(\Gamma \cup \left(\cup_{p \in \mathcal{P}} \prec_p\right)\right)$$

Reflexive precedence is defined analogously:

Definition 17 (Reflexive Precedence: $\preceq_{\mathcal{E}} \subseteq \mathcal{E} \times \mathcal{E}$) *Reflexive precedence is the transitive closure of the union of* Γ *and* $\preceq_p$ *over all processes.*

$$\preceq_{\mathcal{E}} = \textit{Transitive Closure}\left(\Gamma \cup \left(\cup_{p \in \mathcal{P}} \preceq_p\right)\right)$$

Several points should be made about these definitions. First, observe that nothing other than transitive closure is required to capture the effect of synchronous-event communication flow. This may be contrasted with modeling synchronous events as collections of constituent events, each of which is treated as atomic. Under that approach additional rules are required to capture precedence, and to give reasonable meaning to the concept of precedence between constituent events within a synchronous event.

Second, note that there is nothing in the formal model we have defined that requires these relations to be anti-symmetric. If they are not anti-symmetric they cannot be partial-order relations. Consider for example $(t_q^2, r_p^1) \in \Gamma$ and $(t_p^2, r_q^1) \in \Gamma$. By Equation 2.7, $r_p^1 \prec_{\mathcal{E}} t_p^2$. However, $t_p^2 \prec_{\mathcal{E}} r_q^1$, $r_q^1 \prec_{\mathcal{E}} t_q^2$ and $t_q^2 \prec_{\mathcal{E}} r_p^1$. Thus, by transitive closure $t_p^2 \prec_{\mathcal{E}} r_p^1$. What has happened here is that messages must have traveled backwards in time. In fact, any such anti-symmetry would require at least one message to travel backwards in time. While this might be a desirable feature in a distributed system, it does not currently exist. Thus, the relations are anti-symmetric.

Third, observe that the $\prec_{\mathcal{E}}$ relation is the "happened before" relation defined by Lamport [96]. It, together with the base set $\mathcal{E}$, is a *strict partial order* as it is not reflexive. Other terms that are used for either this relation, or the $\preceq_{\mathcal{E}}$ relation, include "precedes" and "causality." The relations themselves more realistically represent potential causality than actual causality. "Precedes" and "happened before" both imply a temporal relation which, while they hold for events within the relation, may also hold for many events not within the relation. That said, we will frequently use any of the above terms in writing where it does not cause confusion. When we wish to be precise, we will use the specific relations $\prec_{\mathcal{E}}$ and $\preceq_{\mathcal{E}}$.

The final aspect of the mathematical abstraction which has not been defined is concurrency. We use the reflexive form of the causality relation to define concurrency as follows.

Definition 18 (Concurrent: $\|_{\mathcal{E}} \subseteq \mathcal{E} \times \mathcal{E}$) *Two events are concurrent if they are not in the causality relation:*

$$e_i^j \|_{\mathcal{E}} e_k^l \iff e_i^j \not\preceq_{\mathcal{E}} e_k^l \wedge e_k^l \not\preceq_{\mathcal{E}} e_i^j$$

Note that we need the reflexive form to avoid the problem of defining an event as concurrent with itself.

2.3 Removing Multicast and Multi-Receive

We now formalize the transformations necessary to remove multicast (including broadcast) and multi-receive events and prove the correctness of these transformations. We do this in three steps. We first specify the meaning of a legitimate transformation. We then formally define the transformations for a multi-receive and a multicast event. Finally, we prove the legitimacy of these transformations.

Definition 19 (Legitimate Transformation) *Given partial orders $(\mathcal{E}_1, \prec_{\mathcal{E}_1})$ and $(\mathcal{E}_2, \prec_{\mathcal{E}_2})$ where $\mathcal{E}_1 \subseteq \mathcal{E}_2$, the transformation function $\mathcal{T} : (\mathcal{E}_1, \prec_{\mathcal{E}_1}) \rightarrow (\mathcal{E}_2, \prec_{\mathcal{E}_2})$ is legitimate if-and-only-if $(\mathcal{E}_1, \prec_{\mathcal{E}_1})$ is a subposet of $(\mathcal{E}_2, \prec_{\mathcal{E}_2})$.*

The justification for calling this transformation legitimate is that it preserves all existing ordering and concurrency relationships and the only new relationships introduced pertain to events in the second partial order not present in the first one. This transformation can equally be expressed as

$$\forall_{e_1,e_2\in\mathcal{E}_1}\ ((e_1,e_2)\in\prec_{\mathcal{E}_1}\Rightarrow(e_1,e_2)\in\prec_{\mathcal{E}_2})\wedge((e_1,e_2)\notin\prec_{\mathcal{E}_1}\Rightarrow(e_1,e_2)\notin\prec_{\mathcal{E}_2}) \tag{2.8}$$

In other words, existing pairs in the partial order remain, and pairs that are not in the $(\mathcal{E}_1,\prec_{\mathcal{E}_1})$ partial order are not introduced in the $(\mathcal{E}_2,\prec_{\mathcal{E}_2})$ partial order.

Define a multicast event e as any event for which $|\rho_1(e)|>1$. Note that $e\in\mathcal{T}$. We now define the transformation of the partial order $(\mathcal{E}_1,\prec_{\mathcal{E}_1})$ containing multicast event t to the partial order $(\mathcal{E}_2,\prec_{\mathcal{E}_2})$ for which $|\rho_2(t)|=1$.

First, define the new process p_t that will contain a sequence of $|\rho_1(t)|+1$ events. The first event, $<p_t,1>$, is a receive event, while the remaining $|\rho_1(t)|$ events are transmit events in $(\mathcal{E}_2,\prec_{\mathcal{E}_2})$. Define γ_2 as a one-to-one mapping from these transmit events to the receive events corresponding to t ($\forall_r\ (t,r)\in\Gamma_1$; there will be $|\rho_1(t)|$ such pairs). Now define Γ_2 as:

$$\Gamma_2=\Gamma_1-\{(t,r)\mid(t,r)\in\Gamma_1\}\cup\{(t,<p_t,1>)\}\cup\left\{\left(t^i,r^i\right)\mid\gamma_2:t^i\to r^i\right\} \tag{2.9}$$

The processes of $(\mathcal{E}_2,\prec_{\mathcal{E}_2})$ are defined as $\mathcal{P}_1\cup p_t$. Likewise the events are $\mathcal{E}_1\cup E_{p_t}$. The partial order $(\mathcal{E}_2,\prec_{\mathcal{E}_2})$ is then defined according to Definition 16 using the Γ_2 relation.

To remove all multicast events this transformation is repeated until such time as no multicast events remain.

We now prove that this is a legitimate transformation. To do so we must show two things. First we show that all pairs in $\prec_{\mathcal{E}_1}$ remain in $\prec_{\mathcal{E}_2}$. Since all of the processes and events in $\mathcal{E}_1$ remain in $\mathcal{E}_2$, and since the only thing removed from the Γ relation was $\forall_r\ (t,r)$, it is sufficient to show that $\forall_r\ (t,r)\in\Gamma_1\Rightarrow t\prec_{\mathcal{E}_2}r$. Since $t\prec_{\mathcal{E}_2}<p_t,1>$ (it is in Γ_2), $\forall_{i>1}\ <p_t,1>\prec_{\mathcal{E}_2}<p_t,i>$ (since p_t is a chain), and $\forall_r\ (t,r)\in\Gamma_1\Rightarrow\exists_{i>1}\ (<p_t,i>,r)\in\gamma_2$ (hence $<p_t,i>\prec_{\mathcal{E}_2}r$), then, by transitive closure, $\forall_r\ (t,r)\in\Gamma_1\Rightarrow t\prec_{\mathcal{E}_2}r$. $\square$

Second, we show that no new pairs are introduced on the event-set $\mathcal{E}_1$. We do so with proof by contradiction, by assuming $\exists_{e_1,e_2\in\mathcal{E}_1}\ (e_1,e_2)\notin\prec_{\mathcal{E}_1}\wedge(e_1,e_2)\in\prec_{\mathcal{E}_2}$. Since $e_1,e_2\in\mathcal{E}_1$, they are not in p_t. Since $e_1,e_2\in\prec_{\mathcal{E}_2}$, they must be in the transitive closure of $\Gamma_2\cup(\cup_{p\in\mathcal{P}_2}\preceq_p)$ which means they must be in the transitive closure of $\Gamma_1-\{(t,r)\mid(t,r)\in\Gamma_1\}\cup\{(t,<p_t,1>)\}\cup\left\{\left(t^i,r^i\right)\mid\gamma_2:t^i\to r^i\right\}\cup\left(\cup_{p\in\mathcal{P}_1\cup\{p_t\}}\ \preceq_p\right)$. However, since $(e_1,e_2)\notin\prec_{\mathcal{E}_1}$, they are neither in the same process, nor in the transitive closure of $\Gamma_1\cup(\cup_{p\in\mathcal{P}_1}\preceq_p)$. Therefore, there must exist some event $e_3\in p_t$ such that $(e_1,e_3)\in\prec_{\mathcal{E}_2}$ and $(e_3,e_2)\in\prec_{\mathcal{E}_2}$. Now, $<p_t,1>$ is the only event in p_t that has an immediate predecessor that is not in p_t, since all other events in p_t are transmit events. Further, $<p_t,1>$ has only one immediate predecessor, *viz.* t. Therefore, $(e_1,t)\in\prec_{\mathcal{E}_2}$, since also $\forall_{e\in p_t}\ <p_t,1>\preceq_p e$ and $(t,<p_t,1>)\in\prec_{\mathcal{E}_2}$. Likewise, $<p_t,1>$ is the only event in p_t that does not have an immediate successor not in p_t. Therefore, $\exists_{e_4\neq<p_t,1>}\ (e_4,e_2)\in\prec_{\mathcal{E}_2}$. Without loss of generality, let e_4 be that event in p_t such that there is no successor to e_4 in p_t that precedes (in $\prec_{\mathcal{E}_2}$) e_2. Thus $\exists_r\ e_4<:r$ such that $(t,r)\in\Gamma_1$, by the definition of γ_2. Therefore $(r,e_2)\in\prec_{\mathcal{E}_2}$.

It is now sufficient to show that $(e_1, t) \in \prec_{\mathcal{E}_1}$ and $(r, e_2) \in \prec_{\mathcal{E}_1}$, since $(t, r) \in \Gamma_1$ and will therefore imply $(e_1, e_2) \in \prec_{\mathcal{E}_1}$ which will provide the contradiction. Suppose $(e_1, t) \notin \prec_{\mathcal{E}_1}$. Since $(e_1, t) \in \prec_{\mathcal{E}_2}$ then there must exist some event $e_5 \in p_t$ such that $(e_1, e_5) \in \prec_{\mathcal{E}_2}$ and $(e_5, t) \in \prec_{\mathcal{E}_2}$. Since $<p_t, 1>$ precedes (in $\prec_{\mathcal{E}_2}$) all events in p_t, then $(<p_t, 1>, t) \in \prec_{\mathcal{E}_2}$. However, t is covered by $<p_t, 1>$, and thus $(t, <p_t, 1>) \in \prec_{\mathcal{E}_2}$, which is a contradiction.

Likewise, suppose $(r, e_2) \notin \prec_{\mathcal{E}_1}$. Again, as above, this implies that $(r, <p_t, 1>) \in \prec_{\mathcal{E}_2}$. This contradicts $(<p_t, 1>, r) \in \prec_{\mathcal{E}_2}$, which is true by transitive closure of Γ_2. $\Box$

The multi-receive transformation is analogous. Define a multi-receive event e as any event for which $|\tau_1(e)| > 1$. Note that $e \in \mathcal{R}$. We now define the transformation of the partial order $(\mathcal{E}_1, \prec_{\mathcal{E}_1})$ containing multi-receive event r to the partial order $(\mathcal{E}_2, \prec_{\mathcal{E}_2})$ for which $|\tau_2(r)| = 1$.

First, define the new process p_r that will contain a sequence of $|\tau_1(r)|+1$ events. The first $|\tau_1(r)|$ events are receive events, while the $<p_r, |\tau_1(r)|+1>$ event is a transmit event in $(\mathcal{E}_2, \prec_{\mathcal{E}_2})$. Define γ_2 as a one-to-one mapping from the transmit events corresponding to r ($\forall_t\ (t, r) \in \Gamma_1$; there will be $|\tau_1(r)|$ such pairs) to the receive events in p_r. Now define Γ_2 as:

$$\Gamma_2 = \Gamma_1 - (\forall_t (t, r) \in \Gamma_1) \cup (<p_r, |\tau_1(r)|+1>, r) \cup \gamma_2 \tag{2.10}$$

The processes of $(\mathcal{E}_2, \prec_{\mathcal{E}_2})$ are defined as $\mathcal{P}_1 \cup p_r$. Likewise the events are $\mathcal{E}_1 \cup E_{p_r}$. The partial order $(\mathcal{E}_2, \prec_{\mathcal{E}_2})$ is then defined according to Definition 16 using the Γ_2 relation.

To remove all multi-receive events this transformation is repeated until such time as no multi-receive events remain. The proof that this is a legitimate transformation is analogous to that of the previous proof, and so we omit it.

We note in passing that these are formal transformations for the purposes of the above proofs. Using them in practice unaltered is probably a bad idea as they result in one additional process for each multicast and multi-receive. In many instances this would not be necessary. For example, a sequence of totally-ordered multicasts from the same sender to the same group of receivers could make use of a single additional intermediary.

We also observe that similar transformations could be developed to convert synchronous events occurring in more than two processes into collections of synchronous pairs, or even to remove synchronous events altogether. Conversely, we could remove asynchronous events and replace them with only synchronous events (note that the previous transformations are legitimate if the new process had synchronous pairs rather than transmit and receive events). As we have noted earlier, we prefer to maintain a general framework that is capable of modeling the computation as it occurred, rather than as might be preferable for our convenience.

2.4 Other Issues

There is little in our formal description regarding partial failure, performance issues or varying numbers of processes, although all three of these were observed to be features of a distributed system. The performance issue will be the subject of specific tools, rather than of the model itself. The monitoring code may be designed to determine performance-related problems. The control portion can then be designed to steer the computation to improve performance. For example, this is the approach used in the Falcon system [37, 58, 81].

Correct handling of partial failure must be designed into the distributed system. This includes the correct handling of failure within the monitoring and control system, and specifically the monitoring and control code that is integrated with the distributed computation. For example, a Byzantine failure in the computation could easily spread to the monitoring entity if the information that is claimed to be observed by the monitoring code is not correct. We do not attempt to deal with this issue in this work. We do presume that the information presented by the monitoring code is correct and represents, in some measure, the behaviour of the computation. We presume it is the responsibility of a user to determine if the observation is in error, and to act to correct it accordingly. This is an area for future investigation. The only specific failure that the model represents is that it does not require transmit and synchronous events to complete successfully.

The problem of a variable number of processes is evaded at present by modeling any process that will ever exist during the course of a distributed computation as being implicitly present, though without events, from the beginning. A process that leaves the computation simply ceases to have events. While there is nothing in this approach that contradicts the formal model, and it is a very simple method of dealing with a variable number of processes, it is not particularly satisfying. The computation may be unbounded (as in the case of a distributed operating system) or may have very many short-lived processes. As we shall see in Section 7.1, the effect of this approach can be very expensive. The only solution to this we are currently aware of is that due to Richard [124]. However, it presumes that the timestamps are used directly in the distributed computations, rather than in the data structure of the observation system, and so is not directly relevant to our problem.

Finally, we wish to observe that in the course of our abstraction we have, in fact, managed to model a broader range of systems than simple distributed ones. We therefore take this brief digression to discuss how broad our model is and what systems it can, in principle, cover.

We observe that it can trivially cover message-passing parallel computing, as this is largely indistinguishable from distributed computing. The primary difference is in application and in the speed of message passing. These issues are not relevant to the formal model we have defined. In addition, it can be applied to concurrent programs, such as thread-based applications or multi-tasking operating systems. Indeed, any system or environment that can be modeled as a collection of co-operating sequential entities is covered by our model.

For the remainder of this dissertation, for the most part, we deal with the partial order as a partial order. As such, although we have used the term “process” or “sequential process” to this point, we will generalize the term to “trace,” where a trace is simply any sequential entity. It may be a semaphore, a monitor, a shared variable, a thread, or possibly even a sequential process, as per our original motivation. The key property of a trace is that it is a totally ordered set of events that have been collected from a parallel, concurrent or distributed system. We choose not to use the term “chain” because the data structures we build will be dependent on the nature of events in the systems we are monitoring, rather than being an abstract mathematical construct. Various properties of these systems will be crucial in our building of efficient data structures.

3 Data-Structure Requirements

To determine the requirements on the partial-order data structure, we must know what queries are performed by various distributed-system observation and control tools. To identify these queries, we must determine the general operations performed by these tools and translate those operations into queries on the partial order. Having looked at debugging, monitoring, and steering tools, we have identified the following as being general operations that are performed by these tools.

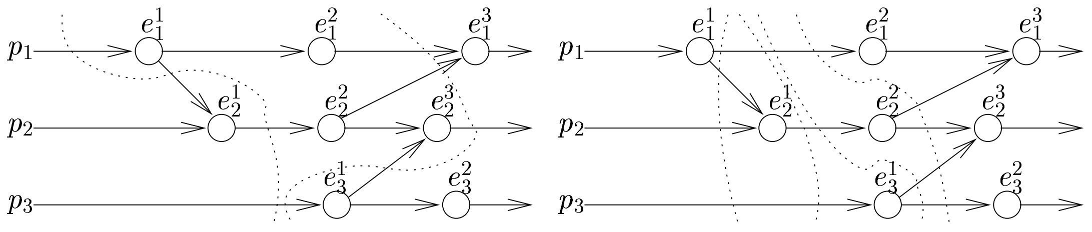
1. Event inspection
2. Seeking patterns
3. Race detection
4. Computation replay
5. Distributed breakpoints
6. Determining execution differences
7. Performance analysis
8. Visualization
9. Abstraction

We now describe these operations and identify the queries they require of a partial-order data structure.

3.1 Event Inspection

Event inspection is informing the user of "relevant information" associated with an event. This relevant information can vary widely. It may be type information such as whether the event is a transmit, receive, synchronous, or some other type. These types will be target-environment specific, not the types we have defined in our formal model. Other information may include partner-event identification, the time at which the event occurred, or various text information that the developer may find useful. As long as the total quantity of data is not substantial this may be collected and provided to the user fairly easily. The primary issue is finding the event of interest within the data structure. This can usually be performed by simple indexing techniques, as we can take advantage of the sequential nature of traces and just use the event identifier. It may be marginally more complex in the target-system-independent environment, where the meaning of what an event is must be associated with the target environment in some manner. POET solves this by the use of the target-description file.

It may also be desirable to provide some subset of the sequential state of a process. This can be particularly useful in reasoning about consistent global states. However, if the quantity of state information required is at all substantial, the cost of collecting it may be prohibitive. It may also



(a) Inconsistent Cuts (b) Consistent Cuts

Figure 3.1: Global State

require some more-complex storage and indexing techniques than if the quantity of information is small. Uninterpreted state information is relatively easy to index, since it can be treated in a similar manner to large objects in database engines [50]. More sophisticated indexing is required for interpreted state information, and is probably target-system dependent. We do not address interpreted state information further in this dissertation.

The query requirements imposed by event inspection are then the ability to return basic and extended event information given an event identifier.

3.2 Seeking Patterns

Currently there are two key types of patterns that may be sought within a debugging, monitoring, or steering context. First we may seek patterns within the structure of the partial order. For example, we may wish to look for the pattern:

$$e^i_{p_1} \prec_{\mathcal{E}} e^j_{p_2} \wedge e^k_{p_3} \prec_{\mathcal{E}} e^j_{p_2} \wedge e^i_{p_1} \parallel_{\mathcal{E}} e^k_{p_3} \; ; \; p_1 \neq p_2 \neq p_3 \tag{3.1}$$

This particular pattern is a crude form of race detection. We are seeking events in traces i and k that both precede an event in a third trace j but that have no synchronization between them. The events thus form a potential race condition. We will discuss race detection in more detail below.

This form of structural pattern searching is equivalent to directed-subgraph isomorphism. Specifically, it is equivalent to asking if the directed acyclic graph that represents the partial order of the computation contains a subgraph isomorphic to the directed graph that represents the pattern being sought. The directed graphs in this equivalence can be either the transitive reductions or the transitive closures of the respective partial orders. This problem is known to be NP-complete, even if the pattern sought is a directed tree [51].

The second type of pattern that we may seek is a pattern within a consistent global state. This is frequently referred to as global predicate detection [18]. We will first explain what is meant by a consistent global state. Consider the execution shown in Figure 3.1. It is not possible for trace 1 to be in the state prior to the event e_1^1 at the same time as trace 2 is in the state between events e_2^1 and e_2^2, as shown in the first cut in Figure 3.1(a). On the other hand, it is possible for trace 1 to be in the state between events e_1^1 and e_1^2 while trace 2 is in the state between events e_2^1 and e_2^2, as seen in the second and third cuts in Figure 3.1(b). To explain these points further we must first provide some clear definitions of our terms. The following definition is a modification of the one due to Basten *et al* [6], adjusted to include synchronous events.

Definition 20 (Consistent Cut: $\mathcal{C} \subseteq \mathcal{E}$) *A subset of the set of events $\mathcal{E}$ forms a consistent cut $\mathcal{C}$ if and only if*

$$\forall_{e_1,e_2\in\mathcal{E}}\ (e_1 \prec_{\mathcal{E}} e_2 \wedge e_2 \in \mathcal{C}) \Longrightarrow e_1 \in \mathcal{C} \tag{3.2}$$

A consistent cut is a set of events such that no event is present in the set unless all of its predecessors are present. It thus represents a possible state of the computation. Since we model synchronous events as single logical events, a synchronous event is either in the cut or it is not. Modeling a synchronous event as more than a single event would require modifications to the definition to ensure that either all constituent events of the synchronous event are in the consistent cut, or none are.

It is called a cut because it cuts the set of events in two, as shown in Figure 3.1. Those events to the left of the cut line are part of the cut; those to the right are not. It is called a consistent cut because the cut creates a consistent global state. Figure 3.1(b) shows various consistent cuts.

The set of consistent cuts ordered by $\subseteq$ is a complete lattice [6]. The lattice for the computation of Figure 3.1 is shown in Figure 3.2. The computation can then be viewed as proceeding on *some* path from the bottom element of the lattice to the top element. Note that this is not strictly true, as it is possible for more than one event to occur simultaneously. However, any such instance would amount to a race condition. If it is a significant race condition, it is a defect in the program. If not, it will not be relevant which path is chosen. We will address this point further below.

Given this view of the computation, and any ascending path as a possible execution sequence, we may then consider the computation to have a possible sequence of global states that is the sequence of states along that path. In other words, a potential global state of the computation is any edge within the lattice. Such a potential global state is referred to as a consistent global state, as it is consistent with a possible execution order of the computation. An alternate way of looking at a consistent global state is to consider it to be a sequence of edges in the process-time diagram which divide the computation to form a consistent cut.

Having acquired a notion of a consistent global state, we can seek patterns within that state. Such patterns are referred to as predicates. There are several varieties that may be sought, such as stable predicates (once the predicate is true, it remains true), definite predicates (the predicate is true on all possible paths in the lattice), possible predicates (the predicate is true on some paths in the lattice), and so forth. From the perspective of a partial-order data structure, the primary concern is the ability to determine what is, or is not, a consistent global state. This is turn means we need the ability to determine consistent cuts.

Given these two pattern-seeking operations, we can identify various requirements for our partial-order data structure. Clearly, from the first type of pattern, we need to be able to seek out precedence patterns in the structure. Thus, we require efficient determination of the $\not\prec_{\mathcal{E}}$, $\preceq_{\mathcal{E}}$ and

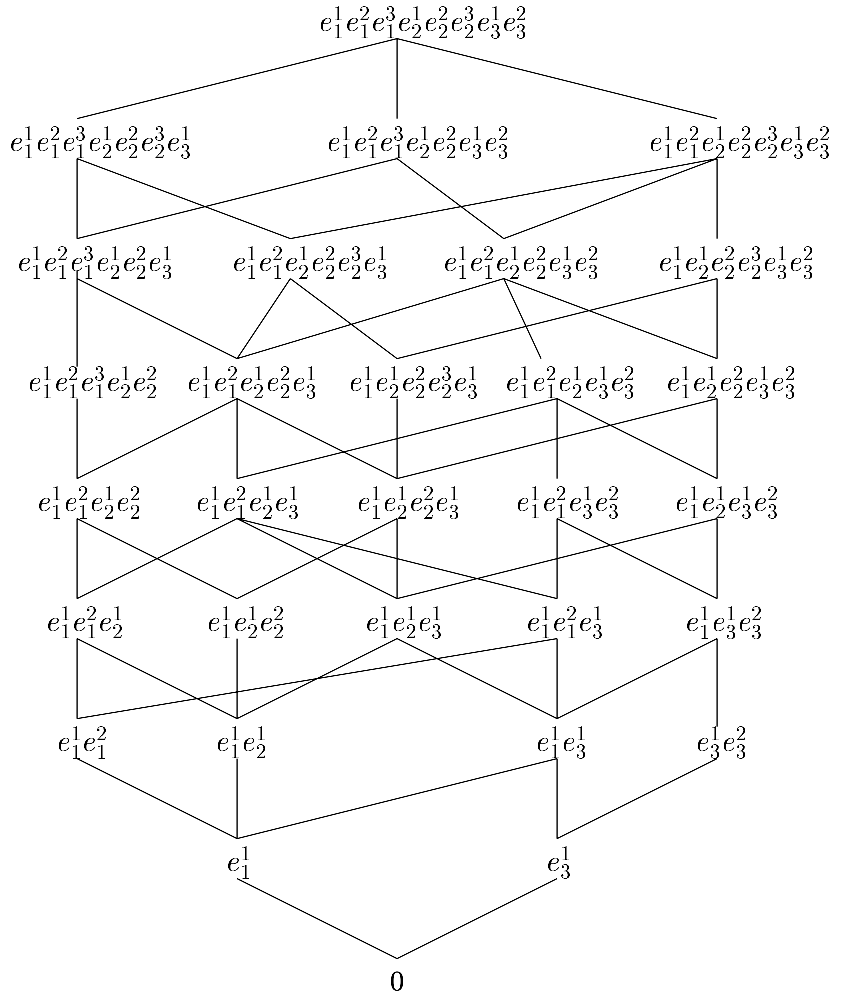

$e_1^1e_1^2e_1^3e_2^1e_2^2e_2^3e_3^1e_3^2$
$e_1^1e_1^2e_1^3e_2^1e_2^2e_2^3e_3^1$
$e_1^1e_1^2e_1^3e_2^1e_2^2e_3^1e_3^2$
$e_1^1e_1^2e_2^1e_2^2e_2^3e_3^1e_3^2$
$e_1^1e_1^2e_1^3e_2^1e_2^2e_3^1$
$e_1^1e_1^2e_2^1e_2^2e_2^3e_3^1$
$e_1^1e_1^2e_2^1e_2^2e_3^1e_3^2$
$e_1^1e_2^1e_2^2e_2^3e_3^1e_3^2$
$e_1^1e_1^2e_1^3e_2^1e_2^2$
$e_1^1e_1^2e_2^1e_2^2e_3^1$
$e_1^1e_2^1e_2^2e_2^3e_3^1$
$e_1^1e_1^2e_2^1e_3^1e_3^2$
$e_1^1e_2^1e_2^2e_3^1e_3^2$
$e_1^1e_1^2e_2^1e_2^2$
$e_1^1e_1^2e_2^1e_3^1$
$e_1^1e_2^1e_2^2e_3^1$
$e_1^1e_1^2e_3^1e_3^2$
$e_1^1e_2^1e_3^1e_3^2$
$e_1^1e_1^2e_2^1$
$e_1^1e_2^1e_2^2$
$e_1^1e_2^1e_3^1$
$e_1^1e_1^2e_3^1$
$e_1^1e_3^1e_3^2$
$e_1^1e_1^2$
$e_1^1e_2^1$
$e_1^1e_3^1$
$e_3^1e_3^2$
e_1^1
e_3^1
0

Figure 3.2: Global State Lattice

$\|_{\mathcal{E}}$ relations. Note also the requirement in the sample pattern of Equation 3.1 to distinguish the traces. It is therefore sometimes useful in pattern searching to be able to determine predecessors and successors by trace of a given event, and likewise to identify all different-trace predecessors and successors of a given event.

From the second pattern type, we can also identify a need to determine the set of greatest predecessors to an event within each trace. This set of greatest predecessors of an event represents the minimal events in their respective traces that must be present to form a consistent cut with that event.

3.3 Race Detection

We have already seen that race detection is, in some sense, a special case of structural pattern seeking. Specifically, Equation 3.1 gave a crude pattern for determining possible races. There are several problems with that particular pattern. First, because it uses precedence, rather than the more specific *covers* relation, it will catch many cases that are simply the transitive closure of the specific race events. Thus, we start by narrowing the pattern to

$$e^i_{p_1} <: e^j_{p_2} \wedge e^l_{p_3} <: e^k_{p_2} \wedge e^j_{p_2} <: e^k_{p_2} \wedge e^i_{p_1} \|_{\mathcal{E}}\ e^l_{p_3}\ ;\ p_1 \neq p_2 \neq p_3 \tag{3.3}$$

This is now probably too restrictive, as it does not allow for any intervening events in trace p_2 between $e^j_{p_2}$ and $e^k_{p_2}$. However, if we replace the covers test with a precedence test, then we may open up the pattern too much, and catch a number of irrelevant cases. Given receive event $e^k_{p_2}$ with corresponding transmit event $e^l_{p_3}$, we need a pattern that catches the first preceding receive from trace p_1 and determines if its corresponding transmit event is concurrent with $e^l_{p_3}$. Thus, in a more general sense, we require a pattern language for specifying desired patterns, complete with variables. Jaekl [68] and Fox [46] have performed some work in this direction, though it is arguably at the level of grep-like matching, though on partial orders rather than strings. What we envision is the next step to a Perl-like language, with all of the constructs of a typical language, such as loops, variables, if-statements, *etc.*, present, but also all of the partial-order pattern matching built in. While such a language is beyond the scope of this dissertation, we enable it by providing the necessary queries for pattern matching on the partial-order data structure. The additional queries implied by this include partner, covers, covered-by, greatest predecessor by type, and least successor by type, where these all have variants of same-trace, different-trace and all-traces.

Unfortunately, for race detection there is a much deeper issue, which is that, even with a perfect specification of a race condition, we have only identified a race condition. We have not concluded that it is significant. In other words, non-determinism exists and is not necessarily incorrect. First, non-determinism at the message-transfer level does not imply non-determinism at the programming-language level. For example, in PVM [53] a receive operation can specify tags on what it will receive. Thus, two successive receive operations may force a message ordering by tags, where there appears to be a race condition when looking only at the corresponding transmit events. Second, some programming languages are built around the concept of non-determinism (*e.g.*, concurrent logic languages [134] which are built around Dijkstra's guarded command [34]). Finding a race condition in a program written in such a language is not significant in and of itself. It is necessary to determine if the race condition is significant.

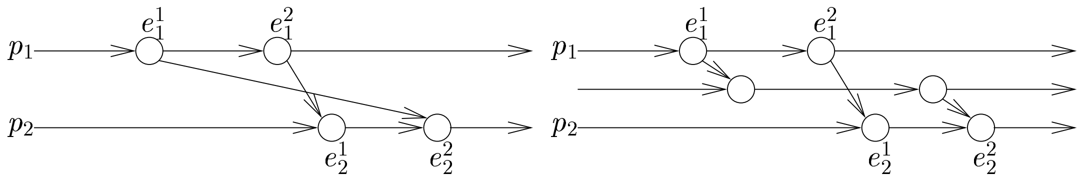
Damodaran-Kamal and Francioni [29] refer to programs with race conditions as nonstrict. They are then classified as either determinate or nondeterminate. A nonstrict determinate program is one whose output remains the same under all executions. That is to say, the outcome of race conditions does not affect the program outcome. More precisely, it should not. If it does, the program has a defect. In other words, a race condition is significant in a nonstrict determinate program if the reordering of the outcome of the race causes a different output by the program. Determining if the various races are significant in this instance is, to some degree at least, possible. After recording the execution of the program, the program is replayed, but it is constrained to follow the opposite outcome for each race condition in turn. If the output of the program remains the same, then the race conditions were not significant. This is, of course, both expensive (though automatable in principle) and requires replay capability. We will discuss replay further in the following section.

The second class of nonstrict programs are those that are nondeterminate. These programs produce a different output under different race outcomes. Damodaran-Kamal and Francioni [29] claim that this is undesirable and do not deal with it further. The problem with this view is that there are legitimate programs which create outputs that differ under differing race outcomes. The canonical example is the case of the joint bank account. The account initially contains $10. Simultaneously one party to the account deposits $100 while the other attempts to withdraw $50. Under one execution the account has $110 at the end. In another, it has only $60. Both are valid outcomes. Any distributed operating system will have behaviour that is similar to this. There is no current solution to this problem that we are aware of.

There are, therefore, several issues that must be addressed by our data structure to deal with race detection. First, it must support efficient structural pattern detection as per our discussion above. Further, the event inspection must include enough detailed information to support race-detection patterns. Thus, clearly the event identifier must be accessible, but also there must be a tie-in to the code base in the event information. It is not clear if this information can be made target-system independent, automatically generated, provide sufficient information for race-detection purposes and be efficient. For the purpose of this dissertation, we assume that any necessary tie-in with the code base can be made by data contained in either the basic or extended event information.

3.4 Computation Replay

Computation replay is the constrained re-execution of a distributed computation such that it follows a specific partial order. Note that the partial order that the re-execution is constrained to follow need not be the same as the partial-order induced by the original execution. Also note that the constraint may be limited to a point in the re-execution, after which the computation may no longer be so constrained. There are several reasons for doing computation replay.



(a) Causal Violation (b) Replay Technique

Figure 3.3: Computation Requiring Message Control

First, the act of debugging a computation interferes with the execution of that computation. This is known as the probe effect [49]. To minimize this effect, we must minimize the impact of debugging. One way of achieving this is to execute the program while collecting the minimum amount of information necessary to determine the partial order of execution. This causes the least perturbation. In order to closely examine the program execution, we re-execute it, constraining it to follow the partial order of the original execution. This is the technique used by Yong and Taylor [180].

While computation perturbation can be minimized using methods such as the above monitoring process, it cannot be eliminated. However, because of the event model that we are using, we can capture the effects of these perturbations [104]. Specifically, any perturbation of the computation caused by monitoring is equivalent to the delaying of an event in one trace relative to some event in another trace (possibly caused by some unintended synchronization resulting from the monitoring). This is nothing more than a race condition. The objective in this case is then to execute the program, capturing the partial order of execution. The program is then re-executed, constrained to follow the original partial order up to the relevant race condition, and then the outcome of the race condition may be altered to the opposite of that which occurred in the original execution. This may be performed automatically, by automating the search for race conditions and then invoking the appropriate re-execution. Kranzlmüller and Volkert [83, 88] have developed solutions for PVM [53] and MPI [109] using this approach, constraining the number of race conditions considered by source-code analysis. Alternately, re-execution may be performed interactively, with the developer selecting which race conditions to test. This is the approach used by Kranzmüller *et al.* in their EMU [56], ATEMPT [84] and PARASIT [85] tool suite.

A third reason for computation replay is the need for efficient distributed breakpoints. We will describe that in the following section.

To constrain an execution to follow a partial order in general requires control of the message facility, since communication is not usually causally ordered. For example, in Figure 3.3(a), while the transmit event e_1^1 must be allowed to proceed, the message from e_1^1 to e_2^2 must delayed by the replay facility to ensure that the message transmitted in event e_1^2 is received first. While FIFO channels would solve the specific problem shown, the message sent from e_1^2 could just as easily be to a third trace, which then transmits a message to be received by e_2^1. Again, the message from e_1^1 must be transmitted, but delayed.

For systems in which communication is causally ordered, either by the design of the communication subsystem (see Isis, Horus, Totem, Newtop, Transis, *etc.* [2, 3, 12, 13, 17, 41, 59, 101, 122, 130]) or because all communication is synchronous (see, for example, DCE [44, 179, 180]), the replay facility does not need to control the communication subsystem. It is sufficient to force an ordering on transmit and receive events (or synchronous calls and receives, in the case of synchronous communication). The nature of the ordering is as follows. For any two receive events, r_1 and r_2 that are causally ordered such that $r_1 \prec_{\mathcal{E}} r_2$, we force r_1 to occur before t_2, where t_2 is the transmit event corresponding to r_2. Note that this will never violate the partial order due to the causal-message-ordering property.

For systems in which causal message ordering is violated, the basic replay algorithm must identify all such violations. Specifically, if $t_1 \prec_{\mathcal{E}} t_2$ but $r_2 \prec_{\mathcal{E}} r_1$, then the message from t_1 must be delayed. The delay can be introduced by using a delay trace, as seen in Figure 3.3(b). The previous algorithm then applies. The delay-trace transmit event is delayed until the $e_{\frac{1}{2}}$ receive event has occurred.

While we have only specified the constraints required for replay algorithms, and specific algorithms will vary in their details, it is clear that the primary data structure requirement for these replay algorithms is efficient event-precedence testing.

An alternate technique is to checkpoint the computation at various intervals, and use the checkpoints to avoid much of the replay. This is only applicable if the intent of the replay is to reach a specific point in the computation and then proceed with some action. The reasons we have cited all fall into this category, though there may be other reasons for which this technique would not be applicable.

3.5 DISTRIBUTED BREAKPOINTS

A distributed breakpoint is intended to perform the same function for a distributed computation that a normal breakpoint does for a sequential computation. That is, it is intended to allow a user of a distributed-debugging tool to stop the computation on some specified trigger (typically reaching a line of code or a change in the value of a variable) and examine the program state. This is clearly a hard problem for a distributed debugger as there is no well-defined global state, but rather a set of possible global states. We will first describe the relevant issues involved in distributed breakpoints and then indicate what support would be needed from our data structure.

First, a trigger event may be as simple as a change in value or reaching a piece of code in a sequential process that is part of the whole computation. However, there is no reason why it should be so specific. The trigger event may be distributed over the computation, in which case it would amount to a pattern that needed to be discovered, per our previous discussion. We will name traces where trigger events occur as trigger-event traces. Likewise, those that do not contain trigger events will be called non-trigger-event traces.

Given that a trigger event has occurred in the computation, we then need to decide exactly what needs to be stopped and where. In a sequential debugger, the entire computation is stopped. In the distributed case it is not quite as obvious. Clearly the trigger-event traces must be halted, and this is a simple matter of applying the relevant sequential-halting solution. However, for non-trigger-event traces, the solution is not quite as clear. One solution would be to allow such traces to continue, on the presumption that they will halt as they become causally dependent on the halted traces. At that point they would block, in a causally consistent state, and sequential debuggers could be applied to those non-trigger event traces as needed.

There are at least two problems with this approach. First, there is no guarantee that non-trigger-event traces will ever synchronize, and thus halt. For example, they may be slave processes that perform a computation, asynchronously return the result, and then terminate. Thus, after start up, they may no longer be causally dependent on a trigger-event trace. This problem may be solved by forcing non-trigger-event traces to halt as soon as we have detected the trigger event. By definition the computation will halt in a consistent global state, since it would not be possible for a receive to occur before the corresponding transmit. However, there may be outstanding messages (ones that have been sent but not yet received), and care must be taken not to lose these. This solution is the method employed by Miller and Choi [107].

The second problem is that relevant debugging information may be lost. Specifically, the reason we desire non-trigger event traces to be halted is that the trigger event may be causally dependent on events that occurred in non-trigger event traces. However, by allowing those traces to continue, the relevant causes of the trigger event may be discarded. Thus Fowler and Zwaenepoel [45] proposed the idea of a *causal distributed breakpoint*. In this technique every trace must be halted at the greatest predecessor event of the trigger event. This ensures that further processing in non-trigger-event traces does not obscure the state that led to the causal message transmission. This technique is not guaranteed to prevent the loss of relevant debugging information, since the happened-before relation reflects potential causality, not actual causality. However, for well-written programs potential causality and actual causality should coincide.

The Miller and Choi approach, insofar as it is useful, makes no requirement on our data structure. The Fowler and Zwaenepoel technique, however, does in two ways. First, we must be able to identify the greatest-predecessor set of events. Second, it cannot be implemented without using replay. We note in passing that it is not clear that any distributed-breakpoint algorithm can be implemented, or implemented efficiently, without replay. A simple approach would require that every transmit event be followed by a local temporary halt until it could be determined whether or not that event was a greatest-predecessor event to the breakpoint-trigger event. However, because of transitivity, it is not clear quite how this might be determined in an online manner. This is, to our knowledge, an open problem. Given this problem, the alternative is to perform a computation replay up to the greatest-predecessor cut.

3.6 CHECKPOINTING

Replay may be time consuming. It may therefore be desirable to checkpoint the individual traces at various times during the course of the computation. Replay to a given breakpoint is then accomplished by first starting at a set of checkpoints and proceeding until the breakpoint is reached. This is more complex than replay from the start. The reason is that the set of checkpoints must either form a consistent cut of the partial order in which no messages are outstanding (*i.e.*, the cut, C, must satisfy $\forall_t \, t \in C \Rightarrow \rho(t) \in C$ in addition to equation 3.2.), or the relevant message information must also be logged.

There are two principle approaches to implementing the no-message-logging technique. The first method is to ensure that when checkpoints are made, there are no outstanding messages. A replay operation then requires the identification of the latest such checkpoint that is prior in all traces to the causal distributed breakpoint. While the creation of such checkpoints is algorithmically possible, to do so would substantially affect the performance of the system since it effectively requires a global synchronization operation. The second method is to checkpoint individual traces as seems appropriate. When a replay operation is required, a set of checkpoints is sought out that form a consistent cut with no outstanding messages and prior to the causal distributed breakpoint. This second method can rely on our data structure to determine the required cut. Note that this method has no guarantee that the required cut will be found.

If message logging is possible then it is not necessary to find such a restricted cut. It may not even be necessary to find a consistent cut. The Fowler and Zwaenepoel technique [45] uses a coordinator to initiate checkpoints in a two-phase protocol. First, all processes are requested to checkpoint. Each process informs the coordinator what the last local event identifier is that it checkpointed. Having received responses from all processes, the coordinator issues a checkpoint confirmation to all processes, indicating the maximum event checkpointed by each process. After a checkpoint is initiated, each receiver logs all messages received until the checkpoint confirmation. After that, receivers log all messages which have a transmit event identifier that is less than the identifier indicated in the checkpoint confirmation. All dependency information is also recorded throughout. This is then sufficient information to restore the computation and commence a replay, even though the restoration may not be to a consistent global state. Using this technique, the requirement of our data structure for checkpointing is simply to be able to identify the checkpoint prior to the causal distributed breakpoint.

3.7 EXECUTION DIFFERENCE

Debugging systems are frequently used in a compile, test, debug cycle. A primary question that a user may wish to know the answer to after making a code fix is whether or not the fix made any difference, and if it did, what difference it made. The same cycle and questions occur in debugging distributed computations. However, when presented with two displays of non-trivial-size partial orders, a user cannot easily determine where those orders differ. Han [60] therefore designed a system to determine those differences (more precisely, the point at which the differences start, by trace). This is similar to structural pattern recognition, though with a couple of important differences. First, the “pattern” being sought is of the same size as the partial order it is being sought within. Second, where we will reject patterns that do not match (*i.e.*, the pattern matching case is strict subgraph isomorphism), we explicitly wish to know the differences between the two orders. That is, our primary concern is not an answer to the graph-isomorphism question (*i.e.*, are these two partial orders identical?). This is one reason why graph-isomorphism approaches to this problem are insufficient. A second, and stronger, reason is that execution-difference algorithms can take advantage of factors in the partial order of computation that are not present in general graph isomorphism. For example, the partial order has totally-ordered traces within it that are not artificially generated. Rather, they correspond to sequential computation objects. Once it is possible to match traces between partial orders, then event differences are simply difference within those traces.

The data-structure support necessary for this is primarily event-identification information and matching-partner information. In addition, trace-identification information might be useful. This trace-identification information may be of limited value, since it is not guaranteed (in an arbitrary target environment) that trace identification or order of appearance will remain the same between executions.

It should also be noted that Han's algorithm is offline. It might be useful to have an online version of this. Two possible online approaches are possible. First, if we have executed a computation, and recorded its partial order of execution, then it may be desirable to monitor its re-execution and invoke the debugger in the event that differences are detected. Second, if may be useful to compare two simultaneously executing computations. This work is beyond the scope of this dissertation.

Finally, it would be useful if more-sophisticated differences could be observed. The Han algorithm is limited to indicating the point at which traces diverge. It may be that there is a portion of difference, after which the computations continue with an identical order again. If this were the case, identifying this difference in whole, rather than the starting point only, would be quite useful. Again, this is beyond the scope of this dissertation.

3.8 Performance Analysis

Performance analysis, and enhancement, is often a key requirement for observation and control. For example, in Vetter's definition [160], program steering is either for performance enhancement or application exploration. Enhancement mechanisms tend to be application-specific, even when the steering mechanisms are of a more general nature, such as load balancing. The analysis that leads to the need for a specific mechanism, though, can be general. We will therefore omit further discussion of enhancement in favour of analysis.

There are various aspects of the partial-order model that make it attractive for performance analysis. Specifically, the minimum execution time can be bounded by determining the longest chain within the partial order (see, for example, the critical-path diagram of ParaGraph [62]). In doing such a check, the partial-order data structure would have to maintain real-time information for each event, and the longest chain would have to be determined with respect to this information. Such a minimum execution time will only be achievable if there are a sufficient number of resources present during the execution. This number is not simply the longest antichain, as traces do not have a one-to-one correspondence with processes. This is apparent in Figures 2.1 and 2.2. As such, it is necessary to determine the longest antichain that corresponds to processes. This is likely target-specific. Also of value is the variation in parallelism over the course of the computation. This is akin to some notion of what the longest antichain is at a given point in time. This may help in the dynamic scheduling of processors between multiple distributed computations so as to maximize throughput for the whole. In a similar vein would be determining the minimum execution time in the presence of a fixed number of processors that is smaller than the maximum degree of parallelism. This is equivalent to determining the maximum chain in the event of creating an extension to the partial order such that the longest antichain is no more than the number of processors. The common theme in all of these ideas is the requirement on the data structure for determining antichains and longest chains, and for extending the partial order so as to reduce the length of the longest antichain.

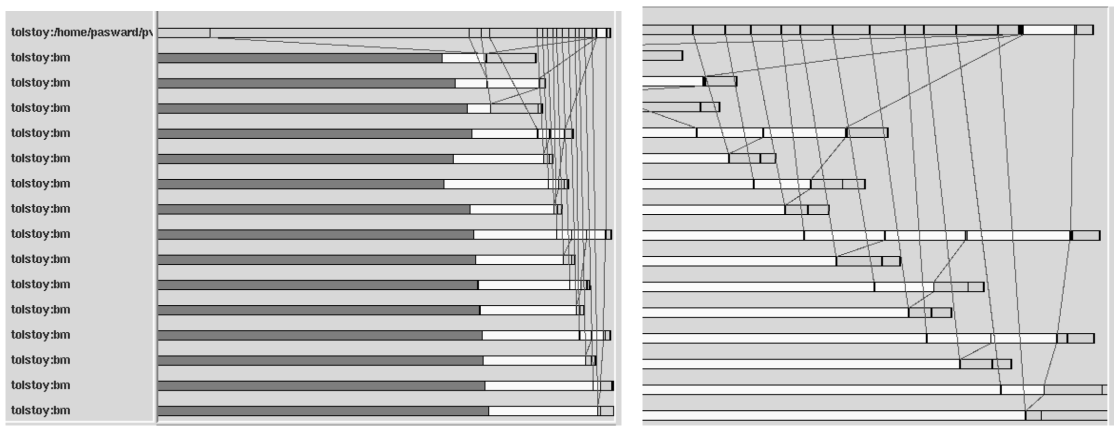
3.9 VISUALIZATION

We now look at the spatial visualization of the partial-order data structure, and the query requirements this imposes. The essence of this problem is to draw the directed graph of the transitive reduction of the partial order. Given this, there are various issues involved. The graph may be drawn based only on the partial ordering, based only on the real-times of the events, or based on the combination of these. Since no non-trivial computation induces a partial-order that can be displayed in entirety in a single screen, scrolling support is needed within the display. Clearly this scrolling will be in the time dimension. For computations involving a significant number of processes, the trace dimension will also have scrolling. An alternate approach for dealing with large computations is to display the entire computation on the screen and allow zoom-in for detailed examination. The display may be generated offline, after the event information is collected, or online, during the course of the computation. If a visualization tool cannot answer the questions that a partial-order data structure can answer (other than by careful observation), we then describe it as an output-only display. Finally, the user may wish to abstract portions of the display for various reasons. We will now describe these aspects in more detail, both in terms of the features and the required algorithms to implement those features.

3.9.1 TYPES OF DISPLAY

The most elementary visualization of the partial-order data structure that we are aware of is the XPVM [80] space-time view. It draws a horizontal bar for each PVM task. Communication between tasks is represented by a line drawn between the two tasks, with the endpoints of the line determined by the “real-time” of the transmission and reception events. We have placed “real-time” in quotation marks as, at least with the earlier versions, no attempt is made to adjust the timestamp values at each PVM task to ensure that messages are not viewed as traveling backwards in time. Such messages are referred to as *tachyons*. This problem of tachyons, combined with the absence of any other visual cue (such as an arrow), means transmit and receive events are not distinguished in the basic display. In fairness to XPVM, there is support in the GUI to determine this information on a message-by-message basis.

Real-time displays suffer from problems in addition to tachyons. The real-time of events may be such that they are clustered in some locations, while sparse in others. An example of this phenomenon is shown in Figure 3.4(a), which illustrates a master-to-slaves distribution followed by a binary merge. Given that most, though not all, of the communication is at the right hand end of the display, Figure 3.4(b) shows a focused display on this portion. Unfortunately, it now misses



(a) Wide Scale (b) Focused Scale

Figure 3.4: XPVM Displays at Different Time Scales

four of the earlier transmit events from the master PVM task, even though there is, in principle, room to show them. An alternate way of dealing with this issue is to have a time scale that is sufficient to distinguish events, and then allow for breaks within the time-axis of the display. The POET system uses this in its real-time display.

Synchronous events present a third problem for real-time displays. Logically a synchronous event is a single event that occurs simultaneously in several processes. Physically it does not. XPVM does not deal with this problem at all. It has a specific target environment, *viz* PVM, and it treats that environment as though it does not support synchronous events. In reality there are PVM operations that can be viewed as synchronous operations. For example, the pvm_barrier() function implements a barrier synchronization operation. This function is implemented by a group of tasks sending barrier messages to the group coordinator. When the coordinator has received the requisite number of barrier messages, it multicasts a continuation message to the group tasks which will block at the barrier pending the arrival of this message. XPVM does not visualize any of this communication. Rather it simply indicates in each task that it is performing a pvm_barrier() function call. The starting and ending time of these will vary by task, though in all tasks there will be an overlap point when the barrier is seen to be reached at the coordinator.

POET only supports synchronous pairs, which effectively map to synchronous point-to-point transmit and receive events. POET's solution is to show a unary "transmit" event at the real-time of the transmit. In addition, a ghost transmit event is displayed on the transmitting trace at the real-time of the receive event. An example of this is shown in Figure 3.5(a). If the receiver blocks prior to the transmit event this will be indicated as shown in Figure 3.5(b). To form a complete synchronous RPC the receiver will return the results of the computation in a mirror image operation of Figure 3.5(a).

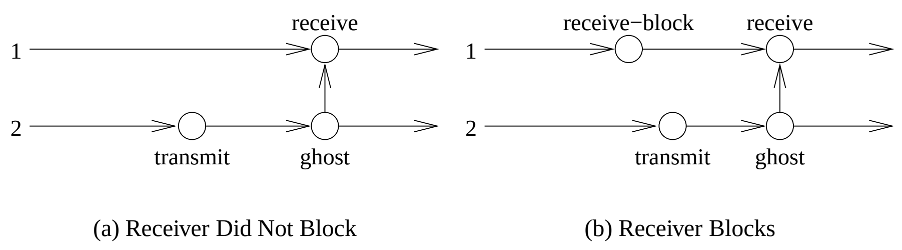


Figure 3.5: POET's Synchronous Real-Time Display

We are not aware of other monitoring systems that provide a combination of real-time and synchronous event support. In particular, we are not aware of any system that supports synchronous events comprising more than two constituent events, with or without real-time support. It is not clear how the POET solution could be extended to deal with these more-complex synchronous events. A possible solution is to use the method used for abstract-event display (see Section 3.10.1) though this might cause confusion as, in general, abstract events are not viewed as occurring at a single logical time.

Visualization based only on real-time requires little data-structure support. It must record the desired event information and the real-time of the event occurrence. For synchronous events, it must record the real-time of the event's occurrence for each process in which the synchronous event occurs. There need not be any specific connection between events recorded, as is evident in XPVM.

While displays based on the real-time of events may be useful in some applications, such as performance analysis, it can hinder others. If the objective is debugging or program understanding, then the logical ordering of events (*i.e.*, the partial order of execution) is often more relevant than the real-time ordering of events. Figure 3.6 shows a partial-order display of the PVM computation whose real-time display was illustrated in Figure 3.4. Note that this is one of many possible displays, as partial-order displays are not, in general, unique. We believe that this display more clearly delineates the logical execution of the program.

The primary issue in creating a partial-order display is choosing an event placement given the wide variety of possibilities afforded by the partial-order of execution. The most common method of event placement (used, for example, in ATEMPT [84], the Conch Lamport view [149], PVaniM [151], and others) is to maintain Lamport clocks in the distributed computation and use them to timestamp events. Event placement is then performed in a grid fashion, with the location on the grid being determined by the trace number and Lamport timestamp of the event. Figure 3.7 shows an example of this type of display, again, for the same computation as is shown in the prior figures.

One of the primary problems with this method of event placement is that it yields a rigid display, as Lamport time is totally ordered. Thus events in one trace may appear to occur substantially prior to events in another trace, even though they may be causally concurrent. The Xab approach [9, 10], or the dual timestamping method of PVaniM, go some way to dealing with this,

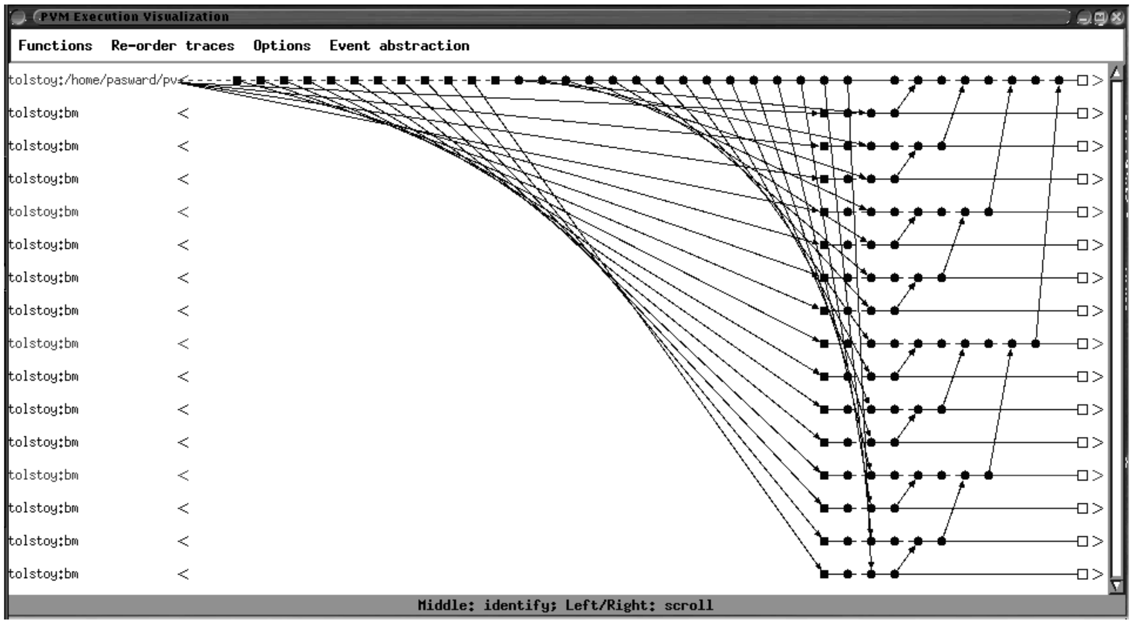

PVM Execution Visualization
Functions Re-order traces Options Event abstraction
tolstoy:/home/pasward/pv
tolstoy:bm
Middle: identify; Left/Right: scroll

Figure 3.6: POET Partial-Order Display

by moderating the Lamport time by the real-time. We will discuss that method below.

A second technique is to use the idea of phase time [98]. The approach is to divide the events into collections of concurrent events, and to display each collection as occurring at the same logical time. Figure 3.6 illustrates such a phase-time view, though phase time is not a feature of POET *per se*. A key problem with this approach is it is not possible in general to implement it. Specifically, although we claimed that Figure 3.6 shows such a display, this is not strictly true. All of the first events in all of the slave tasks are concurrent with all of the second set of transmit events in the master task (those transmit events that are received at the third event in each slave task). This is not apparent in this display. While this could be corrected for, what cannot be corrected for is the initial events in the slaves themselves. They are concurrent with each other, but not all are concurrent with the first sequence of transmit events from the master. Some are successors and some are concurrent.

Stone's concurrency maps [138] deal with the above problem by showing regions in each trace, rather than individual events, in a similar fashion to the XPVM display. Regions have starting and ending points within the time-axis. If a region R_1 in one trace is causally prior to a region R_2 in another trace then the end point of R_1 is placed at a lower time location in the time-axis than the start point of R_2, if this is possible. The *caveat* is necessary because it is not always possible to satisfy this requirement, as illustrated by Summers [140]. One valuable feature of the concurrency map is the ability to illustrate an abstract set of events within a trace that is in various precedence relationships with an event or region in other traces.

The Falcon system [37, 58, 81] thread-lifetime view and XPVM both use a method that is

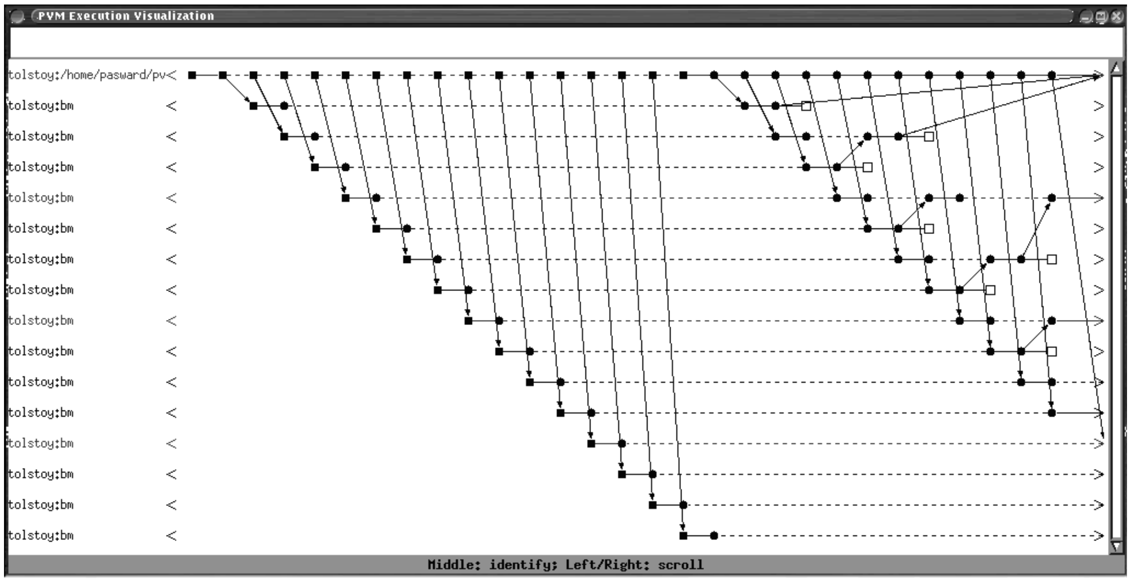

PVM Execution Visualization
tolstoy:/home/pasward/pv
tolstoy:bm
Middle: identify; Left/Right: scroll

Figure 3.7: Lamport Style Display

akin to concurrency maps, though with substantially different implementation approaches. The XPVM method, as previously noted, is not partial-order-based. The Falcon technique uses a rule-based approach. Different event types have rules associated with them to determine precedence of events as they arrive at the visualization engine. For example, there is a rule that a mutex operation must be preceded by a thread-create operation. It seems doubtful if this technique could work in general, as the complexity of these rules will grow with the number of event types and the likelihood of an error, resulting in an incorrect precedence display, would then be high.

Other techniques for implementing partial-order displays include maintaining vector clocks within the distributed computation, using causal message ordering, and topological sorting of the events based on event identity. The GOLD system [135] uses Fowler/Zwaenepoel dependency vectors within the computation, though it is unclear what they gain by this. POET uses Fidge/Mattern timestamps, but not within the computation. Rather, it computes them within the monitoring entity for the purposes of answering partial-order queries. Parulkar *et al* [116] use causal message ordering within the network monitoring system. Presumably the monitoring station must be a part of the causal-message-ordering system, and then it will never receive events out of causal order. We do not believe such a system is practical because of the high overhead of ensuring causal message ordering (it is $O(n^2)$ in the number of processes [17]). The minimal approach necessary is to order events within traces and match transmit and receive events. In other words, the monitoring entity should perform a topological sort on the events. ParaGraph [62] does this after all of the event information is collected. There is no fundamental reason why this cannot be performed online. Each event would be sent to a priority queue determined by trace id. Events processed from the priority queues would be causally ordered if all prior events in the trace were already processed and receive events had had their corresponding transmit event processed. It is therefore unclear what value is obtained by using other techniques if causal ordering of the display is the only requirement.

The basic problem with partial-order-based displays is that no single view will suffice. The reason is that any single view must order events that are unordered. A reasonable partial-order-based display must allow scrolling within the display in such a manner that events can slide relative to other events in recognition of their unordered nature. We will discuss scrolling in detail in Section 3.9.2.

The final display technique used is to combine both real-time and partial-order-based displays. There are two features of this combined approach. First, some systems provide two data displays, one real-time-based and the other partial-order-based. This is the approach taken by POET and ATEMPT [84]. The second aspect is correcting the real-time of events to ensure that it is consistent with the partial order of execution. Systems that perform such corrections include the Xab system [9, 10], POET and PVaniM [150]. There are several ways in which these corrections may be performed. The simplest is to apply the Lamport clock algorithm [96] to the real-time of events. However, it has been observed that this can significantly distort real-time values, and so Taylor and Coffin developed an alternate algorithm which minimizes such distortions [146]. The PVaniM system uses a dual-timestamp approach [150]. It presents partial-order displays using a Lamport timestamp and Gantt charts (essentially concurrency maps) using a Lamport-adjusted real-time timestamp. It is not entirely clear what the benefit is of maintaining the Lamport timestamp.

Most of the systems we have studied are tied to specific target environments. Thus, XPVM is tied to PVM, Falcon to Cthreads, and so forth. However, the features of these systems that we have discussed are generalizable beyond their specific target. POET has achieved a high degree of target-system independence [148], and works with almost a dozen different target environments.

To summarize the requirements imposed by these basic visualizations, the data structure must maintain the real-time of events, including the multiple real-times associated with synchronous events, and must be able to produce a Lamport ordering.

3.9.2 SCROLLING AND ZOOMING

Any non-trivial computation cannot be meaningfully displayed within a single screen. There are two (somewhat orthogonal) solutions to this problem. First, a detailed display may be created, and then the user can scroll within that display. Alternately, the complete display may be shown, and the user given the opportunity to zoom within that display. This problem affects both the time and trace dimensions of the display. We will deal first with time-dimension scrolling.

The vast majority of systems present a text-editor-like approach to this problem. That is, conceptually they create a fixed display of the entire partial-order. They present a portion of it, which is effectively a window over that fixed display. Scrolling is then a question of moving the location of the window over the fixed partial-order display. Given that creating such a complete display is neither generally feasible nor necessary, various optimizations are applied. Usually only a portion of the partial-order display is created, with the window into a subset of that. As long as the scrolling operation is within the portion of the display that has been computed, it will be a fairly quick operation. When it moves outside of that range a new display will have to be computed. For a real-time based display this would appear to be the preferred form of time-dimension scrolling.

Taylor [143, 144] argues fairly effectively that this is the wrong model for scrolling partial-order based displays. Specifically, in providing only a possible ordering of events, or an ordering consistent with the partial-order of execution, it effectively treats the partial-order as though it were a total order. This will give a user an incorrect view of the system. Taylor's model is to drag an event to one or other end of the currently visible portion of the time dimension. Other events are then moved as required by the partial-order constraints. Note that other events will only move if they have to, and only by as much as they have to. Thus, if an event is concurrent to the one being dragged to the edge, it will likely change its position relative to the dragged event.

In the trace dimension most systems also treat the display in a text-editor-like fashion. As with the time dimension, this is likely not a correct approach. Traces do not have order with respect to each other, except insofar as is afforded by the structure of the software being monitored. The default ordering of traces in most systems is determined by the order of arrival of events from the monitoring entity, which, other than partial-order constraints, is determined by the arrival order from the distributed computation. This may not be the best ordering possible, though it is necessary for an online display algorithm. Alternate trace orderings are offered by various systems. Both XPVM and POET allow arbitrary trace re-ordering by the user. In addition, POET offers the ability to optimize trace location and move by precedence. Trace-location optimization minimizes the apparent communication distance. This potentially corresponds to the structure of the distributed computation software in that entities that communicate frequently are likely related, while those that do not do so are likely less-connected in the software structure. Move by precedence moves traces relative to some event. Specifically, traces that have causal predecessors or successors to that event are moved closer to the trace containing that event.

The requirements on the partial-order data structure to enable display scrolling vary from none, in the case of the text-editor-like systems, to greatest-predecessor and least-successor sets for the POET scrolling algorithm. Trace movement imposes no requirements, but determining minimal apparent communication distance does require the ability to determine aggregate communication statistics by trace. For local optimization, statistics over a range of events in the trace are required.

The alternate solution to scrolling is zooming. Zooming should not be considered to be simple magnification [115], though many systems limit it to this. Rather, it should provide additional information. A very good example of such a zooming system is the System Performance Visualization Tool for the Intel Paragon [67]. The full view shows the processors in the system, with colouring to reflect activity. Zooming in shows actual load averages on a small number of processors and message traffic load to neighbouring processors. Further zooming in reveals detailed information about performance within a processor.

With this view in mind, there are several examples of zooming. Most have not traditionally been considered to be zooming, though they are in this wider view. XPVM provides traditional magnification zooming, though the original image is then lost. The information-mural approach [73] retains the original image of the entire computation, while allowing a zoom-in on a portion. While it appears that this could provide additional information on zoom-in, rather than simple magnification, it does not appear that it does. Further, it does not appear to be the focus of their work.

There are at least three other common features of monitoring systems that can be described as forms of zooming: trace clustering, event abstraction and source-code display. We will discuss trace clustering and event abstraction in detail in Section 3.10. Source-code display could be viewed as the highest level of zoom-in, though it is usually implemented in the form of querying the display, and so we will discuss it in the next section.

3.9.3 QUERYING DISPLAYS

While the primary purpose of a visualization is to present information to a user in such a manner as to allow the information to be readily apparent, for a non-trivial computation this is not possible. Some information can be made clear, but only at the expense of making other information obscure. All of the information that a user needs to know cannot be shown in a single display. Rather, the display of information prompts questions from the user. Some of these questions, such as "What happens if I drag this event to the left?", may be resolved by scrolling around the display. Others, such as "What does this look like up close?", require zooming in or out. Still other questions, such as "What is causally prior to this event?", do not really fit either the zooming metaphor or scrolling, and must be asked more directly. The large variety of questions can be grouped into two broad categories: questions pertaining to individual events and those concerning the relationship between events. We will now describe the ways in which various systems deal with these two groups of questions.

With the exception of scrolling capabilities, a significant fraction of visualization tools cannot answer questions from either of these groups. Systems such a ParaGraph simply present a display and hope that it is of value to the viewer. Such an approach we term an output-only display. It requires little data-structure support, and scrolling within it is, of necessity, text-editor-like. The typical approach in implementing such a system is to have a separate visualization engine that acquires data from the monitoring entity but does not otherwise interact closely with it.

The next level, answering questions about events, is provided for in a variety of ways. A typical minimal query capability is provided by XPVM. It allows a user to click on an event and it will indicate the task identity, the PVM function and parameters that the event represents, and the starting and ending times of the function call. ATEMPT allows the user to display the line of code that the event corresponds to, providing a simple text window into the relevant source-code file. Note that showing lines of source code does not provide the dynamic information that XPVM provides. The combined functionality of these features is achieved in Object-Level Trace (OLT [65]) which allows a direct connection between the visualization tool and a sequential debugger. As with output-only displays, little data-structure support is required to implement these capabilities. Specifically, each event requires the relevant information to be collected and stored with the event.

The most complex query types pertain to relationships between events. XPVM provides only the crudest support for this, in that a user can query messages. The result of this query is that XPVM indicates which task the message is from, which it is to, the transmission time, the message tag and the number of bytes. While this gives the illusion of value, in reality it provides no more information than is available from the simple event querying. As with event querying, it requires only that the relevant information is collected with the transmit event, with the exception of the message receipt time which must be collected from the receive event. The POET system allows several types of user query. It can display all successor, concurrent, and predecessor events to a given event. Data-structure support is required to compute these event sets. Likewise, pattern-searching and execution-difference operations, described previously, are displayed through the visualization engine. Finally, the PARASIT system [85], which is built on top of the ATEMPT system, allows a user to reorder race message outcomes and re-execute to determine the effect of this change. This requires replay-capability support in the data structure.

3.10 ABSTRACTION

Abstraction is the act of ignoring the irrelevant to focus on the essential. What is irrelevant and what is essential is very much a function of the user's current requirements. We shall discuss some of the user's criteria in the subsequent sections. In the context of our previous discussion on visualization, abstraction can be seen as zooming in on some portion of the display, or zooming out on the whole. It is zooming in because it focuses on essentials. It is zooming out because details are omitted. Particular abstractions can be seen more as one or other of these views.

However, abstraction is more than simply a component of scalable visualization. It is also a means of program understanding, and potentially enables analysis algorithms to focus on data of interest, at a level appropriate for the algorithm.

There are several desiderata in abstraction. Initially, the primary purpose was to aid user understanding, and insofar as this was achieved, the abstraction performed was a good one. Subsequently it was observed that abstraction might be used to improve data-structure performance. In principle, abstraction can help distributed management systems cope with excessive data by limiting the amount of data that must be dealt with in various algorithms and operations. For example, trace abstraction (discussed below) might allow algorithms to look only at interface events of an abstract trace rather than all events within that abstract trace. Unfortunately, at present this remains an idea in principle, not in practice. Further, we know of no current system that can use abstraction in general for the efficient realization of a partial-order data structure.

There are currently two main abstraction techniques used in distributed observation and control. These are event and trace abstraction. We now examine these abstractions, again with a focus on the requirements they would impose on our data structure.

3.10.1 EVENT ABSTRACTION

At a very basic level, event abstraction is the act of creating abstract events from primitive ones. In the most general definition, an abstract event is a set of primitive and other abstract events.

Unfortunately, most researchers use this vague definition, without formalizing it further, resulting in a number of errors and inconsistencies. We will therefore create our own formalism and express current methods as well as possible in that formalism, noting where problems or omissions occur.

First, since events may now be primitive or abstract, we will henceforth preface the term "event" with either "primitive" or "abstract" when we wish to refer to events of that particular type. Without the preface, the term "event" will refer to events of either type. Second, we explicitly require that an abstract event cannot contain itself, either directly or through a recursive chain. Third, since abstract events will be permitted to contain other abstract events, we will create an abstraction hierarchy. We define level 0 of the hierarchy, $\mathcal{A}_0$, to be the set of abstract events that each contain a single primitive event. Thus

$$\mathcal{A}_0 = \{\{e\} \mid e \in \mathcal{E}\} \tag{3.4}$$

We now define the higher levels of the hierarchy. To reduce complexity, we require that abstract events at level N can only be composed of abstract events at level $N-1$, rather than of arbitrary abstract events in any of the lower levels. Thus for $N > 0$

$$\mathcal{A}_N \subseteq 2^{\mathcal{A}_{N-1}} \tag{3.5}$$

This definition does not force abstract events to be non-overlapping, although most researchers seem to implicitly assume this. However, we do not believe this to be a significant issue, as it does not appear to affect any of the following analysis.

We describe an abstract event that is composed of a single event as "trivial." It is useful to formalize an equivalence of trivial abstract events with their component event, thus.

$$\left\{a^j\right\} = a^j \tag{3.6}$$

Note that this equivalence means that not only does every abstract event defined at level i exist as a trivial abstract event at level $i+1$, but also in every higher level of the abstraction hierarchy. Likewise, note that $\mathcal{A}_0$ contains only trivial abstract events. An abstract event that is composed of two or more events is "non-trivial."

The set of all abstract events is then the union of these sets.

$$\mathcal{A} = \bigcup_{i=0}^{\infty} \mathcal{A}_i \tag{3.7}$$

Note that this definition does not imply that there are infinitely many abstract events. Specifically, the equivalence of Equation 3.6 constrains the total to be not more than all possible enumerations of primitive events.

For the remainder of this dissertation we will use the letter a, possibly sub- or superscripted, to indicate an abstract event. We will use the subscript to refer to the abstract-set level in which the event resides. Thus $a_i \in \mathcal{A}_i$.

In addition to these simple definitions, various researchers have proposed further structure on abstract events. Most of this structure has revolved around two issues. The first issue is attempting to make abstract events in some sense atomic. What this means in practice is that after the inputs are satisfied, the abstract event should be able to execute to completion, without an intervening dependency.

The second issue is the question of what precedence relationships should be defined between abstract events. There are several possibilities for extending the primitive precedence relation. The first requirement, satisfied by all proposed extensions, is that it suitably degenerate to the correct case for abstract events that are composed of single primitive events. That is

$$\{e_i\} \prec_{\mathcal{A}_0} \{e_j\} \Longleftrightarrow e_i \prec_{\mathcal{E}} e_j \tag{3.8}$$

where $\prec_{\mathcal{A}_0}$ is the precedence relation for abstract events in abstract set $\mathcal{A}_0$. In addition to this requirement, it is usually presumed that the precedence relation over all abstract events, $\prec_{\mathcal{A}}$, is the union of the precedence relations over the individual levels.

$$\prec_{\mathcal{A}} = \bigcup_{i=0}^{\infty} \prec_{\mathcal{A}_i} \tag{3.9}$$

It is not always clearly defined what precedence means between abstract events at differing levels in the abstraction hierarchy, or even if such a definition is meaningful. Given our previous definition of trivial events, we will presume that precedence is resolved by “raising” the lower abstract event to the same level in the hierarchy as that event with which it is being compared, and then using whatever precedence relation is defined at that level.

$$a_i^k \prec_{\mathcal{A}} a_j^l \Longleftrightarrow \begin{cases} \{a_i^k\} \prec_{\mathcal{A}} \; a_j^l & i < j \\ a_i^k \prec_{\mathcal{A}_i} a_j^l & i = j \\ a_i^k \prec_{\mathcal{A}} \{a_j^l\} & i > j \end{cases} \tag{3.10}$$

The definition of “covers” remains as in Equation 10, though it uses the abstract-event precedence relation.

$$a^i <: a^j \Longleftrightarrow a^i \prec_{\mathcal{A}} a^j \wedge \nexists_{a^k} a^i \prec_{\mathcal{A}} a^k \wedge a^k \prec_{\mathcal{A}} a^j \tag{3.11}$$

The final assumption in most prior work is that the definition of concurrency is essentially unchanged from that of Definition 18, altering only the precedence relation from that over primitive events to that over abstract events. It is unclear whether this model of precedence and concurrency is a good one for abstract events. Specifically, where primitive events are atomic, non-trivial abstract events have duration. As such, it is not clear that their causal inter-relationship will form a partial order. Disclaimers aside, we will now describe the various forms of abstract-event precedence.

Kunz [89] lists two obvious possible definitions for abstract-event precedence. First, all primitive events in one abstract event must precede all primitive events in the other abstract event:

$$a^i \prec_{\mathcal{A}} a^j \Longleftrightarrow \forall_{e_k^l \in a^i} \forall_{e_m^n \in a^j} \; e_k^l \prec_{\mathcal{E}} e_m^n \tag{3.12}$$

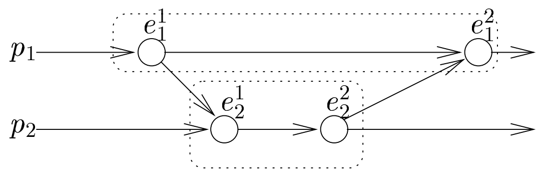


Figure 3.8: Ordered Primitive Events forming Concurrent Abstract Events

Note that while this definition is in terms of precedence of primitive events, it is is isomorphic to the definition based solely on abstract events:

$$a^i \prec_{\mathcal{A}} a^j \iff \forall_{a^k \in a^i} \forall_{a^l \in a^j}\ a^k \prec_{\mathcal{A}} a^l \tag{3.13}$$

This approach yields a partial order over the set of abstract events, which has some appeal. However, there are few abstract events for which precedence would then hold [27]. It would also require a modification of the definition of concurrency. The existing definition would imply that abstract events were concurrent even though there might be significant communication between constituent events. Indeed, the constituent events could be totally ordered, but the abstract events considered concurrent under the present definition (See Figure 3.8).

The second obvious approach is that some primitive event in one abstract event must precede some primitive event in the other abstract event.

$$a^i \prec_{\mathcal{A}} a^j \iff \exists_{e_k^l \in a^i} \exists_{e_m^n \in a^j}\ e_k^l \prec_{\mathcal{E}} e_m^n \tag{3.14}$$

Likewise, this is isomorphic to the equivalent definition using just abstract events. While this definition requires no modifications to the definition of concurrency, it has the interesting property that it is neither transitive nor anti-symmetric. The Ariadne debugger [27] takes this approach as is, but defines the *overlaps* relation for such cases of symmetric dependency between abstract events. It does not appear to deal with lack of transitive dependency at all. Cheung [20], Summers [140], Basten [7] and Kunz [89] also take this view, but attempt to prevent symmetric abstract events from being created by imposing further structure on abstract events. For the remainder of this section we will use this precedence definition while describing the work of Cheung, Summers, Basten, and Kunz.

To add additional structure, both Cheung and Summers define the input and output event sets for an abstract event. These are as follows:

Definition 21 (Input Set: $I \subseteq a$**)** *The input set, I, of abstract event a is the set of events in a which have an immediate predecessor that is not in a.*

$$I = \big\{ i \mid i \in a \wedge \big(\exists_{a^j}\ a^j <: i \wedge a^j \notin a\big)\big\} \tag{3.15}$$

The output set is defined analogously:

Definition 22 (Output Set: $O \subseteq a$) *The output set, O, of abstract event a is the set of events in a which have an immediate successor that is not in a.*

$$O = \big\{o \mid o \in a \wedge \big(\exists_{a^j}\, o <: a^j \wedge a^j \notin a\big)\big\} \tag{3.16}$$

Note that although this has been written in terms of arbitrary abstract events, it appears that both Cheung and Summers had in mind abstract events in $\mathcal{A}_1$. Given these definitions, three classes of abstract events were defined: the central-event, the complete-precedence, and the contraction.

The central-event class of abstract events requires that there exist an event within the abstract event such that all input events precede it and all output events succeed it. This definition does ensure that the precedence relation over abstract events is both anti-symmetric and transitive. However, it appears to be excessively restrictive. A simple two-way information exchange between processes could not be formed into an abstract event, even though this might be desirable. The problem is the requirement for a single event through which all precedence must flow.

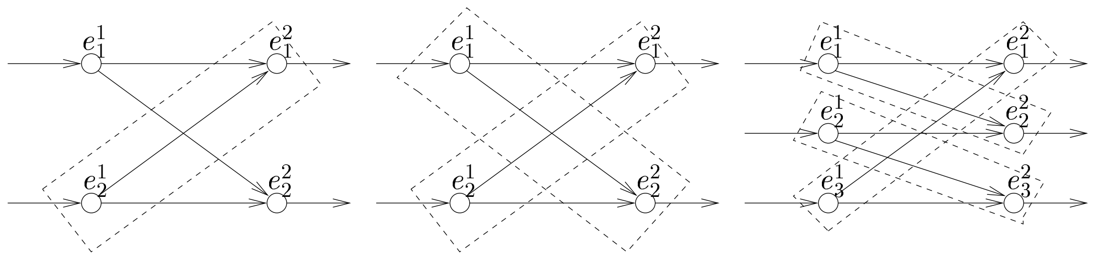
The complete-precedence class of abstract events relaxes this restriction and enforces only that all input events must precede all output events. This definition still ensures that the precedence relation over abstract events is a partial order. This is probably the weakest structural restriction that allows arbitrarily interconnected abstract events and still provides this guarantee. However, it is arguably too restrictive a definition. It does not, for example, permit multiple concurrent events to be a single abstract event. This particular usage of abstract events would be prevalent for parallel computations in which multiple processes perform the same actions on different data (the SPMD model).

Cheung studied a more general class of abstract event, contractions, which was first defined by Best and Randell [11]. The definition is as follows:

Definition 23 (Contraction)

1. *All primitive events are contractions*
2. *An abstract event a_i^j is a contraction if and only if*
   (a) $\forall_{a_{i-1} \in a_i^j}\; a_{i-1}$ *is a contraction*
   (b) $\forall_{a_i^k ; a_i^l}\; a_i^k <:_{\mathcal{A}_i} a_i^j <:_{\mathcal{A}_i} a_i^l \Longrightarrow \exists_{a_{i-1} \in a_i^j}\; a_i^k \prec_{\mathcal{A}_{i-1}} a_{i-1} \prec_{\mathcal{A}_{i-1}} a_i^l$

A contraction that has only one immediate predecessor and one immediate successor is a simple contraction. Systems of simple contractions retain the anti-symmetric and transitive properties with respect to precedence. As such they have some appeal. The EBBA Toolset [8] is an example of a distributed debugging system using simple contractions for abstraction. Strictly speaking they are less restrictive than complete-precedence events. The manner in which they are less restrictive is that there can be input events that do not precede output events and output events that are not preceded by input events. This appears to have no practical value. Contractions that are not simple cannot be arbitrarily interconnected and reflect the transitivity of the underlying event set. Summers created a set of rules for interconnection that would ensure transitivity, though it is not clear that this could be efficiently used in practice.



(a) Abstract Event $\{\{e_2^1\},\{e_1^2\}\}$ (b) Are These Convex? (c) Cyclic Precedence

Figure 3.9: Problems with Convex Abstract Events

Given the problematic nature of ensuring transitivity, Kunz [89] and Basten [7] abandoned it as a requirement, but attempted to maintain anti-symmetry by defining the class of convex abstract events as follows:

Definition 24 (Convex Abstract Event) *An abstract event a_i is convex if and only if*

$$\forall_{a_{i-1}^j, a_{i-1}^k \in a_i \,;\, a^l \in \mathcal{A}}\; a_{i-1}^j \prec_{\mathcal{A}} a^l \wedge a^l \prec_{\mathcal{A}} a_{i-1}^k \Rightarrow a^l \in a_i$$

Several notes should be made about this definition. First, the terminology is somewhat imprecise, since it is taken, essentially as is, from the Kunz's dissertation. He does not clearly indicate exactly what set the precedence relation is over, and the most we can presume is that it is over the whole set of abstract events. Further, it is not clear if an abstract event a_i is convex if

$$\exists_{a_{i-1}^j, a_{i-1}^k, a_{i-1}^l \in a_i \,;\, a^m \in \mathcal{A}}\; a_{i-1}^j \prec_{\mathcal{A}} a^m \wedge a^m \prec_{\mathcal{A}} a_{i-1}^k \wedge a^m \notin a_i \wedge a^m \in a_{i-1}^l$$

It could be argued that this is due to a poor formalism on our part, though no alternative has been suggested. This problem could be corrected by changing the requirement from membership in the abstract event to membership in the event or in any of its subcomponents. The difficulty with this alteration is in the cost of testing for convexity. However, a deeper problem exists, exhibited in Figure 3.9. In Figure 3.9(a) we see that $\{\{e_2^1\},\{e_1^2\}\}$ is a legal convex abstract event. However, we now cannot form $\{\{e_1^2\},\{e_2^1\}\}$ into a convex abstract event, since $\{\{e_2^1\},\{e_1^2\}\}$ will be both a successor and a predecessor, but will not be part of the new abstract event. On the other hand, if $\{\{e_2^1\},\{e_1^2\}\}$ had not yet been formed into an abstract event, then $\{\{e_1^2\},\{e_2^1\}\}$ would be a perfectly legal convex abstract event. Clearly it is problematic for the legality of convex abstract events to depend on the order in which they are created.

Basten provides a slightly different definition.

Definition 25 (Convex Abstract Event) *An abstract event a_i is convex if and only if*

$$\forall_{a_{i-1}^j, a_{i-1}^k \in a_i \,;\, e \in \mathcal{E}}\; a_{i-1}^j \prec_{\mathcal{A}} e \wedge e \prec_{\mathcal{A}} a_{i-1}^k \Rightarrow e \in a_i$$

Again, the terminology is somewhat imprecise. He uses $\preceq$ though this will not alter the substance of the definition. He also does not clearly indicate exactly what set the precedence relation is over, and the most we can presume is that it is over the whole set of abstract events. The difference is that he requires the intervening event be a primitive one. Thus, in Figure 3.9(b), both $\{\{e_2^1\},\{e_1^2\}\}$ and $\{\{e_1^2\},\{e_2^1\}\}$ are legal convex abstract events, and the order in which they are created does not affect their legality. The notion behind this definition is that an abstract event, once all primitive input events have occurred, can complete execution independently of external influence. This was the premise behind contractions, and thus it is a reasonable one, and appeared easier to work with than contractions.

Unfortunately, this definition too has deep problems. First, observe in Figure 3.9(b) that abstract events are symmetric. Worse, in Figure 3.9(c), the abstract events, legal convex events according to Definition 25, are not symmetric but form a precedence cycle. It is one thing to abandon the transitivity of precedence, but quite another when precedence is cyclic. We know of no simple way to fix these problems.

This brings us back to the core issue of this, which is the data-structure requirements imposed by abstract events. There are two levels at which these requirements may be formed. First, a client of the data structure may wish to simply compute abstract events according to an algorithm of its choosing. In this regard, the data structure must provide the necessary precedence-determination algorithms, as required by many previous operations. At a deeper level, however, we would like to incorporate an abstract-event hierarchy within the data structure. Thus, the data structure would be responsible for maintaining the hierarchy, ensuring that abstract events met the appropriate criteria, and keeping track of the various abstract events to which primitive events belonged. Having dissected the various event-abstraction algorithms and found them lacking, we choose not to impose such a requirement on our data structure at this time.

Two final comments must be made about event abstraction. First, the data-structure requirements prior to our work have largely been formed around the question of what timestamps are required by abstract events to reflect the relevant precedence relationship. The timestamps developed have in turn been based on Fidge/Mattern timestamps at the primitive-event level. This is probably not a good approach.

Second, many distributed observation systems do not display or otherwise process abstract events. The ATEMPT system incorporates edge contractions, a subset of contractions. Specific unary events may be merged in the display into a single abstract event. This is displayed using a triple event object, to indicate that it is hiding multiple unary events. The POET system incorporates convex abstract events. Figure 3.10 shows a possible event-abstraction view of the binary-merge computation shown in Figures 3.4 and 3.6. The abstract events are rectangles, usually covering multiple traces, with solid coloration on traces where primitive events exist as constituents of the abstract event.

3.10.2 TRACE ABSTRACTION

Just as event abstraction is the act of creating abstract events from primitive ones, trace abstraction creates abstract traces from primitive ones. In the most-general definition, an abstract trace is a set

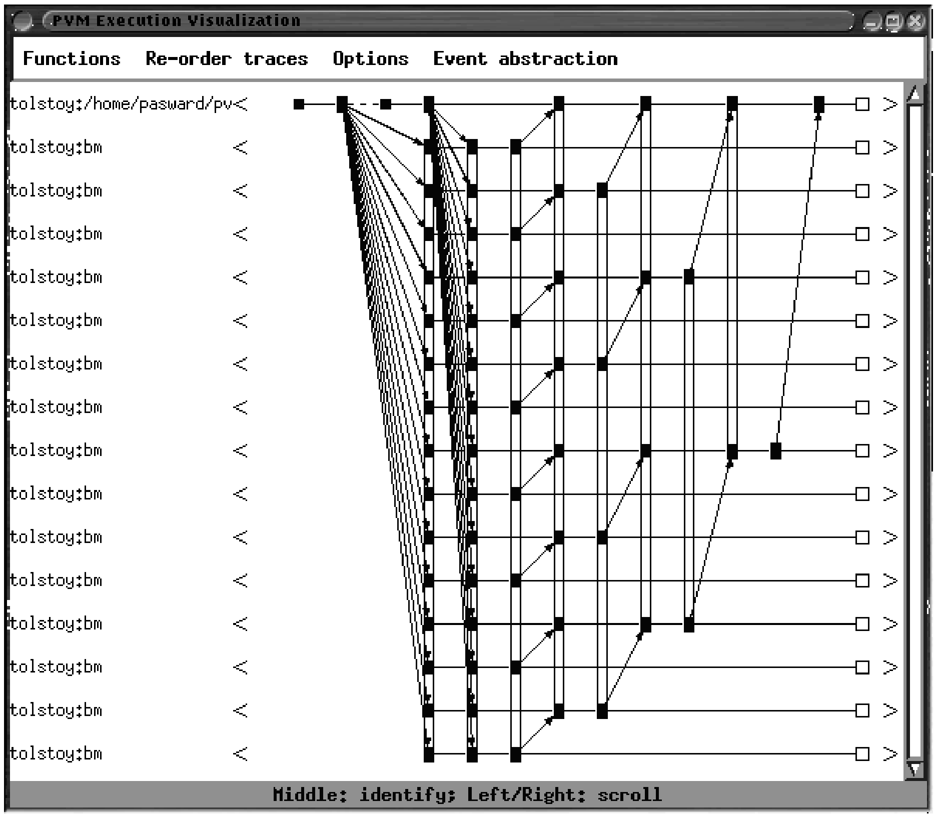


Figure 3.10: POET Event Abstraction

of primitive and other abstract traces. It is usually implicitly assumed that an abstract trace cannot contain itself, either directly or through a recursive chain. This yields a hierarchy of abstract trace sets, with primitive traces at the base. We therefore define sets in an analogous fashion to abstract event sets.

$$\mathcal{P}_N \subseteq \begin{cases} 2^{\mathcal{P}_{N-1}} & N > 0 \\ \{\{p\} \mid p \in \mathcal{P}\} & N = 0 \end{cases} \tag{3.17}$$

where $\mathcal{P}_N$ is the set of abstract traces at level N. As with abstract events, we describe an abstract trace that is composed of a single trace as "trivial" and formalize the equivalence of trivial abstract traces with their component trace, thus.

$$\{p^j\} = p^j \tag{3.18}$$

An abstract trace that is composed of two or more traces is "non-trivial." The set of all abstract traces is

$$\mathcal{P} = \bigcup_{i=0}^{\infty} \mathcal{P}_i \tag{3.19}$$

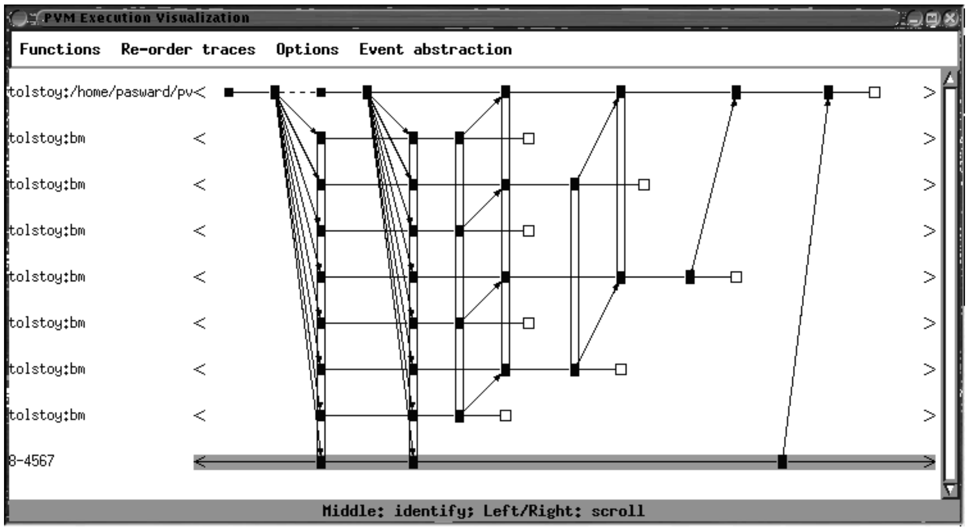
which is finite for the same reason that the set of all abstract events is finite.

As in event abstraction, the primary problem is to deal with the correct representation and manipulation of event precedence. In this regard, Cheung [20] defined a consistent interface cut as follows. Consider a set of traces, which are to be divided into two abstract traces. Create a virtual interface, the interface cut, between the two abstract traces, which intersects messages that are passed between the abstract traces. Every message between the abstract traces creates two events in the interface cut, a receive from the sending abstract trace and a transmit to the receiving abstract trace. Such an interface cut is consistent if the events it is composed of are ordered such that no new precedence relationships are implied by its presence.

There are several problems with this idea of a consistent interface cut. First, it cannot exist in the general case, since Cheung required it to be totally ordered. His solution to this was to require that an abstract-trace interface point have a representative trace through which all communication would flow. This may appear to be an excessive restriction, though it is not difficult to overcome in practice. It is possible to declare there to be multiple interfaces, one for each communicating trace between the two abstract traces. This approach is effectively equivalent to a single partially-ordered interface, composed of multiple totally-ordered interface cuts. Note that creating a minimal set of such cuts would be NP-hard.

A more serious problem, observed by Taylor [145], is that a consistent interface cut can cause user confusion. Specifically, the events on the interface cut are totally ordered even though the communication that they represent may not be so ordered. This point remains true even with abstract traces that have a representative trace. As such, POET takes the view that events on the interface should maintain the same partial-order relationship as the communication events to which they correspond. The effect of this is to move from a symmetric view, where the objective is to maintain the same view on both sides of the abstract-trace interface, to an asymmetric one, where only the traces inside the abstract trace are used to determine the interface. The quality of the resulting abstract traces is then a function of the quality of the clustering. If traces that communicate together frequently are placed in the same abstract trace, and in distinct abstract traces from those with which they communicate infrequently, then the abstract traces should be of good quality. If the clustering is poorer, then it is more likely that the abstract traces will require as many totally-ordered interface cuts as the real traces they represent.

The primary work of data structures for trace abstraction revolves around the problem of precedence-relationship determination in the presence of trace abstraction. Specifically, as with event abstraction, the issue has tended to be posed in terms of what is the appropriate timestamp for an event given that it is in an abstract trace and other traces are also in abstract traces. This can be decomposed into two distinct problems. The first is the timestamp determination for the events that are in the interface cuts. Cheung provides an algorithm for this in term of Fidge/Mattern timestamps. The second issue is determining if it is possible to reduce the amount of information that needs to be processed by the observation system when using trace abstraction. Both Cheung and Summers give abstract-trace-timestamp algorithms, but both are offline and require an initial calculation of Fidge/Mattern timestamps for all events. It does not appear to be the case that integrating trace abstraction with the partial-order data structure, as opposed to merely allowing



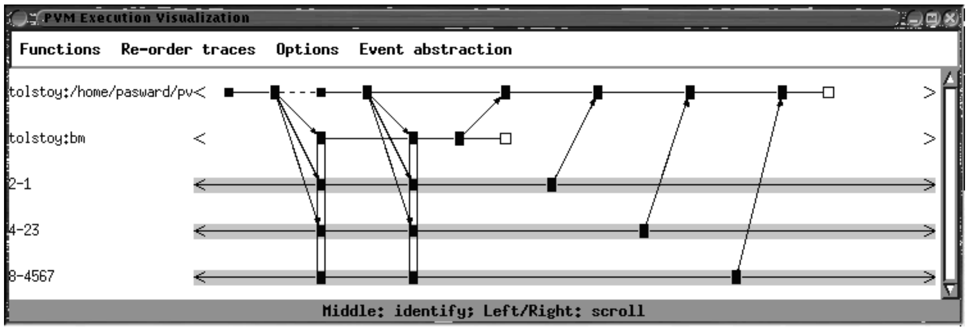


Figure 3.11: POET Trace Abstraction

clients to compute trace abstractions, would be of particular value.

Displays of abstract traces tend to be similar to displays of real traces, with some simple marker to indicate that it is an abstract, rather than a single, trace. The ATEMPT system simply labels the relevant interface cut with the user-selected name. It is able to do this as no effort is made to have a correct interface cut, which in the general case requires more than a single trace line. The POET system uses an alternate background colour for the cuts that compose the interface. An example of the POET trace-abstraction method is shown in Figure 3.11. It is the same binary-merge computation as that of Figure 3.10, with the same abstract events. The difference is that in the top illustration the lower eight processes have been abstracted into a single abstract trace and in the bottom one a hierarchy of trace abstractions has been created.

The figures also have some errors in them, as a result of incorrect interaction between trace and event abstraction. Specifically, abstract events that have constituents both within and without the abstract trace do not have an event displayed at the abstract-trace interface trace if those events within the abstract trace are not communication events with partner events outside the abstract trace. Thus, the third multi-trace abstract event in the upper figure fails to show an event in the interface trace even though that abstract event spans the abstract trace. The essence of this problem is that trace abstraction is not completely orthogonal to event abstraction. In particular, event abstraction can cover multiple traces, as well as multiple events within a trace. The lower figure has an additional problem that the bottom interface trace abstract events do not have the transmissions to them displayed. This problem may simply be a code defect, since there is no clear abstraction-interaction issue.

3.11 Summary of Requirements

We now briefly enumerate the requirements, as identified above.

1. Event lookup
2. Basic and extended information store, containing at least
   (a) real time(s) of event
   (b) line of code of event
3. Determining precedence relationships between events
4. Trace and non-trace predecessor and successor identification
5. Greatest predecessors and least successors of an event
6. Partner identification
7. Longest (anti-)chain determination
8. Aggregate communications statistics by trace
9. Statistics over events within a trace

4 Partial-Order-Data-Structure Interface

Having identified the general operations and the associated queries that they imply, we enumerate the formal interface for the partial-order data structure. We do so in the form of a set of C++ classes. The major classes are PartialOrder, CutRef, SliceRef, EventRef, EventID, EventPosition, and TraceID. In addition to these, there are iterators associated with the PartialOrder and SliceRef classes. We now describe these classes, starting with the simplest and working up to the PartialOrder class.

4.1 Event Identifiers

The event identifier is composed of the event position and trace. These classes are intended to be lightweight, primarily intended to allow comparison within a trace and to facilitate event lookup. We describe the position and trace classes first, and then the composite.

The EventPosition class has the following definition.

```
class EventPosition {
public:
   EventPosition();
   EventPosition(const unsigned int);
   EventPosition(const EventPosition&);
  ~EventPosition();
   const EventPosition& operator= (const unsigned int);
   const EventPosition& operator= (const EventPosition&);
   const EventPosition  operator+ (const int) const;
   const EventPosition  operator- (const int) const;
   const EventPosition& operator+=(const int);
   const EventPosition& operator-=(const int);
   const EventPosition& operator++();
   const EventPosition& operator--();
   const EventPosition  operator++(int);
   const EventPosition  operator--(int);
   bool operator==(const unsigned int) const;
   bool operator==(const EventPosition&) const;
   bool operator!=(const unsigned int) const;
   bool operator!=(const EventPosition&) const;
   bool operator< (const unsigned int) const;
   bool operator< (const EventPosition&) const;
   bool operator<=(const unsigned int) const;
   bool operator<=(const EventPosition&) const;
   bool operator> (const unsigned int) const;
   bool operator> (const EventPosition&) const;
   bool operator>=(const unsigned int) const;
   bool operator>=(const EventPosition&) const;
};
```

The specification of these methods is essentially as might be expected given the definition of an event position in Section 2.2. For example, the comparison operations correspond to the process-precedence equations (Equations 2.4 and 2.7), where it is assumed that the event positions occur on the same trace. The significant differences are as follows. The event position created by `EventPosition(0)` does not correspond to an event (*per* the definition of event position, they are assigned natural numbers), but rather is that position prior to the first event. In like fashion, the decrement operations will not decrement prior to this position.

There is no upper bound on the event position, though the interface does limit assignment or comparison with unsigned integers to UINT_MAX (typically defined in /usr/include/limits.h). It is expected that an implementation of this class will impose an upper bound on event positions. In doing so it should ensure that the increment operations do not wrap.

Note that the postfix increment and decrement operations return a copy rather than a reference. The reason for this is that they are not returning the value after the alteration, and thus cannot return a constant reference to the object. Rather, they must return the value of the object prior to the alteration. This can only be done by making a copy of the object and returning that. Likewise, the addition and subtraction operators must return a copy, rather than a reference. The expense in such operations is not in the copy. Indeed, this class and the subsequent TraceID and EventID classes are sufficiently lightweight that copying is often cheaper than the dereference cost required when the reference is used. However, returning a copy rather than a reference requires the constructor to be invoked. Since most usage of these operators discards the result, the destructor is then immediately invoked. It is generally preferable to evade this by returning a constant reference to the object, and that will be our practice for most of the remaining classes.

Finally, for reasons that will become apparent in Section 4.2, the class must provide a non-position EventPosition. That is, any comparisons (excluding equality) with this non-position shall result in an exception being thrown. Likewise, incrementing or decrementing it will result in an exception.

All of the methods of the EventPosition class should probably be implemented inline.

The TraceID class has the following definition.

```
class TraceID {
public:
   TraceID();
   TraceID(const TraceID&);
  ~TraceID();
   TraceID& operator= (const TraceID&);
   bool     operator==(const TraceID&) const;
   bool     operator!=(const TraceID&) const;
};
```

The TraceID class is relatively simple because trace identifiers are not ordered in the same way that event positions are. Rather, the identifier simply has to be unique relative to other trace identifiers. Although the constructor is public, TraceID objects created outside of the partial order are not in general meaningful. The purpose of the public constructor is for client code to be able to use TraceID objects returned from the partial order. As with the EventPosition class, the TraceID class must provide a non-trace TraceID. It is expected that the TraceID class will be implemented inline.

The EventID class encapsulates the EventPosition and TraceID, providing a unique identifier for every event. It provides most of the methods of the EventPosition class, though with slightly modified specification. First, since it is not meaningful to compare an event identifier with an unsigned integer, those methods from the EventPosition class are not part of the EventID class. However, we do retain (in)equality testing with integers. This is solely for the purpose of testing against 0, which is considered to be equal to that EventID which contains either the non-position EventPosition or the non-trace TraceID. We refer to such an EventID as the NULL EventID.

Second, as the class contains the trace identifier, comparisons between EventID objects with differing traces will result in an exception being thrown. It is not the intent of this class to provide partial-order event-comparison capability. Rather it is providing the $\prec_p$ and $\preceq_p$ capability *per* the EventPosition class. It should also be noted that the multiple event identifiers of synchronous events will not be considered equal. To determine their equality requires examination of the Event object. Third, there are member access functions for the TraceID and EventPosition. They both enable the provision of information and the direct alteration of the EventPosition or TraceID.

The formal specification of the EventID class is as follows.

```
class EventID {
public:
   EventID();
   EventID(const EventID&);
   EventID(const TraceID&, const unsigned int);
   EventID(const TraceID&, const EventPosition&);
  ~EventID();
   const EventID& operator= (const unsigned int);
   const EventID& operator= (const EventID&);
   const EventID& operator+ (const int) const;
   const EventID& operator- (const int) const;
   const EventID& operator+=(const int);
   const EventID& operator-=(const int);
   const EventID& operator++();
   const EventID& operator--();
   const EventID  operator++(int);
   const EventID  operator--(int);
   bool           operator==(const EventID) const;
   bool           operator!=(const EventID) const;
   bool           operator==(const int) const;
   bool           operator!=(const int) const;
   bool           operator< (const EventID) const;
   bool           operator<=(const EventID) const;
   bool           operator> (const EventID) const;
   bool           operator>=(const EventID) const;
   TraceID&       traceID();
   const TraceID& traceID() const;
   EventPosition& eventPosition();
   const EventPosition& eventPosition() const;
};
```

Note that as with with the TraceID class, creating an EventID outside of the partial order is not in general meaningful. It is only useful to enable EventID objects to be used by client code. As with the constituent classes, all methods are expected to be implemented inline.

4.2 Slices and Cuts

A slice is a subset of the set of events such that there is at most one event per trace. It may be thought of as the boundary events of a cut (not necessarily a consistent cut). Indeed, a cut, again not necessarily a consistent cut, is the closure of a slice over the $\preceq_{\mathcal{P}}$ relation. Thus

$$\forall_{e^1 \in \mathcal{E}} \forall_{e^2 \in \sigma}\, e^1 \preceq_{\mathcal{P}} e^2 \Longrightarrow e^1 \in \kappa \tag{4.1}$$

where σ is a slice and κ is its corresponding cut.

The need for the distinction between slices and cuts is that the operations differ. For example, iterating over a slice is a reasonable operation, costing $O(N)$ where N is the number of traces. By contrast, iterating over a cut could be extremely expensive, since the cut could cover most of the events within the partial order. On the other hand, comparing cuts and checking containment are reasonable operations that make less sense on slices. Due to their similarity, easy conversion between the two types is required.

In general slices and cuts will require a non-trivial amount of space. As such copying is a poor choice. On the other hand, slices and cuts are not guaranteed to be integral components of the partial order. That is, their presence is not necessarily required to maintain the partial order. As such they will need to have space allocated, and subsequently deleted when it is no longer needed. This effectively rules out the use of C++ references. The remaining choice, C++ pointers, is unattractive for several reasons. First, it leaves the correct memory-management deletion operations up to the client code. This is problematic for several reasons, not least of which is that it divides the allocation and deallocation of memory between different entities (*viz* the partial-order data structure library will allocate and the client code will deallocate) which invariably results in memory leaks. Second, slices can have iterators and the lifetime of the iterator must be tied to that of the slice, otherwise it leads to the possibility of using an iterator on a deleted slice, with undefined results. In particular, detecting and fixing such defects is not always trivial. Third, it makes the use of overloaded operators substantially less clear in client code.

We therefore opt to specify slices by means of a SliceRef class which, when implemented, provides the necessary memory management and limits the required copying. We use a variant technique from one described in Question 16.22 of the C++ FAQ Lite [23]. It provides Java-like reseatable references, rather than C++-style aliases or the copy-on-write semantics of the C++ FAQ Lite approach. The former is insufficient, as we have already noted, while the latter is more expensive and less flexible than our approach. The scheme can be subverted, but not accidentally. Further, it is not clear what value would be obtained by such a subversion.

The specification for the SliceRef class is as follows.

```
class SliceRef {
public:
   SliceRef();
   SliceRef(const CutRef&);
   SliceRef(const SliceRef&);
  ~SliceRef();
   const SliceRef&      operator= (const CutRef&);
   const SliceRef&      operator= (const SliceRef&);
   EventPosition&       operator[](const TraceID&);
   const EventPosition& operator[](const TraceID&) const;
   const EventID        operator()(const TraceID&) const;
   const SliceRef&      operator+ (const int) const;
   const SliceRef&      operator- (const int) const;
   const SliceRef&      operator+=(const int);
   const SliceRef&      operator-=(const int);
   const SliceRef&      operator++();
   const SliceRef&      operator--();
   const SliceRef       operator++(int);
   const SliceRef       operator--(int);
private:
   class Data {
   public:
      Data();
      Data(const Data&);
     ~Data();
      mutable unsigned int _count;
      // Other slice data
   };
   void            operator&();
   void*           operator new(size_t);
   Data*           _slice;
};
```

First note that the private portion is part of the interface specification. An implementer is not free to change this as desired, though it may be augmented. Several facts about this private portion should be emphasized. First, the class is reference-like in size. It should require no more than a pointer. This is important since it allows us to treat the class as a reference, having multiple copies of a given slice, without incurring a substantial space or performance penalty. In particular, the copy constructor needs only to allocate and initialize this pointer and the count field of the Data subclass. Second, the address and new operators are private. This is to prevent the accidental subversion of the memory-management scheme. Though we could have provided a SlicePtr class and tied it into the memory management scheme, it is not clear that this would provide substantial additional value. The smallness of the slice class enables us to return copies rather than references, and thus have only automatic copies. Third, the Data subclass maintains the actual slice data, and a count variable for memory management purposes. Any alternate techniques that provide the requisite data storage and memory management capacity are permissible.

The public methods of the SliceRef class can be broken down into four categories. First, it contains the necessary constructors and destructor. These behave essentially as expected, with the exception of the default constructor. In normal C++ usage, references are aliases that are not reseatable. They must therefore be assigned at their declaration. The SliceRef class, however, specifies a reseatable reference. As such its default constructor does not require it to reference a slice at declaration. However, it is probably not meaningful to invoke methods on such a reference. There are at least two solutions to this. The implementer may constrain such a SliceRef object to have only one valid method, which is assignment. The invocation of any other methods (excluding deletion) will cause an exception to be thrown. This approach requires that every method invocation first verify that the _slice pointer is not null. This unnecessarily degrades runtime performance. An alternate solution, as presented in the C++ FAQ, is to use statically assigned data for the default constructor. In our application a slice with no members for every trace would be appropriate. Normal usage of the SliceRef class will likely never use this specific slice, but it allows the implementer to avoid the check on the _slice pointer.

The assignment operator will reseat the reference. In doing so it is responsible for ensuring that its current referent slice is correctly disposed of as needed. Likewise, it must ensure that the new referent is aware of the additional reference. The operator shall not make a copy of the slice data, and is thus a relatively cheap operation. Due to the potentially problematic nature of assigning a slice from a cut, the interface also specifies an assignment operator for that purpose. That assignment operator must not simply reseat the reference, since it is a reference to a different object type. Rather, it must make a copy, though it may do so using copy-on-write semantics, *per* the C++ FAQ.

Third, the SliceRef class provides access by trace identifier to the corresponding event, if any.

This can be used to determine or alter the event position recorded for a given trace. Alternately, it will provide a copy of the event identifier for the event recorded for that trace. We note at this point that access to a trace for which no event is recorded, per the definition of slice, must return a non-position EventPosition.

Finally, a slice may be incremented or decremented. To do so is to alter the event position of all of the events recorded in the slice by the increment or decrement amount. This is done *per* the rules for EventPosition increment and decrement. An implementation may choose to perform the alteration at the time of request, or may defer until the slice is used further. A deferral approach enables the implementer to avoid iterating over the events until needed. However, care must be taken to ensure that the EventPosition increment and decrement rules are correctly followed. This does not apply to the operator+() and operator-() methods which must create a new copy of the data to be returned. It should be noted that one application of these operators is to make a copy of the slice data. This is achieved by adding nothing to a slice and saving the result.

A slice has an iterator subclass to provide iteration over the slice events. It is defined as follows.

```
class SliceRef::Iterator {
public:
   Iterator(const SliceRef&);
   Iterator(const Iterator&);
  ~Iterator();
   void            initial();
   bool            done() const;
   const EventID   nextEventID();
   const EventRef& nextEvent();
   void            operator= (const Iterator&);
private:
   void            operator&();
   void*           operator new(size_t);
};
```

As with the SliceRef class, it is expected to be small and is therefore constrained to being automatic, rather than heap allocated. Likewise, it must be tied-in to the SliceRef memory-management scheme. The assignment requires, as with the SliceRef class, a pointer and local-state copy, and not a copy of the slice. The methods are essentially as might be expected for an iterator. The initial() method will reset the local state to enable re-iteration over the events of the slice. Three points should be observed here. First, the iterator does not guarantee an ordering of the events. As such, there is no guarantee that invoking the initial() method will result in a repetition of the events in the same sequence. Second, the slice itself may be altered during the course of an iteration. This will not, in general, be caused by the partial-order data structure library code. Rather, nothing is included in this interface to require concurrency control of any variety on the slice, and thus the client code may change it. While addressing concurrency-control issues in the data structure is beyond the scope of this dissertation, it is expected that any implementation of this structure in a multi-threaded environment will provide serializability at the method-invocation level. Third, on first use the initial() method is not needed. It is only required if, after iterating, the client code needs to re-iterate over all events in the slice.

The done() method returns true when all events in the slice, if any, have been returned through invocations to nextEvent or nextEventID since the initialization or the last invocation of initial() occurred. The nextEvent() and nextEventID() methods return the next event or its identifier respectively. Note that these are not separate iterators. Both will advance the same local-state value that is recording the current slice event. The value in having nextEventID() is that it does not require a lookup in the partial order, where nextEvent() does. If the EventID is all that is required, then this will be faster. On the other hand, if the actual event is required, the nextEvent() method will be faster as it does not require the creation and destruction of an intermediate EventID. After all events, if any, in the slice have been returned and nextEventID() is invoked, the EventID returned will be the NULL EventID. Likewise, if nextEvent() is invoked under the same circumstances, the EventRef returned will be one with a NULL EventID.

The CutRef class, as we have noted, provides a closure of the SliceRef class. It therefore maintains essentially the same information, though it provides a different interface to that information. For essentially the same reasons, it must provide the same type of memory-management capability as does the SliceRef class. We will not specify the memory-management aspects here, though, partly to avoid redundancy, but also because there is more flexibility in achieving it. A simple approach that an implementer might choose is to have a single SliceRef object as member data. In such an approach a CutRef object would have the same size as a SliceRef object, *viz* the size of a pointer.

The formal specification of the CutRef class interface is as follows.

```
class CutRef {
public:
   CutRef();
   CutRef(const CutRef&);
   CutRef(const SliceRef&);
  ~CutRef();
   bool          contains(const EventID&) const;
   bool          consistent() const;
   bool          operator==(const CutRef&) const;
   bool          operator!=(const CutRef&) const;
   bool          operator< (const CutRef&) const;
   bool          operator<=(const CutRef&) const;
   bool          operator> (const CutRef&) const;
   bool          operator>=(const CutRef&) const;
   const CutRef& operator= (const CutRef&);
   const CutRef& operator= (const SliceRef&);
   const CutRef& operator& (const CutRef&) const;
   const CutRef& operator&=(const CutRef&);
   const CutRef& operator| (const CutRef&) const;
   const CutRef& operator|=(const CutRef&);
};
```

The methods fall into two broad categories: containment testing and set alteration. Containment-testing methods consist of the usual subset ($<$), superset ($>$), *etc.* methods, as well as element containment. In addition, the CutRef class provides a consistency check that determines if a cut is consistent. It should be noted that this consistency check is likely to be an expensive operation. The obvious algorithm is $O(N^2)$, where N is the number of traces. The set-alteration methods are assignment, intersection (`operator&`), and union (`operator|`). Set difference is not provided because the result is not a cut. As with the SliceRef class, the CutRef specifies a reseatable-reference assignment operator for CutRef assignment, and copy semantics (expected to be copy-on-write in an implementation) for assignment from a SliceRef object.

4.3 Events

The EventRef class provides methods for all of the properties that an event has. These can be broadly broken down into four categories: constructors, destructor and assignment, basic and extended event information including type information and arithmetic operators, comparison operators, and precedence-related event sets. In addition, for reasons we will describe shortly, the EventRef class must provide a memory-management capability in the same vein as the SliceRef and CutRef classes. Before describing these we first present the formal interface specification.

```
class EventRef {
public:
   EventRef();
   EventRef(const EventRef&);
   EventRef(const RawEvent&);
  ~EventRef();
   const EventRef& operator= (const EventRef&);

   class Type {
      public:
      enum Type {unary, transmit, receive, synchronous,
                 minimum, maximum, undefined};
   };
   Type            type()        const;
   bool            unary()       const;
   bool            transmit()    const;
   bool            receive()     const;
   bool            synchronous() const;
```

```
const EventID&       eventID() const;
const EventPosition& eventPosition() const;
const TraceID&       traceID() const;
SliceRef             eventIDs() const;
const EventID&       partnerID() const;
const EventRef&      partner() const;
SliceRef             partners() const;
SetRef               allPartners() const;
int                  partnerCount() const;
BufferPtr&           extended() const;

const EventRef& operator+ (const int) const;
const EventRef& operator- (const int) const;
const EventRef& operator+=(const int);
const EventRef& operator-=(const int);
const EventRef& operator++();
const EventRef& operator--();
const EventRef& operator++(int);
const EventRef& operator--(int);

bool            operator==(const int) const;
bool            operator!=(const int) const;
bool            operator==(const EventRef&) const;
bool            operator!=(const EventRef&) const;
bool            operator< (const EventRef&) const;
bool            operator<=(const EventRef&) const;
bool            operator> (const EventRef&) const;
bool            operator>=(const EventRef&) const;
bool            operator||(const EventRef&) const;

const CutRef    predecessorSet()       const;
const CutRef    successorSet()         const;
const SliceRef  greatestPredecessor()  const;
const SliceRef  leastSuccessor()       const;
const SliceRef  leastConcurrent()      const;
const SliceRef  greatestConcurrent()   const;

bool            covered()              const;
const SliceRef  covers()               const;
const SliceRef  coveredBy()            const;
```

```
   const EventRef& nonTraceCovers()       const;
   const EventRef& nonTraceCoveredBy()    const;
};
```

Before describing these methods we first comment on the memory-management requirement. There are a variety of reasons for specifying this. As with slices and cuts, events may consume a non-trivial amount of memory (indeed, this is the core scalability issue that this dissertation addresses). As such, copying is a poor choice. However, unlike slices and cuts, events are an integral component of the partial-order data structure. It might therefore be reasonable to use pointers to const or const references to specific event objects within the data structure. The reason we choose against this option is that it limits flexibility in the implementation of the data structure. Once a pointer or reference is returned to client code the object that is pointed or referred to can never be moved or deleted. This substantially constrains an implementation, especially as the data-structure size grows beyond main memory. We therefore have chosen to specify the same reseatable-reference technique that we adopted for the SliceRef and CutRef classes.

The constructors and destructor for the EventRef class are specified as follows. The default constructor will specify an event with a NULL EventID. The only legal operations that may be performed on such an event are assignment and equality testing. Other method invocations will result in an exception being thrown. The copy constructor will seat the reference to the input parameter, ensuring that the memory-management scheme is appropriately updated. The destructor likewise ties in with the memory-management scheme. The RawEvent constructor is required to specify an appropriate interface to the EventRef class for raw event data according to the specific tool for which it is implemented. The various restrictions imposed on this RawEvent structure will be addressed when we present the formal partial-order storage method in Section 4.4. Finally, assignment will reseat the reference in the same fashion as SliceRef assignment.

The various event-information methods are as follows. The type-information methods behave essentially as expected. The two exceptions are the minimum and maximum types, which are those types assigned to that event that precedes or succeeds, respectively, all current events on the trace. The undefined type is used wherever the EventID is NULL. The basic event information is somewhat more complex. *Per* Definitions 11 and 12, every non-synchronous event has exactly one position, while synchronous events can have more than one position, but at most one per trace. We therefore define the method eventID() as returning the unique EventID object for this event if it is non-synchronous and any one of the multiple EventID objects if it is synchronous. All of the event IDs for a synchronous event will refer to the same event, though they will not be equal under EventID equality testing. For consistency in the case of a synchronous event, the EventID returned by the method should be the same one for any invocation for that event. We will refer to that EventID as the canonical EventID for that synchronous event. The method eventIDs() returns a slice containing the event positions for each trace in which the event has a position. It thus corresponds directly to the function $\varphi(e)$. Since the EventPosition and TraceID are accessible through the EventID, for coding simplicity we provide those methods directly in the EventRef class.

Partner information for transmit and receive events is returned by the various partner methods. In the general case a system can have multicast and multi-receive events. In such a case there is no guarantee that there will be at most one partner event per trace. Indeed, the example of a multi-receive operation shown in Figure 2.1(a) has three transmit partners in one trace for the receive event. For this purpose we define the allPartners() method which returns a SetRef of all partners, where the SetRef class is any reasonable set class, such as that provided by the standard template library. For unary events the set returned will be empty. While synchronous events, like unary events, have no partners, it is sometimes convenient to treat the various EventIDs other than the canonical EventID as partners. We thus define this as the expected behaviour of this and the subsequent partner methods.

Normally all partners are not required. Rather, the greatest preceding or least succeeding partner in a given trace is all that is required, since the other partners are superfluous with respect to precedence issues. For this slightly restricted case we define the partners() method which returns a SliceRef of the greatest preceding (for receive events) or least succeeding (for transmit events) partner for each trace, if any. This method is sufficient if the system being monitored does not have multi-receive, and multicast reception is limited to one per trace. This does not appear to be a significant restriction in practice.

A more significant restriction, though a common one in existing observation tools, is a limitation of events to one partner, with the application of the appropriate transformations of Section 2.3. This limitation is imposed on transmit, receive, and synchronous events. For this restricted case, we define the partner() and partnerID methods that returns the EventRef and EventID, respectively, of the partner event. These methods will throw an exception if invoked by an event with more than one partner. For unary events and transmit events for which the corresponding receive event has not yet been stored in the data structure, partnerID() returns the NULL EventID while partner() returns the an EventRef with the NULL EventID. A partnerCount() method is provided, in part to prevent such invocations. This method returns the current number of partners for the event.

The extended() method simply returns a pointer to an uninterpreted buffer of event information. This information will be tool-specific, and so little more can be said about it. The one observation that we make is that a BufferPtr class must be defined that allows flexibility with respect to the physical location of the buffer of extended information. The standard smart-pointers techniques of C++ can be applied here.

For the sake of convenience, the various arithmetic operators provided by EventID are also provided with EventRef. There are three significant differences. First, the EventRef operators return EventRef objects, and thus require a retrieval operation on the partial-order data structure. If only the EventID is required, then such operations should be performed on the EventID, not on the EventRef. Second, decrements prior to the first event in a trace will result in an EventRef being returned that has the EventPosition prior to the first event. Such an event has type minimum. Third, if the result of an arithmetic operation exceeds the current maximum event recorded for the given trace, then the EventRef returned has that EventPosition that succeeds the current maximum event in the trace. Such an event is defined initially as type maximum, though it is subject to change at such time, if ever, as a new maximum event is stored in that trace. Due to the nature of the EventRef class, the data for the new event would be immediately accessible through this retrieved event.

The comparison operators provide the expected functionality, *per* the definitions of $\prec_{\mathcal{E}}$, $\preceq_{\mathcal{E}}$, and $\|_{\mathcal{E}}$ given in Chapter 2. The only significant observation that should be made is that all EventIDs of a synchronous event will be equal under EventRef equality testing, since all such EventIDs will retrieve the same EventRef. Equality testing with an integer is only meaningful if the integer is 0, and is used for testing if an EventRef has a NULL EventID.

The precedence-related event sets are defined as follows. The predecessorSet() method returns a cut of all events causally prior to this event, using $\prec_{\mathcal{E}}$. We would like the successorSet() method to return a similar cut of all events that causally succeed this event, again according to $\prec_{\mathcal{E}}$. Unfortunately, this is not a cut, since a cut is closed under $\prec_p$. However, we can return a cut of all events that are causally prior or concurrent to the given event. The successors are then all events not in the returned cut. This is then the cut that is returned by the successorSet() method. The greatestPredecessor() and leastSuccessor() methods provide the slices corresponding to these cuts, respectively. Thus,

$$\mathrm{greatestPredecessor}_{\mathcal{E}}(e) = \left\{ e_p^j \mid e_p^j \prec_{\mathcal{E}} e \wedge \nexists_{e_p^k} \left(e_p^j \prec_p e_p^k \ \wedge \ e_p^k \prec_{\mathcal{E}} e \right) \right\} \tag{4.2}$$

and

$$\mathrm{leastSuccessor}_{\mathcal{E}}(e) = \left\{ e_p^j \mid e \prec_{\mathcal{E}} e_p^j \wedge \nexists_{e_p^k} \left(e_p^k \prec_p e_p^j \ \wedge \ e \prec_{\mathcal{E}} e_p^k \right) \right\} \tag{4.3}$$

The leastConcurrent() method provides the slice of those events that are concurrent to the given event such that there is no event causally prior that is also concurrent.

$$\mathrm{leastConcurrent}_{\mathcal{E}}(e) = \left\{ e^j \mid e^j \|_{\mathcal{E}} e \wedge \nexists_{e^k} \left(e^k \|_{\mathcal{E}} e \ \wedge \ e^k \prec_{\mathcal{E}} e^j \right) \right\} \tag{4.4}$$

Likewise, the greatestConcurrent() method provides the slice of those events that are concurrent to the given event such that there is no causal successor that is also concurrent.

$$\mathrm{greatestConcurrent}_{\mathcal{E}}(e) = \left\{ e^j \mid e^j \|_{\mathcal{E}} e \wedge \nexists_{e^k} \left(e^k \|_{\mathcal{E}} e \ \wedge \ e^j \prec_{\mathcal{E}} e^k \right) \right\} \tag{4.5}$$

The least- and greatestConcurrent sets were defined by us for the computation of critical pairs (see Section 8.1.1).

Finally, there are five methods that pertain to event coverage. First, the boolean method covered() returns true when all events that will cover an event have been stored in the data structure. For a unary or a receive event, this occurs when the trace successor is stored. For a synchronous event, all trace successors must have been stored. For a transmit event, in addition to the trace successor, all matching receive events must have been stored. The covers() method returns the slice of all events that this event covers. Likewise, the coveredBy() method returns the slice of all events that cover this event. Note that covers() is expected to be complete on any invocation, where coveredBy() may be incomplete because the relevant events may not yet have been stored in the data structure. The nonTraceCovers() and nonTraceCoveredBy() methods provided the same functionality, though for the restricted case of single partner events. As with the partner() method, if more than a single event is covered by either of these methods an exception will be thrown. The coverage methods are almost synonyms for the partner methods, though they satisfy the formal definitions of **covers** of partial-order theory.

4.4 The Partial Order

The PartialOrder class provides event storage and retrieval, state information, iterators, and callback methods. It is defined as follows.

```
class PartialOrder {
public:
   PartialOrder();
  ~PartialOrder();

   const EventRef& store     (const RawEvent&);
   int             remove    (const int);
   float           remove    (const float);
   int             remove    (const CutRef&);
   int             remove    (const EventID&);
   const EventRef& operator()(const EventID&) const;
   const EventRef& operator()(const TraceID&,
                              const EventPosition&) const;

   int             traceCount()                  const
   int             totalTraceCount()             const
   int             eventCount()                  const
   int             totalEventCount()             const
   int             rawEventCount()               const
   int             totalRawEventCount()          const
   int             totalRawStorageCount()        const

   const EventID   minEventID   (const TraceID&) const;
   const EventRef& minEvent     (const TraceID&) const;
   const SliceRef& minEvents()                   const;

   const EventID   maxEventID   (const TraceID&) const;
   const EventID   maxRawEventID(const TraceID&) const;
   const EventRef& maxEvent     (const TraceID&) const;
   const EventRef& maxRawEvent  (const TraceID&) const;
   const SliceRef& maxEvents()                   const;
const SliceRef& maxRawEvents()                const;

class iType {
public:
   enum iType {allTraces, activeTraces,
               linear, traceOrder, events, rawEvents};
   };
};
class Iterator {
   Iterator(const iType::iType, const PartialOrder&);
   Iterator(const Iterator&);
  ~Iterator();
   void            initial();
   bool            done() const;
   const TraceID   nextTrace();
   const EventRef& nextEvent();
   void            operator= (const Iterator&);
};

Iterator& iterator(const iType::iType,
                   const PartialOrder&) const;

class Callback {
public:
   virtual      ~Callback();
   virtual bool  callback(const TraceID&);
   virtual bool  callback(const EventRef&);
   virtual bool  callback(...);
};

class Parameters {
   friend PartialOrder;
public:
   Parameters(PartialOrder&);
   Parameters& event(const EventID&);
   Parameters& trace(const TraceID&);
   Parameters& cut(const CutRef&);
   Parameters& slice(const SliceRef&);
   Parameters& eventType(const EventRef::Type::eType);
   Parameters& iterator(const iType::iType);
   Parameters& eventCount(const int);
```

```
      Parameters& spaceConsumption(const int);
      Parameters& spaceConsumption(const float);
      bool        activate();
   };

   class cbType {
   public:
      enum cbType {firstEvent, newTrace, iterator,
                   store, rawStore, remove, retrieve,
                   newPartner, newPartnerStored, covered,
                   eventCount, spaceConsumption,
                   structuralPattern, predicate};
   };

   Parameters& registerCallback  (cbType::cbType, Callback&);
   Parameters& deregisterCallback(cbType::cbType, Callback&);
   Parameters& enableCallback    (cbType::cbType, Callback&);
   Parameters& disableCallback   (cbType::cbType, Callback&);
};
```

The store() method takes raw events and incorporates them into the partial order. It imposes the following limitations. First, events do not have to be stored in a linearization of the partial order. However, any event that is stored that has not had all the events it covers stored will not, in the initial instance, be incorporated into the partial order as far as precedence is concerned. That is, while a retrieval will find the event, most of the comparison and all of the precedence-related methods of the EventRef class will throw an exception if invoked on this event. The comparison operators that will still work correctly are equality tests and comparison to events within the same trace. The event may be considered to have a local, but not a global, timestamp (see the discussion of timestamps in Chapter 6).

The second limitation pertains to synchronous events. While the partial order treats them as single, multi-homed events, they will not typically be received that way as raw data. Rather, several events will be received by the monitoring entity from different traces, each identifying itself as a synchronous event, and giving some partner information. Two requirements are imposed on this partner information. First, the closure of it must encompass all of the event's homes. Second, there must be some clear way of identifying when event information has been received for all of the homes. That is, the cardinality of $\varphi(s)$ must be computable. In the case of systems restricted to one partner, the cardinality is two or less. Note that until all of this information is received, the synchronous event cannot be fully incorporated into the partial-order data structure. As in the issue of events received out of linear order, some operations will result in exceptions being thrown. The partial failure of some component of the distributed system or of the monitoring code may result in the data structure eventually no longer being able to incorporate new events. This issue is beyond the scope of this dissertation.

Event removal is facilitated by the four remove() methods. The purpose of event removal is to deal with the fact that any monitoring entity cannot accumulate information indefinitely. The system will run out of physical storage. From time-to-time, which should be user-selected, the system must remove old information. For a partial order, events removed should form a consistent cut. The reason is that this will preserve the property that if an event is present in the data structure, its successors, if any, will be present. If all of the events for a given trace are removed, that trace is referred to as inactive. Traces with events are active.

The cut to be removed can be system or user specified. For a system-specified cut, remove() is invoked with either an integer, specifying approximately how many events should be removed, or a float, specify the percentage of events that should be removed. The returned value is the actual number or percentage of events removed, respectively. For a user-specified cut, the user can either specify the cut directly or specify an event to be removed. If a cut is specified, it will be checked for consistency. If it is not consistent, no events will be removed. If an event is specified, it and all of its predecessors will be removed. Each of these removal methods will return a count of the number of events removed.

Event retrieval is through operator(). For convenience, it allows retrieval by either EventID or TraceID and EventPosition. If the event sought was at one time present, but has been removed, then the EventRef of type minimum and with EventPosition(0) for that trace will be returned. We do not provide an analogous maximum for event retrieval that exceeds the current maximum on the trace, as new event storage will invoke the store callback. If the event sought is not present and the previous condition does not apply, then an EventRef with the NULL EventID will be returned.

Various state-information methods provide trace- and event-count information, and identify the least and greatest current event by trace. The trace count is a count of all active traces. The total-trace count includes all traces that have ever had events. The event count is a count of all events that are currently fully incorporated into the partial order. The total event count is a count of all events that are or ever were fully incorporated into the partial order. The raw-event and total-raw-event counts are similar, but they include all events that have been stored, whether or not they have been fully incorporated. Thus, they includes incomplete synchronous events and events received out of linear order. The total-raw-storage count is a strict counting of the number of store() invocations that did not result in an exception being thrown.

The least event by trace may be retrieved either as an EventID or an EventRef. A slice of all least events can likewise be retrieved. If the trace is inactive then the EventID will have EventPosition(0) and, if an EventRef was requested, type minimum.

The greatest event by trace may be retrieved either as an EventID or an EventRef, and may be either the greatest raw event or the greatest fully-incorporated event. A slice of all greatest events can likewise be retrieved, again either the greatest raw or fully-incorporated event. If the trace is inactive then the EventID will have that EventPosition that is one greater than was the maximum prior to its removal. If an EventRef was requested it will have type maximum.

Iterators are provided through the iterator() method. The parameter iType::iType determines the types of iterator. Six types are currently provided. First, allTraces, provides iteration over all traces, whether or not they are active. The traces are accessed through invocation of the nextTrace() method. The activeTraces iterator is similar, but provides iteration only over active traces. The invocation of nextEvent() on either of these iterators will cause an exception to be thrown.

Iteration over events is provided by linear, traceOrder, events, and rawEvents. The linear iterator provides the events in a linearization of the partial order, and is only over events that have been fully incorporated into the partial order. Different linear iterators will not necessarily produce the same linearization. Events are provided sequentially by trace using the traceOrder iterator, again only over events that have been fully incorporated. Different traceOrder iterators will not necessarily provide the traces in the same sequence, though the events on the trace will for correctness be in the same sequence. The events iterator is likewise only over events that have been fully incorporated, but does not guarantee that they are provided in a linearization or in trace order. It is probable that the events will not be in a linearization, as that is not likely the most efficient method for providing all events. Finally, the rawEvents iterator is an iterator over all events that have not been fully incorporated. It, together with the events iterator, provides iteration over all stored events. Access to the events is provided by the nextEvent() method. The invocation of nextTrace() on any of these event iterators will cause an exception to be thrown.

The various other methods, notably initial() and done(), perform the same function as those of the slice iterator. As with the slice iterator, none of these iterator guarantees a particular view of partial order that does not change during the invocation of the iterator. In particular, events and traces may be added or removed from the partial order at any time.

The callback methods are dependent on two subclasses, Callback and Parameters. The Callback subclass must be inherited by objects that wish to register as callback objects. It defines the methods that may be invoked for various callbacks. The variable argument callback method enables arbitrary parameters to be passed in the callback. However, most callbacks will only need to pass either an EventRef or a TraceID, and as such those methods have been defined to improve efficiency and enable lightweight callbacks. The callback methods return a boolean indicating success or failure. The PartialOrder class will disable the relevant object from being further invoked if boolean failure is returned. A virtual destructor is required to ensure correct disposal.

For Callback object registration a Parameters class is defined. This class enables the passing of parameters to the callback triggers using the named-parameter idiom described in Question 10.15 of the C++FAQ Lite [23]. This is best illustrated by example. To register a callback for every synchronous event that is stored on a particular trace, we would write the following.

```
partialOrder.registerCallback(PartialOrder::cbType::store,
                              callbackObject).
             eventType(EventRef::Type::synchronous).
             trace(traceID);
```

where `callbackObject` is the callback object and `traceID` is the TraceID of the trace for which we wish callbacks. Default values are set for the various parameters as follows. The

EventID will be set to the NULL EventID. The TraceID will be set to the non-trace TraceID. The SliceRef and CutRef will be empty. The EventRef type is undefined. The iterator type will be linear. The event count will be -1, as will the space consumption integer and float parameters. The PartialOrder, required by the Parameters constructor, will be set by the PartialOrder register() method to that PartialOrder for which the register() method was invoked. The PartialOrder is needed for the activate() method.

The activate() method can be optionally used to force immediate activation of the callback, and to provide feedback on the success or failure of the specific callback method. It must be the last method of the method chain. The primary cause of failure would be to deregister a callback that was not present, enable one that was not present or not disabled, or disable one that was not present or not enabled. The fairly minor nature of these failures is the reason failure feedback is not typically needed. If activate() is not invoked, the callback method will not be activated until the next invocation of a PartialOrder method. For most callbacks this will be the first occurrence at which the callback needs to be activated. However, for iterator callbacks, early activation may be desirable.

The specific set of parameters is not yet fully defined, and is likely to be implementation-specific. In particular, predicate and structural-pattern callbacks do not have parameters defined. Predicate callbacks would likely require the examination of the extended event information which, as we have already noted, is system specific. It is probable that the correct approach to this is to enable the creation of user-defined functions that operate on this extended data, and have those functions passed as parameters. Structural-pattern callbacks will have parameters that, in part, depend on the pattern language chosen. These issues are beyond the scope of this dissertation.

There are four callback methods: register, deregister, enable, and disable. They perform essentially the functions as might be expected by their names, though it should be noted that a callback that is registered will be enabled until it is disabled, either explicitly or by the data structure. Every Callback object that is registered and enabled will cause a small space and performance penalty on the data structure. As such, a Callback object should be disabled if it is not needed for some time, and not going to be invoked for that period. Likewise, a Callback object that is registered but disabled should be deregistered if it is no longer needed, as it will cause a small space penalty and likely a small performance penalty when new Callback objects are registered.

There are fourteen callback types, many of which have at least optional parameters. We describe them in turn. The firstEvent callback has one optional parameter, eventType. If the parameter is not set, the callback will be invoked on the full incorporation of the first event of every new trace into the partial order. If it is set, it will only be invoked if the first event on the trace is of the specified type. The callback method invoked is callback(EventRef&).

The newTrace callback is similar, though parameterless, being invoked on the creation of a new trace for the storage of a new event. Note that the event stored that causes the creation of the new trace may not be fully incorporated into the partial order. The callback method invoked is callback(TraceID&).

The iterator callback provides a callback version of the iterators. If the optional iType::iType parameter is not altered, it will default to a linear iterator. The callback will commence at such time as the PartialOrder is subsequently invoked. For immediate activation, the activate() Parameter method should be included. Thus

```
partialOrder.registerCallback(PartialOrder::iterator,
                              callbackObject).
             iterator(PartialOrder::activeTraces).
             activate();
```

will cause an activeTraces iterator to be set up that repeatedly invokes the callback(TraceID&) method of the callbackObject with the results of nextTrace() until there are no further active traces. An iterator callback will iterate through its sequence at most once, after which it will deregister itself.

The store callback optionally takes EventRef::Type::eType, TraceID, EventID, SliceRef, and CutRef parameters. If no parameter is passed, it will invoke the callback whenever an event is fully incorporated into the partial order. The callback method invoked is callback(EventRef&). If the eventType is specified, it will only be invoked for the specified event type. Note that specifying EventRef::Type::minimum or maximum will never match on a store, and will result in the callback automatically being deregistered. In all of the remaining instances, if the eventType is specified then the invocation will only occur if the event matches the specified event type. If the TraceID parameter is set, the callback will be invoked whenever an event on that trace is fully incorporated into the partial order. Likewise, if the EventID parameter is set, the callback will be invoked when the event corresponding to that EventID is fully incorporated into the partial order. Since that callback can only occur once, after it has happened the callback will automatically be deregistered. If the SliceRef or CutRef parameters are set, the callback will be invoked when any event in the slice or cut is fully incorporated into the partial order. If more than one parameter is set then the behaviour will be the union of the behaviours, with the exclusion of the automatic deregistration.

The rawStoreEvent callback is identical to the store callback, except that it is invoked immediately after the store() method has completed, whether or not that partner event is fully incorporated into the partial order. This is created as a separate callback rather than a parameter to the store callback because it is invoked at a different time from the store callback.

The remove callback behaves in a similar manner to the store callback, though it is invoked on the occurrence of event removal. If no parameters are set it will invoke callback() when one or more events are removed by the remove() method. If the eventType is specified, it will only be invoked if an event of the given event type is removed. This remains true for all of the remaining instances. If the TraceID is set it will invoke callback(TraceID&) when one or more events are to be removed from the given trace. If the EventID is set it will invoke callback(EventRef&) just prior to the removal of that event. Likewise, if the SliceRef is set it will invoke callback(EventRef&) for any event in that slice just prior to the removal of that event. If the CutRef is set it will invoke callback(CutRef&) when one or more events within the cut are to be removed. In all cases the removal operation will occur after the callback.

The retrieve callback is invoked on the occurrence of event retrieval. As with store, it can be parameterless, or take an EventRef::Type::eType, EventID, TraceID, SliceRef, and/or CutRef. With parameters it will invoke the callback only if the event retrieved is an element of one of the parameters, matching the appropriate type. A possible use would be to keep track of which events have EventRefs passed to client code. The callback method invoked is callback(EventRef&).

The newPartner callback is invoked whenever an event has a new partner fully incorporated into the partial order. It takes the same optional parameters as the store callback, and treats them in essentially the same manner. Note that the parameters specify details about the event whose partners are sought, not details about the partners. If the eventType specified is unary, the callback will automatically be deregistered.

The newPartnerStored callback is identical to the newPartner callback, except that it is invoked immediately after the store() method has completed for the new partner event, whether or not that partner event is fully incorporated into the partial order. The covered callback is similar, being invoked when an event (possibly partially specified by the parameters passed) is fully covered by events that are fully incorporated into the partial order.

The eventCount callback requires a single integer parameter, eventCount. When the number of events fully incorporated into the partial order reaches that count the callback method callback(count) will be invoked. The callback will remain registered and enabled as event removal may cause the callback to be invoked subsequently. However, if the user knows that this will not be the case, then the callback should be deregistered. A simple rollover technique would be to register eventCount callback, with the maximum number of desirable events, and have that Callback object invoke remove() with the desired reduction in events.

The spaceConsumption callback is similar in intent, but operates on physical memory consumption, either bytes (given the integer parameter) or percentage of system physical memory (given the float parameter). At such time, if ever, that the space consumption is estimated to exceed that specified by the spaceConsumption threshold the callback() method is invoked with either an integer or float parameter that specifies what threshold was exceeded.

The remaining two callbacks, structuralPattern and predicate, are beyond the scope of this dissertation to fully specify. Their intent is enable a callback on the occurrence of a pattern match, either of the predicate or structural variety. We hope to explore mechanisms to incorporate these into the partial order in future work.

Finally, we observe that multiple callbacks caused by a given trigger do not have a clear order of invocation. Further, the order in which two callback method are invoked is not guaranteed to remain the same the next time they are triggered simultaneously. In a multi-threaded environment, they may probably be invoked concurrently in separate threads. Deadlock prevention is not guaranteed if a callback object invokes a method on the partial order, though if the method does not alter the partial order, it should not cause a deadlock.

4.5 SATISFYING THE REQUIREMENTS

We now demonstrate how the data structure we have defined satisfies the requirements as enumerated in Section 3.11. For many of those requirements, the solution in the interface is either explicit or obvious, and we will not belabour the point. A few requirements, however, appear to be absent, and so we indicate how this interface makes provision for them.

The most clearly missing data is the real-time and line-of-code information. The reason for its absence is that it is placed within the extended buffer. The justification for this is as follows. First, such information is not relevant to the maintenance of the data structure, and therefore is not strictly essential. Further, it is not relevant for all applications, and polluting the basic information with non-essential requirements will only add space-consumption costs to those who do not need it. This can be considered a variant of the end-to-end argument [129]. Second, the treatment of such information will be application-specific. For example, if we wish to compute the longest chain using the real time as the edge weight, it is not clear which real-time would be used for synchronous events that have more than one real-time associated with their operation. Likewise, a breakpoint on a synchronous event could not know which line of code to display. However, the event could trigger an application-specific callback that would know. Third, the neither real-time nor line-of-code information is fixed-size. In the case of the latter, it is clear that it requires file identification, and thus will be variable. In the case of real-time information, it is not fixed-size because synchronous events are not atomic in time.

Next, there are no (anti-)chain determination methods in the interface. This is deliberate because, as we noted, modeling computations as partial orders does not yield a one-to-one mapping between traces and processes or threads. Rather, a given process or thread is often represented in multiple traces. Chain and anti-chain computation must take into account what process or thread a given trace belongs to. This is application-specific. That said, the data structure does provide sufficient information for any client code to determine chains or anti-chains as required by its application.

Third, there is no explicit statistics information provided. This can be gathered instead by callback functions associated with the desired entity. Thus, event statistics for a given trace can be computed by a callback that is triggered as events are stored in that trace. Any desired level of statistical information can be collected, according to the needs of a given application.

Finally, we note that there are some elements of the interface whose existence is not predicated on the requirements. In particular, the least- and greatest-concurrent sets have no corresponding requirement. They are provided to enable one of our scalable-timestamp solutions.

PART II

CURRENT TECHNIQUES

5 EVENT STORAGE AND ACCESS

Having defined the formal requirements, we now explore both current solutions for satisfying those requirements and the application of existing data-structure techniques where solutions do not currently exist. The requirements can be broken down into event storage and access, and precedence determination. Precedence determination largely revolves around algorithms for implementing the comparison and precedence-related-event-set methods of the EventRef class. Event storage and access is that part of the structure that provides all facilities other than event-precedence operations. Thus it provides the algorithms and data structures for the implementation of the remaining methods of the EventRef class, and the methods of all other classes. In this chapter we will present solutions for event storage and access.

The three basic classes, EventPosition, TraceID, and EventID, together with many methods of the remaining classes, are sufficiently straightforward that little comment is needed. A couple of minor, but significant, details do need to be dealt with correctly.

First, the specification has identified EventPosition objects as being able to compare positions with unsigned integers. This requires an architectural limitation of the structure, which is dependent on the word size of the machine for which it is implemented. We believe this to be an acceptable tradeoff as infinite-precision integers would be undesirable for such a lightweight class. A 64-bit machine can operate at a high event-data rate (one billion events per trace per second) for more than a century. Since this is far in excess of current capacity, this is an acceptable implementation approach.

Second, EventPosition, TraceID, and EventID are lightweight classes. For example, the increment-by-one of EventPosition(0) will always result in EventPosition(1), whether or not that particular event is currently in the data structure (assuming the EventPosition in question is part of an EventID). However, the EventRef class does not ignore event removal. Thus, the increment of an EventRef object with EventPosition(0) will be dependent on which events, if any, have been removed from the trace.

We now deal with the non-trivial aspects of event storage and access, starting first with the way events are received from instrumented processes and integrated into the data structure.

5.1 RAW-EVENT PROCESSING

The storage of raw events requires four things:

1. trace identification
2. positional information within the trace
3. type information
4. partner information, if any.

In addition, event storage must deal correctly with the storage of non-maximal events. We now describe existing solutions to these issues.

Trace identification is usually made by the monitoring code and can be passed to the monitoring entity once, if communication is through a persistent channel, such as a TCP stream, or on a per-event basis, if communication is datagram based. We make no comment on the advantages or disadvantages of either technique, as it is outside the scope of this dissertation. Suffice it to say, trace identification is part of the RawEvent structure.

Positional and type information are likewise the responsibility of the monitoring code, and it is expected that these are part of the RawEvent structure. By way of example, in POET the EVENT struct contains integer members e_trace and e_evcnt, as well as EV_TYPE e_etype and short e_flag. The e_trace value uniquely identifies the trace. The e_evcnt likewise identifies the event's position within the trace, where the first event has position 0. The event's type may be determined from the e_type and e_flag together with the global event-description table, which is required by POET to maintain target-system independence. The type is inferred as follows

```
bool unary(EVENT& e) {
  return event_tab[e.e_etype].num_partner == 0;
}

bool transmit(EVENT& e) {
  return (e.e_flag & E_ASYNC &&
          event_tab[e.e_etype].num_partner < 0);
}

bool receive(EVENT& e) {
  return (e.e_flag & E_ASYNC &&
          event_tab[e.e_etype].num_partner > 0);
}

bool synchronous(EVENT& e) {
  return (!(e.e_flag & E_ASYNC) && !unary());
}
```

where `event_tab` is the event-description table, `E_ASYNC` is that bit in an e_flag that identifies an event as asynchronous, and `num_partners` identifies the number of partners. Within POET the number of partners is specified as -1, 0 or > 0, where -1 indicates that a partner is expected, but the instrumentation cannot be expected to know what the partner is at the point the event occurs. It is therefore used when instrumenting transmit events, which cannot know the trace identifiers or positions of the receiving events.

Partner information is somewhat more difficult to deal with. There are four instances that we distinguish: transmit-event partners, receive-event partners, synchronous events, and special cases for each of the previous three. We describe first the special cases, where the only current significant special case is when the number of partners is limited to one. This is the only case permitted by POET, Object-Level Trace, and the EMU/ATEMPT/PARASIT tool suite.

A unicast transmit event may know the trace to which it transmits, but it cannot know the EventPosition of the receiving event. Likewise, the receive event may know the trace of its corresponding transmit event, but cannot know the EventPosition. This information can be gained in several ways, though in practice only one technique is used. Each trace is required to keep track of its current EventPosition. When a message is transmitted, the current EventPosition of the transmitting trace is appended to the message. If the system is such that a receiver cannot know the transmitting trace, then the TraceID of that trace is also appended to the message. The receiver then records the transmitting EventID and forwards this, along with its own EventID, to the the monitoring entity. The stored transmit event then has no partner information recorded until its matching receive event is stored. At that time the transmit event is updated to include the partner information.

An alternate solution can be employed if the instrumented system is known to have FIFO channels and the sender can identify the receiving trace and *vice versa*. In this case the receiving and transmitting events identify in the RawEvent structure the partner trace. When the receive event is stored, it then seeks the first unmatched transmit event with the appropriate receiver specified on the identified partner trace. Likewise, the transmit event seeks the first unmatched receive event with the appropriate transmitter specified on the identified partner trace. The primary advantage of this technique is that it requires no alteration to the actual message transmitted. In certain instances, such alteration may not be possible.

Synchronous events are handled in a like manner to transmit/receive events in POET, though it is possible to perform matching within the monitoring code. The synchronous call will append its EventPosition to the message. The synchronous receive will then be able to record its partner's EventID, which it forwards, along with its own EventID, to the monitoring entity. A synchronous-call event is then partially stored, but precedence information will not be determinable until its partner is also stored.

The alternate solution is to take advantage of the fact that synchronous partners must all communicate with each other. It is therefore quite possible for the synchronous receive to append its EventPosition to its reply. The synchronous call is then able to record the partner information, and the storage of one of the synchronous partners would be strictly redundant, as the partial order treats a synchronous event as single, multi-homed event. The primary advantage of this solution is that it can more easily handle partial failure within the distributed system or monitoring code. Specifically, once the first synchronous event is recorded, the system will not deadlock if the partner is never received. The drawback is that it delays the availability of the synchronous-call event to the monitoring entity. More exploration of handling partial failure in such monitoring systems is required.

We now turn to the general case of multiple partners. For multicast transmit events, it is sufficient to append the EventPosition of the transmitting event to the message. It may also be desirable, insofar as it is possible, to record the number of receivers, so that the partial order can more easily determine when the transmit event is fully covered. If the partial order does not know this information, then it must assume that the event is not covered until such time, if ever, as the event has a successor in every trace. Receivers of such multicast messages must record the transmit EventID and forward it, along with their own EventID, to the monitoring entity for storage.

A multi-receive event is mildly more complex. If each corresponding transmit event appends its EventIDs, then the receiver must be instrumented to capture each of these EventIDs and forward them, along with its own, to the monitoring entity for storage. Note that the EventPosition may be insufficient, since it may not be possible to distinguish the trace for which each EventPosition corresponds.

For synchronous events that have more than two homes, the two-home alternate solution can be naturally extended by including all EventIDs of all constituent homes that are known at the time. Each RawEvent would then record the entire set of homes. The drawback with this is that an N-way synchronous event will now require messages of size $O(N)$, which is undesirable. More efficient extensions of this method are probably possible, though it would depend on the specific implementation of N-way synchronous events to determine exactly how they worked.

The POET solution would require a slightly more complex approach. One of the synchronous homes, we will identify it as the initiator, must identify the total number of homes, and forward this, along with its EventID, to the monitoring entity for storage. A transmitted message from the initiator, either directly or one that is forwarded, includes the EventID of the initiator. Thus a binary-tree implementation of a synchronous event would forward the initiator EventID along each edge of the tree. This EventID, along with the EventID of the receiver, is forwarded to the monitoring entity for storage. The store() method then matches the constituents as follows. Any synchronous-event store() either identifies the initiator and the total number of traces, or it identifies a receiver and the initiator. The store() is complete when all the homes, as specified by the initiator, have been stored and matched.

The coverage methods, being almost synonymous with the partner methods, are handled in essentially the same manner. As such, we do not comment further.

Finally, we must correctly deal with out-of-order storage. Some systems (*e.g.* the network monitoring system of Parulkar *et al* [116]) avoid this problem by using causal message ordering within the system, including the monitoring station. In this way events are never received, and hence stored, out of causal order. Such an approach is not practical because of the high overhead of ensuring causal message ordering.

A simpler approach, adopted by POET, is to maintain event priority queues, one per trace. Events within these queues are stored in EventRef format, and they may be sought using the event-retrieval mechanisms, per the specification of the previous chapter. When store() is invoked for an event, it is first determined if this is the next expected event for its trace. Note that this is not generally an issue, as raw events are typically forwarded to the monitoring entity in FIFO order. This is only required if events can be forwarded in non-FIFO order. In this instance, the processing of the event is delayed at least until its EventPosition is the next expected.

We now deal with events that are the next expected event within their trace. For unary and transmit events, full incorporation into the partial order takes place immediately, though the partner information for a transmit event may not yet be complete. This is possible because such events are maximal at the time of their storage. For receive and synchronous events, some delay may be

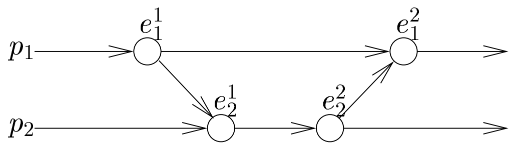


Figure 5.1: Not Quite Transitive Reduction

incurred. Specifically, a receive event cannot be fully incorporated until its corresponding transmit events have been incorporated. Its queue will therefore block, pending the storage of those transmit partners. Synchronous events will likewise block pending the arrival of information from each home (assuming that the POET approach is used for collecting the event information). Either of these cases will deadlock if the required information is not forthcoming, possibly because of partial failure within the system. The correct solution to such a deadlock is beyond the scope of this dissertation.

The specific implementation of the described storage approach can be single-threaded, though there is no reason why there cannot be up to one thread per queue for RawEvent addition to the queue, and up to one thread per queue to incorporate the queued events into the data structure. Neither approach is beyond the straightforward application of current techniques, and so we do not describe it further.

5.2 Event-Access Data Structure

We now address the core data structure within which events are stored. All systems we are aware of store, approximately, the transitive reduction of the partial order. The qualifier is required because they will continue to maintain the full sequential-trace information, which is sometimes slightly more than a transitive reduction. This can be seen in Figure 5.1, where the e_1^1 to e_1^2 edge is not in the transitive reduction, but will be stored implicitly. It is the storage of this approximate transitive reduction with which we are now concerned.

To determine the appropriate structure we must know something of the data that is being stored. The following observations can be made based on the POET experience. First, while in some target-environments traces contain similar numbers of events, in others the number of events varies by substantial orders of magnitude. Second, traces may appear or disappear (that is, cease to be active) at any time. Third, traces may disappear after many or very few events. Fourth, events are stored in strictly linear order within a trace. Finally, rollover happens.

Given this behaviour, event data for each trace is stored in a variant on the B-tree structure, and an extensible array is used to access each trace. Our only comment on the extensible array is to observe that it must contain pointers to the roots of each trace structure, and various information about that structure that is fixed on a per-trace basis.

The variant B-tree structure is illustrated in Figure 5.2. It maintains fixed-size blocks of pointers, with the bottom level containing the fixed-size portion of the event data and pointers to any variable-size component of event data. It is possible to have the bottom level point to individual events, rather than contain a block of events, though it is unclear if this provides an

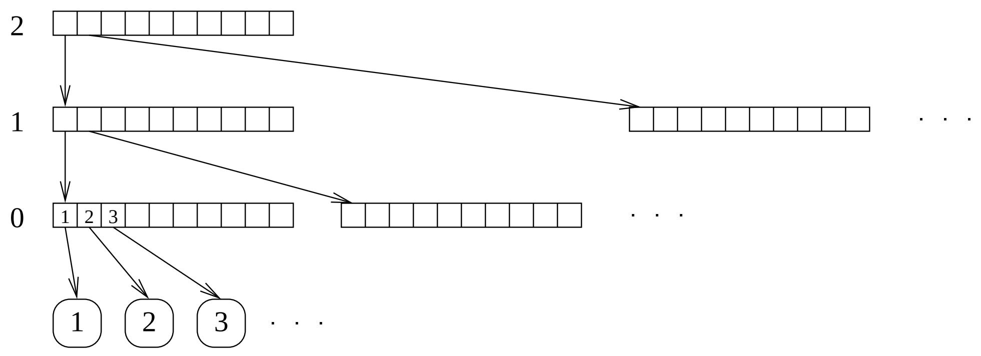


Figure 5.2: Trace Data Structure

advantage. The presumption behind our approach is that a significant fraction of event retrievals will only wish to access the fixed-size data. This reduces the number of dereference operations required by one, which is significant given that an expected blocksize of 256 will require only one or two dereference operations to access the data.

The differences from a B-tree structure are as follows. First, events are always appended, rather than inserted between other events. Second, events that are removed are always removed from the front, and never in the middle. As a result there is never a need to split or merge index blocks, as is the case with B-trees. However, as the number of events stored within a trace grows, it is possible that a new level is needed in the tree. Likewise, the removal of events may cause the tree to shrink by one or more levels.

The basic operations are retrieval, insertion, and deletion. Event retrieval starts at the root and recursively computes the pointer to follow, thus.

```
retrieve(position, block) {
  if (block.level > 0) {
    offset = pow(blocksize, block.level);
    retrieve(position % offset, block[position/offset]);
  }
  else
    return(block[position]);
}
```

Note that `position` is the position of the event within the leaves of the structure not the Event-Position. These positions are numbered from zero, where the zero position is not the position of the least event recorded, but the position of the least event recordable. That is, it is that position that would be reached if the least pointer in each block, at each level, were followed, whether or not that block is in fact still present. This is illustrated in Figure 5.3. Note that the event with EventPosition(1) is the least recordable event in that figure, even though the least recorded event is that with EventPosition(12). This value of `position` is determined by subtracting from the EventPosition the current value of the least recordable event. This subtraction is performed prior

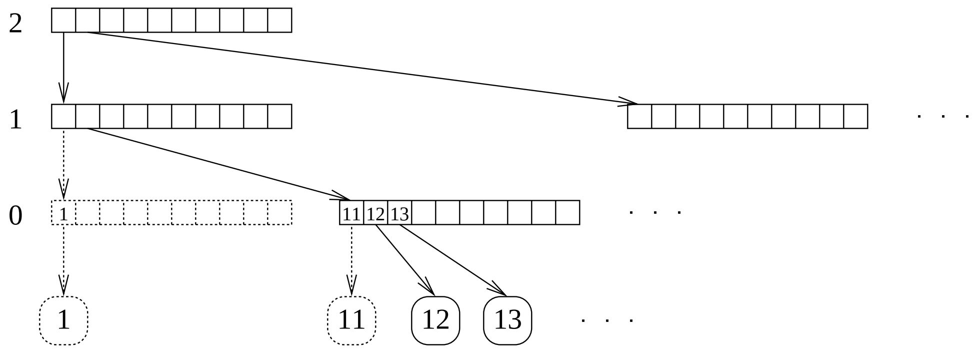


Figure 5.3: Least Recordable Event (Dotted Lines Indicate Deleted Data)

to the initial invocation of retrieve(). In the initial instance, the least recordable event is that with EventPosition(1). After deletions, this will likely change.

It is also apparent from the retrieve() code that it is desirable for the blocksize to be a power of two, allowing the division to be computed using a bitshift operation and the remainder by a bitmask operation. With a blocksize of 256 most current trace data will retrieve events with two lookups, and certainly within three.

Event insertion requires identifying the pointer to follow, while adding new pointer blocks as needed.

```
Block* insert(event, position, block) {
  if (block.level > 0) {
    offset = pow(blocksize, block.level);
    if (position >= offset) {
      Block* newBlock = new block[blocksize];
      newBlock[0] = block;
      newBlock.level = block.level + 1;
      return insert(event, position, newBlock);
    }
    if (block[position/offset] == 0) {
      block[position/offset] = new block[blocksize];
    }
    return
      insert(event, position % offset, block[position/offset]);
  }
  else {
    block[position] = event;
    return block;
  }
}
```

The algorithm returns a pointer to a block so that the originally invoking method can adjust its pointer to a new root block if one was created. At most $h + 1$ new pointer blocks will be created, where h is the initial height of the tree. A new pointer block at level l will only be needed after the insertion of b^l events, where b is the blocksize. It therefore only adds an amortized constant to the cost of insertion.

Event removal is somewhat more complex, as it may also cause the removal of levels in the tree and alter the value of the least recordable event. The functions of event removal and tree rebalancing are easily separable, and so we describe them as distinct operations, though this is probably not the most efficient approach in practice.

```
remove(position, block) {
  if (block.level > 0) {
    offset = pow(blocksize, block.level);
    for (i = 0 ; i < position/offset ; ++i)
      delete block[i];
    remove(position % offset, block[position/offset]);
  }
  else
    for (i = 0 ; i <= position ; ++i)
      delete block[i];
}
```

Note that the destructor for the block class, as invoked by delete, must delete anything that the block is currently pointing to. This is standard practice, but is also a common source of errors, and so we emphasize the point.

```
rebalance(block) {
  if (block.level > 0) {
    if(block[blocksize - 2] == 0 &&
       block[blocksize - 1] != 0) {
      root = block[blocksize - 1];
      leastRecordableEvent += pow(blocksize, block.level);
      block[blocksize - 1] = 0;
      delete block;
      rebalance(root);
    }
  }
}
```

First, observe that the test for whether or not a pointer block can be deleted is not strictly as shown. That test simply determines that there are events stored below the last pointer, but not below the penultimate pointer. It is not strictly guaranteed that this means there are no active pointers earlier in the block. However, it would require out-of-order event insertion within the trace itself, which is generally unlikely as monitoring code tends to forward RawEvent data in FIFO order. Further, it would require a gap in the insertion order of at least blocksize events, which is doubtful when the blocksize is 256. Second, note that we are implicitly assuming that the `root` and `leastRecordableEvent` variables are global. This is simply for convenience of description, and would not be the approach taken in the actual code. We leave it to the reader to verify the correctness of these algorithms.

We must now turn very briefly to the data structure required for the EventRef class. The requirements specified in the previous chapter lead naturally to a structure that consists of the following.

```
class EventRef {
...
private:
  class Data {
  public:
    Data();
    Data(EventBlock&);
    EventID           _eventID;
    Type              _type;
    bool              _covered;
    SliceRef          _partners;
    BufferPtr         _extended;
    unsigned int      _count;
    EventBlock* const _block;
  };
  Data* _event;

}
```

In the case where we are limited to a single partner, the `_partners` member data is replaced with `_partner` of type EventID. Note that this structure is fixed-size, as the `_partners` and `_extended` member data are single pointers. For the single-partner case the size of an EventRef together with the EventRef::Data will be approximately 9 words, or 36 bytes for a 32-bit machine. For the multi-partner case, the fixed size will be approximately 32 bytes, with additional variable space required for SliceRef data. The amount of space required for this will be dependent on the number of partners. Note also that the data stored will be augmented to enable precedence determination. This will require an additional pointer, and a variable amount of space according to the specific precedence technique adopted. We may therefore observe that the fixed-size storage requirement per event is approximately 44 bytes, allowing 4 bytes for the block-lookup pointer. Note that we could reduce this by 8 bytes per event by not recording the TraceID, since it is the same for every event on the trace, and the EventPosition, since it is known in order to perform the event lookup. However, this would require a more complex, and probably slower, retrieval process, since any EventRef returned to the client code must contain the EventID. Further, this space saving is negligible relative to the space required for precedence determination. Additional storage may be required for serialization in a multi-threaded environment, though this will depend on the degree of concurrency expected. If it is expected to be small, a lock on the associated EventBlock object may be sufficient. The use of the `_block` and `_count` member data will be discussed in Section 5.3.

Finally, we address the SliceRef structure. This is essentially an array of event positions, indexed by trace. However, the specific implementation is dependent on the number of traces for which event positions exist. For many uses, in particular when specifying the greatest predecessor and least successor of most events, a dense-array representation is probably the correct approach. On the other hand, there are cases where it is likely that the SliceRef data will be sparse. Specifically, partner information is unlikely to be dense, since most communication is unicast even where multicast is possible. For such cases, an associative array is preferred. Unfortunately, we know of no general data structure that adequately covers the entire range, though hashing techniques may be applicable. This issue requires further study.

5.3 Freestore Management

Freestore management is required at three levels within the data structure. First, as indicated by the requirements, the SliceRef, Slice::Iterator, and CutRef objects must be correctly disposed of after client code no longer references them. This also extends to EventRef objects which have been removed from the data structure, but for which client code still has references. Second, the data pointer and fixed-size component of EventRef objects must be placed in a specific location in the trace-tree structure. This complicates the reference-counting technique for these objects. Finally, various objects are likely to be created and deleted frequently, but are not automatics. For example, slices, slice iterators, and cuts returned to client code usually will have a short life, but are heap-allocated. In such instances the default new and delete operators are undesirable, as their general-purpose nature tends to impede performance.

As we have already indicated in Section 4.2, we solve the first of these problems by adopting a variant technique from the C++ FAQ Lite[23]. There are two differences from the method specified in that FAQ. First, we do not provide copy-on-write semantics. Rather, we reseat the reference. Thus methods that mutate the data are not altered, unlike the approach of that FAQ. Second, if the data structure exists in a multi-threaded environment, then serialization of the `_count` member is required. The reference-counting technique is otherwise unaltered.

The extension of this technique to EventRef objects is more complex. EventRef::Data objects are created within a block of the trace structure. As such they cannot simply be deleted using the described reference-counting technique. First, we show the structure of event blocks, and then we provide an extension to the reference-counting mechanism for such blocks.

A block of events, as required by the trace structure, can be created as a just a block of EventRef objects. The problem with this approach is that EventRef objects are fundamentally just pointers to EventRef::Data objects. Little advantage is accrued by placing these pointers together while allowing the associated EventRef::Data objects to be in essentially random locations in memory.

Alternately, we could create just a block of EventRef::Data objects, together with various block-level data, such as a count of how many of the objects are currently referenced. This would save the cost of a pointer per event stored. The problem with this approach is that the EventRef::Data objects are of no value in and of themselves. To be used they must be accessed though an EventRef object. While the various partial-order objects could evade this through friendship, this would probably be poor for data-structure maintenance, and would certainly not solve the problem for returning EventRef objects to client code. This second problem could be solved by returning copies of EventRef objects, rather than references, as specified in the requirements. However, as noted in the previous chapter, returning a copy requires the constructor and destructor to be invoked in quick succession. This is particularly poor in this circumstance where a count must be kept, possibly requiring a lock.

The third approach, and the one we adopt, is to keep both EventRef and EventRef::Data objects in block structures within the trace structure, returning a reference to the EventRef as required. The issue then becomes one of where the respective blocks of objects should be located. We could maintain two separate blocks, one for EventRef objects and one for EventRef::Data objects. However, this is probably not wise. Whenever an EventRef object is accessed, its associated EventRef::Data object will be accessed immediately afterward. It therefore makes sense from a cache-performance perspective to locate the EventRef object immediately prior in memory to the EventRef::Data object to which it refers. While EventRef objects in client code cannot be so located, they can be for the blocks of events stored in the trace structure. We therefore define the EventBlock class as follows.

```
class EventBlock {
  friend EventRef::Data;
public:
  EventBlock() : _count(1),
                 _events(this) {};
 ˜EventBlock() {
    if (--_count <= 0)
      delete this;
  }
  const EventRef& operator(const int i) const {
    return _events[i]._er;
  };
  ...
private:
  class ER {
  public:
    ER(EventBlock* b) : _erd(b), _er(_erd) {};
    EventRef       _er;
    EventRef::Data _erd;
  };
  ...
  unsigned int _count;
  ER           _events[blocksize];
};
```

Note that this creates a block with a sequence of EventRef, EventRef::Data objects, where the `_data` pointer of the EventRef object will point to the EventRef::Data object that immediately follows it. The EventRef::Data will be set to default values, with the exception of the `_block` member, which will be set to point to the EventBlock in which the EventRef and EventRef::Data objects reside. Note that the line of code that achieves this is not actually syntactically correct C++ code. There is no syntactically correct way to invoke any constructor other than the default on array member data. The effect of this can be achieved by other, more complex, means, however, and so we leave it as it is.

We then perform freestore management on this structure as follows. The `_count` member data in the EventBlock class contains a sum of two separate numbers. The first number is an indication of the liveness of the EventBlock object as far as the trace-tree structure is concerned. Initially this will be one, since the EventBlock object has been created, presumably to be used to store data. At such time, if ever, that the EventBlock object is to be removed, this number will be decremented to zero. Thus the value of `_count` will be decremented by one. This will only cause the deletion of the EventBlock if the resulting value of `_count` is then zero. If not, it will defer the deletion until `_count` does reach zero.

The second component of the `_count` data is a count of how many of the contained events have references outstanding in the client code. Note that it is not a count of the number of outstanding references, either to the events or to the EventBlock class itself. This then requires that the `_count` member of the EventBlock object is changed whenever the corresponding `_count` members of EventRef::Data objects within the EventBlock are incremented to two, or decremented to one. The EventBlock `_count` member is incremented or decremented by one respectively in these circumstances. The rationale is that the EventRef::Data `_count` must always be at least one, since the EventBlock contains an EventRef element, and will only exceed one when a reference is outstanding in the client code.

Note that this design is based on an assumption that the environment is multi-threaded, and that the concurrency level will be sufficiently high that it is undesirable to have multiple EventRef::Data objects simultaneously accessing a single count variable. If this is not the case, then a simpler design may be made. Each EventRef::Data object would not maintain its own count variable, but would use the EventBlock object count variable. This would have the added advantage of requiring four fewer bytes per event stored. The difference is analogous to the difference between row-level and page-level locking in databases [50].

We deal with the third problem using a standard technique, for which, unfortunately, we have no reference. We overload the new and delete operators for each class for which we desire very rapid allocation and deallocation. We then maintain a stack of deleted objects. Whenever new is invoked, the stack is checked and, if it not empty, the top object is returned. If the stack is empty, then a new object is created in the freestore. The delete operator in turn adds deleted objects to the stack, rather than freeing the space.

This technique can cause a large amount of freestore to be tied up in an otherwise unused stack. There are two approaches to dealing with this. One is to keep track of the number of objects contained in the stack. If it grows too large, many of them may be freed. Alternately, we may wait until we are unable to allocate further memory, and then invoke a cleanup operation on these stacks. Since most systems will not run out of memory until they run out of virtual memory, it is probably wise to keep track of the total allocated freestore, and invoke a cleanup when that amount exceeds some threshold.

5.4 Callback Implementation

Finally, we present the callback implementation details. There are two things that must be dealt with for correct callback operation: registration and trigger determination. We deal with the second problem first, as the solving of it determines what is required of event registration.

We wish to ensure low cost in determining that a callback has been triggered. To do so, we first divide the callbacks into four groups, according to how we determine when a callback has been triggered.

1. iterator
2. firstEvent, newTrace
3. eventCount, spaceConsumption
4. store, rawStore, newPartner, newPartnerStored, covered, retrieve, remove.

The first group, containing only the iterator callback, is not really a trigger-driven callback. Rather, it is an alternate method for iterating over events. As such, any method invocation on the partial order will simply check to see if an iterator callback is currently enabled and not operative. If there is such, an appropriate iterator will be created, and the callback will be invoked for each iteration, unless the callback returns false. At such time as all iterations are completed, or the callback has returned false, the iterator will be cleaned up, per the semantics of the iterator callback.

The second group is invoked when a new trace is required by the storage of some RawEvent. Neither of the callbacks in this group have significant parameters. As such, it is sufficient to have two stacks of such callbacks, one for each type. When a new trace is created, the newTrace stack is iterated through and the callback(TraceID&) is invoked. For firstEvent callbacks it is slightly more complex, as they may have an associated type and the event must be fully incorporated before the callback is invoked. When the event is fully incorporated, we iterate through the stack and determine if there is a type match, invoking the callback if there is. This is reasonable as there are only four event types and these callbacks are only invoked on new-trace creation.

The third group has simple numerical parameters. It is therefore very easy to perform numerical computations to determine when these callbacks will be invoked. For an eventCount callback we record the number of events required to be stored to trigger it. For every full incorporation we decrement that count by one. For event removal, we increment the count by the number of events removed. When the count reaches zero, we invoke the callback. For spaceConsumption callbacks, we convert the parameter to an estimated number of events to be stored. We then treat it in the same manner as an eventCount callback. It is not unreasonable to expect that multiple callbacks of these varieties will be registered with different parameter values. For this, we adopt the timeout data structure described by Tanenbaum [141]. The callbacks are ordered according to the number of events that need to be stored to trigger the callback, from least to greatest. In this way only a single computation and a single comparison is needed per store() invocation to check these callbacks.

The callbacks of the final group take a significant number of parameters, each of which has a significant domain. They must, therefore, be treated carefully to ensure efficient invocation. We first observe that all of the parameters are optional, and it is expected that for any given callback registered, most of the parameters will be unaltered from their default value. In particular, of the five possible parameters that each of these callbacks can be registered with, it is expected that the vast majority of registrations will specify an event type and/or only one of the four other parameters (EventID, TraceID, CutRef, and SliceRef). Second, we observe that a SliceRef can be broken down into a collection of EventIDs and EventIDs, in turn, may be broken down to TraceIDs and EventPositions. The CutRef likewise has an EventPosition for each TraceID, beyond which it will not be triggered. Finally, we observe that all but two (retrieve and remove) of these callbacks occur on the invocation of the store() method. We know that events are received and stored approximately in a linearization of the partial order. Further, we can assume that for some target environments or tools, events will not be stored out of EventPosition sequence, though we have not made that a strict requirement hitherto.

We therefore adopt the following structure. All callbacks in this group are first divided into those that only specify an event-type, and those that make additional or other specifications. Each of the event-type callbacks is placed in a stack associated with its type. On the occurrence of a trigger, we index into the appropriate stack, according to the event type of the triggering event, and invoke any callbacks in that stack.

Any remaining callbacks in this group must have at least one of the other parameters set. Those parameters, as we have observed, all specify at least a TraceID. We therefore index by the TraceID of the triggering event. Any callbacks that are specified based on TraceID, possibly with an event type, are recorded at this level of the structure in one of five stacks, according to the event-type specification of the callback object. We then invoke all callbacks in the undefined event-type stack, and all those in the event-type stack that matches the event-type of the triggering event.

This now leaves only callbacks that specify an EventID, SliceRef (which is an EventID, now that we have indexed by TraceID), or CutRef (which is an upper-bound on EventID). We deal first with the EventID and SliceRef callbacks, since these are discrete references. We sort these callbacks (without distinguishing between them) by EventPosition, from least to greatest. Since events are stored in EventPosition sequence, we can check in one step if any of these are triggered.

Even if events are not stored in strict EventPosition sequence, they will be close to it. As such, we can check if the particular event is in sequence, and if so determine if it has triggered a callback. If it is not in sequence, we can perform a simple linked-list search, which is likely to be efficient given the near-sequentiality of the incoming data. Since it is doubtful that there will be more than one or two callbacks on a given event, we simply check the event type for a match against that specified by the callback, rather than maintaining five stacks.

For event removal, we work up the sorted list of callbacks, invoking them in turn, until we reach a callback with an EventPosition that exceeds the greatest event removed for the given trace. Again, as these callback types specify discrete events, it is doubtful that there will be multiple such for a given event, so we simple check the event type for a match against that specified by the callback.

For event retrieval we cannot take advantage of any ordering technique. There are three approaches that might be used. First, we can perform a search on the sorted list of EventPositions. This will presumably be no better than $O(\log c)$, where c is the number of positions in the sorted list, which is the number of discrete callbacks registered over that trace. This is undesirable, since the retrieval method is highly optimized for efficient operation. However, if only a few EventIDs were specified for a retrieval callback, this might be acceptable. Second, we might add callback member data to the EventRef or EventRef::Data classes, or possibly to an object that is collocated with those objects in the trace structure. This would allow very efficient callback operation, though at the expense of extra storage. Such an approach could also be used for the other callback types of this group, though it would somewhat slow down the retrieval operation, since it would increase the likelihood that the callback variable referenced a non-retrieval callback method. Third, we might simply advise or require that the retrieve callback be limited to event-type and TraceID parameters. We are currently agnostic as to the preferred alternative.

For CutRef callbacks we similarly sort the callbacks registered by EventPosition. However, in this instance when a given trigger-event occurs, we must invoke every callback that has a CutRef with an EventPosition equal or greater than that of the trigger event. We therefore link the callback methods in the same sequence as the EventPositions, though separated into five lists, one for each event type specifiable. For event removal we likewise follow these lists, invoking each callback in turn. Event retrieval has similar problems to those of the EventID and SliceRef cases, and so we do not reiterate those arguments here.

Having resolved the issue of efficient callback invocation, we can now see the requirements imposed on the registration methods. We have already indicated that the registration of a callback object does not directly accept the parameters, if any, for that callback object. However, the registration method must create an associated Parameters object for the callback object registered, and return a reference to that Parameters object. Any parameters are then filled in by invocation of methods on that Parameters object.

Clearly, the parameters cannot be used in raw form. The registerCallback() method must therefore identify a pending registration. This is achieved by pushing the registration information on a stack. Any partial-order method invocation is required to check the state of this stack and, if it is not null, parse the Parameters and set up the appropriate callback. The parsing process also determines any duplication in the specification, and removes it accordingly. For example, if both an EventID and a TraceID are specified, and the TraceID is the same as that contained in the EventID, we ignore the EventID. This prevents the multiple invocation of a callback object that was registered only once.

Since parsing and incorporation into the trigger-determination structure is not likely to be a fast operation, we can optimize slightly. When a callback is registered, its type is specified in the registerCallback() method. We therefore know where it can possibly be invoked from in the partial order code. This allows us to adopt a lazy incorporation approach, in similar vein to lazy evaluation in functional programming [47, 63]. For example, if the method invoked is a retrieve operation, the only pending callback operations that would need their parameters parsed and to be incorporated into the callback trigger-determination structure, would be retrieve callbacks. The activate() method of the Parameters class overrides this lazy parsing, and forces immediate parsing and incorporation into the callback trigger-determination structure.

The deregisterCallback() method must likewise parse its arguments and then remove the associated callback from the trigger-determination structure. This is a more complex process. In particular, the defined structure has so carved up the original parameters that it is not possible to determine from that structure what the various original parameters were to the callbacks that have been registered. Two approaches may be taken to this issue. We may choose to simply parse the deregister parameters, seek them in the trigger-determination structure and remove them accordingly. After all, deregistering something that is not there is largely harmless. If, on the other hand, it is desirable to maintain a consistency between objects registered and deregistered, then we can maintain a simple structure of raw callbacks registered, together with their specified Parameters object. Any deregister operation is first verified against this structure to determine if, indeed, the relevant callback has been registered. If it has, then the deregistration can proceed, per the first approach. If it has not, then the deregistration is halted and, if activate() were called, an error returned.

Given the proposed trigger-determination implementation, it is unclear if the enableCallback() and disableCallback() methods are operationally distinguishable from the registerCallback() and deregisterCallback() methods, respectively. We therefore choose to implement these methods as aliases of one another. In an alternate trigger-determination structure, this might not be the case.

6 A Brief History of Timestamps[1]

We now present current techniques for implementing the comparison and precedence-related-event-set methods of the EventRef class. We have termed this a brief history of timestamps as this is what the work has largely revolved around. In particular, little work has been applied from directed-graph data structures, though this is perhaps a consequence of the more specialized nature of the distributed-computation partial orders. This is evident, for example, in the work of Han [60] on execution differences. We will not remedy this deficiency in this chapter, concentrating instead on techniques that have found wider applicability. In Chapter 7 we will present, albeit briefly, a fuller range of previous approaches, together with justification for their inapplicability to our problem.

The approaches for the precedence-sets methods will build on, and depend on, the specific techniques used for the comparison operators. We therefore first present current solutions for those comparison operators. In dealing with this problem, it is sufficient to solve the single operator `operator<()`. Equality can be determined by comparison of EventIDs. This in turn allows `operator<=()` to be computed. The operators `operator>()` and `operator>=()` are the reverse of the previous two operators, respectively. Finally, concurrency is the negation of `operator<=()` and `operator>=()`. For the remainder of this dissertation, the phrases **precedence determination** and **precedence test** will be synonymous with `operator<()`.

The simplest method for precedence determination would be to store the partial order as a directed acyclic graph. In such a case precedence determination is a constant-time operation because the partial order is transitively closed and so there is an edge between any two events that are ordered. However, the space consumption for this method is likely to be unacceptably high. A naive implementation would require $O(n^2)$ space to store the matrix of edges, where n is the number of events in the computation. We are not aware of any work that has approached the problem starting from this point and attempting to improve the space consumption. The primary problem, from our perspective, is that it ignores the optimizations available from the sequentiality of the traces.

As we have noted in the previous chapter, all systems we are aware of, including our own, store, approximately, the transitive reduction of the partial order. This approach is the least space-consumptive possible. Precedence determination then requires computing whether there is a directed path between the two events. There are standard methods for such graph-searching operations available in most algorithm texts (*e.g.* [26]). Unfortunately, such techniques tend to be slow and space-consumptive. A depth-first approach would take time $O(n + m)$, where n is the number of events and m the number of messages, and space $O(d)$, where d is the depth at which either precedence is determined or backtracking takes place. While there is no bound on this depth in an online environment, we have required that events cannot be fully incorporated

[1]With apologies to Stephen Hawking.

until all of their predecessors are so incorporated. As such, the algorithm can treat the partial order as fixed. Even so, there is no guarantee that it would not require examination of all events currently stored. In particular, this is likely if the precedence test fails, either because the events are concurrent or because they precede in the opposite sense of that being tested. The breadth-first alternative is no better, with a time-cost of $O(n + m)$ and space-consumption of $O(b^d)$ where b is the average number of message transmissions per event (the branching factor of the graph), though in this case d is the depth of the shortest path.

6.1 Logical Time

To compensate for this deficiency some additional information is added to allow for a more-efficient precedence test. The most common form this additional information takes is a logical timestamp, which receives its name from the fact that it determines the logical-time ordering of events. In the directed-graph view of a partial order, timestamps amount to the addition of some edges, beyond the transitive reduction of the partial order, to reduce the search time. There are several logical timestamps which can be categorized based on five features.

Distributed v. centralized creation: A distributed timestamp-creation algorithm is one in which the timestamp can be generated within the distributed computation of interest. By contrast, a centralized-creation algorithm requires the gathering of event information by the observation tool to enable timestamp creation.

Examples of distributed creation algorithms include the Fidge/Mattern [43, 100], Fowler/-Zwaenepoel [45], and Jard/Jourdan [70] timestamps. Examples of centralized creation include the Ore timestamp [114, 140], Summers cluster algorithm [140], and our own dimension-bounded [166] and centralized-cluster [168, 169] algorithms.

For the purpose of distributed-system observation, either technique will suffice, as the timestamps are typically generated in the monitoring entity, and any distributed-creation algorithm can be implemented centrally.

To our knowledge, we are the first to recognize this distinction. Hitherto, all logical-time algorithms presumed that they were to be created in a distributed environment. The value of centralized creation proved crucial to the solutions we will present in Part III.

Distributed v. centralized precedence-testing: Distributed precedence-testing is the property of being able to determine the precedence relationship between two events using just the timestamps of those events. Centralized precedence-testing requires additional information, typically timestamps for additional events, to determine the precedence relationship between two events.

Examples of algorithms that enable distributed precedence-testing include the Fidge/Mattern and Ore timestamps. Examples of algorithms requiring centralized precedence-testing include the Fowler/Zwaenepoel, Jard/Jourdan and Summers cluster timestamps.

For the purpose of distributed-system observation, distributed precedence-testing is preferred, though not essential. Specifically, it enables a more flexible architecture. While our interface buries the actual querying of other events' timestamps behind the `operator<()`

method, the various effects (performance, architectural, *etc.*) of having to perform those additional queries cannot be buried.

As with the previous item, we are the first, to our knowledge, to recognize this distinction between distributed and centralized precedence-testing. A similar, though distinct, concept was recognized by Meldal, Sankar, and Vera [105]. They observed that in a distributed environment a process is likely only interested in determining the precedence relationship between messages that arrive at that process, rather than between arbitrary messages.

Static v. dynamic: Static timestamp algorithms are those in which all events must be processed before the timestamps can be generated. Dynamic algorithms can timestamp events either as they occur or as they are received by the monitoring entity. Typically the ordering of the processing of events is restricted to being a linearization of the partial-order if the creation algorithm is centralized. This is equivalent to the (implicit) requirement of distributed-creation algorithms that the events must occur in a valid state of the computation.

Examples of static algorithms include the Ore and Summers cluster timestamps. Examples of dynamic algorithms include Fidge/Mattern, Fowler/Zwaenepoel, and Jard/Jourdan.

For the purpose of distributed-system observation, dynamic algorithms are preferred and sometimes essential. For example, both monitoring and control require dynamic algorithms. Most users of distributed debugging systems prefer dynamic tools. Exceptions to this include computation replay, which by its very nature, admits static algorithms.

Time-bounds: There are several time-bounds of interest in logical timestamps, which can be broadly reduced to two: creation time-bound and usage time-bound. The creation time-bound varies from constant in the case of Fowler/Zwaenepoel to linear in the number of traces in the Fidge/Mattern case. In other cases, such as the Ore timestamp, it is indeterminate, since the algorithm does not fully specify the creation method.

Usage time-bounds depend on the specific query. Although the time-bound to compute event precedence, greatest predecessor set and least successor set varies from constant-time to linear in the number of messages in the computation, no single timestamp algorithm achieves the constant time-bound for all three queries. Further, no dynamic algorithm achieves a constant time-bound for the least-successor set. Note that to actually use the greatest-predecessor or least-successor set would require algorithms of lower-bound $O(N)$, where N is the number of traces, and so an $O(N)$ algorithm for either of these queries would be acceptable.

For the same reason that dynamic-timestamp algorithms are preferred and sometimes essential, lower time-bounds are preferred and sometimes essential. Specifically, precedence-test algorithms that have a cost that is proportional to the number of events, messages, or traces are likely unacceptable. It is doubtful that such algorithms will scale.

Space consumption: The space consumption of a timestamp varies from constant size to linear in the number of traces. Since each event stored in the partial-order data structure includes the timestamp of the event, timestamps that have size linear in the number of traces will not scale. We comment on this issue in detail in Chapter 7.

We now present some of the more-significant timestamp algorithms, indicating where they are lacking in the required features, as identified above. The descriptions that follow are strict mathematical representations of how the given timestamp is computed, together with a formal precedence test for that timestamp. They do not indicate how the computation is to be performed. In particular, they do not indicate how the data necessary to the computation is acquired. This is distinct from the usual manner in which timestamps are described, typically in terms of their creation within a distributed environment. Our description is directly applicable to the centralized creation that the PartialOrder class would be required to compute for the store() method. The extensions in the following for synchronous events are our own, as we are almost unique in modeling a synchronous event as a single event that occurs in multiple traces.

6.1.1 The Lamport Timestamp

The Lamport timestamp [96] is designed as follows. Each event e is assigned an integer timestamp, $\mathcal{L}(e)$, in terms of the events it covers (Definition 10, page 11). First, define $\mathcal{L}'(e^c)$ as follows:

$$\mathcal{L}'(e^c) = \mathcal{L}(e^c) + 1 \tag{6.1}$$

Then the Lamport timestamp of e is:

$$\mathcal{L}(e) = \max_{e^c <: e} \left(\mathcal{L}'\left(e^c\right)\right) \tag{6.2}$$

where $\max$ is the maximum of a set of integers. The first event in a trace is timestamped by creating a virtual event in the trace at EventPosition(0), which is thus covered by the first event. This virtual event is assigned a timestamp set to 0. Equation 6.2 is then used to determine the timestamp of the first event in the trace. As can readily be seen, both the creation time-bound and space consumption of this algorithm are $O(1)$. It is dynamic, since the timestamp for any event depends only on the immediate predecessors.

The Lamport timestamp cannot completely determine precedence. Given two events e^i and e^j with $\mathcal{L}(e^i) < \mathcal{L}(e^j)$ the only thing that can be concluded is that $e^j \not\prec_{\mathcal{E}} e^i$. We cannot determine with Lamport time whether $e^i \parallel_{\mathcal{E}} e^j$ or $e^i \prec_{\mathcal{E}} e^j$. The reason is that Lamport time is imposing a total order on the partial order. From a directed-graph viewpoint, what is going on with Lamport time is that the edges being added to the graph are not simply reducing the precedence-test search time by adding edges implied by the transitive closure of the partial order. Rather, Lamport time is extending the partial order, adding edges to the graph not implied by the partial order.

An example of Lamport timestamps is shown in Figure 6.1. Note that $e_1^3 \parallel_{\mathcal{E}} e_2^3$ even though $\mathcal{L}(e_1^3) < \mathcal{L}(e_2^3)$. Since this timestamp is not capable of complete precedence determination, we do not discuss it further.

6.1.2 The Fidge/Mattern Timestamp

The Fidge/Mattern timestamp [42, 43, 100, 131] augments the transitive-reduction graph with a set of edges to every event from its greatest predecessor in each trace. It is designed as follows.

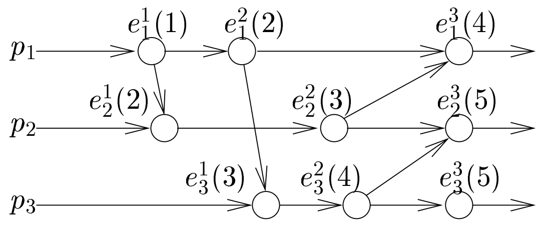


Figure 6.1: Lamport Timestamps

Each event e is assigned a vector timestamp $\mathcal{FM}(e)$ of size N, indexed by TraceID, in terms of the events it covers. First, define $\mathcal{FM}'(e^c)$ as follows:

$$\mathcal{FM}'(e^c)[i] = \begin{cases} \mathcal{FM}(e^c)[i] + 1 & \text{if } i \in \phi(e^c) \\ \mathcal{FM}(e^c)[i] & \text{otherwise} \end{cases} \tag{6.3}$$

(Recall that $\phi(e^c)$ maps event e^c to its traces.) Then the Fidge/Mattern timestamp of e is:

$$\mathcal{FM}(e) = \max_{e^c <: e} (\mathcal{FM}'(e^c)) \tag{6.4}$$

where $\max$ is the element-wise maximum of a set of vectors. The first event in a trace is timestamped by creating a virtual event in the trace at EventPosition(0), which is thus covered by the first event. This virtual event is assigned a timestamp with all elements set to 0. Equation 6.4 is then used to determine the timestamp of the first event in the trace. As can readily be seen, both the creation time-bound and space consumption of this algorithm are $O(N)$, where N is the number of traces. It is dynamic, since the timestamp for any event depends only on the immediate predecessors.

Precedence testing between two events, e^i and e^j, can be determined by the equivalence:

$$e^i \prec_{\mathcal{E}} e^j \iff \exists_{p \in \phi(e^i)} \; \mathcal{FM}(e^i)[p] < \mathcal{FM}(e^j)[p] \tag{6.5}$$

It is a relatively easy matter to demonstrate that

$$\exists_{p \in \phi(e^i)} \; \mathcal{FM}(e^i)[p] < \mathcal{FM}(e^j)[p] \iff \forall_{p \in \phi(e^i)} \; \mathcal{FM}(e^i)[p] < \mathcal{FM}(e^j)[p] \tag{6.6}$$

As such, only a single test is needed, whether or not the event is synchronous. Thus, the precedence test is constant time.

While it is possible to implement the vector as an associative array, storing only the non-zero values, we have found that in practice this does not save space. The reason is that the timestamp captures transitive causality. As such most elements of the vector for most events for typical computations are non-zero. Alternate encodings for the Fidge/Mattern timestamp will be discussed in Chapter 7.

An example of Fidge/Mattern timestamps is shown in Figure 6.2. The reader can easily verify that the precedence test is correct for the timestamps shown.

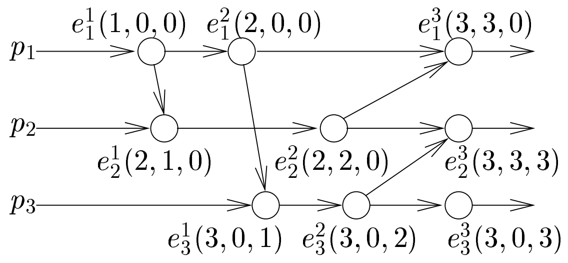


Figure 6.2: Fidge/Mattern Timestamps

6.1.3 Fowler/Zwaenepoel Timestamps

The Fowler/Zwaenepoel timestamp [45] augments the transitive-reduction graph with a set of edges to every event e from the greatest event in any trace that has communicated directly with the trace that e is in. Note that these greatest events may not be the greatest predecessors of e, as this timestamp does not capture transitive dependency. The Fowler/Zwaenepoel timestamp is designed as follows. Each event e is assigned a vector timestamp $\mathcal{FZ}(e)$, indexed by TraceID, in terms of the events it covers. For all events e^c, covered by event e, we define $\mathcal{FZ}'(e^c)$ thus:

If $\phi(e) \cap \phi(e^c) \neq \emptyset$ then:

$$\mathcal{FZ}'(e^c)[p] = \begin{cases} \mathcal{FZ}(e^c)[p] + 1 & \text{if } p \in \phi(e^c) \\ \mathcal{FZ}(e^c)[p] & \text{otherwise} \end{cases} \tag{6.7}$$

Otherwise:

$$\mathcal{FZ}'(e^c)[p] = \begin{cases} \mathcal{FZ}(e^c)[p] & \text{if } p \in \phi(e^c) \\ 0 & \text{otherwise} \end{cases} \tag{6.8}$$

Then the Fowler/Zwaenepoel timestamp of e is:

$$\mathcal{FZ}(e) = \max_{e^c <: e} \left(\mathcal{FZ}'\left(e^c\right)\right) \tag{6.9}$$

As with Fidge/Mattern, there is a virtual event prior to the first event in a trace to enable the timestamping of the first event. Again, as with Fidge/Mattern, it is dynamic, since the timestamp for any event depends only on the immediate predecessors. Unlike the Fidge/Mattern algorithm, though, the time and space complexity is not clear. While the Fowler/Zwaenepoel timestamp is, in the worst case $O(N)$, where N is the number of traces, this presumes that every trace communicates *directly* with every other trace at some point in the computation. While it is true that some computations exhibit this property, most do not. Indeed, many parallel computations are structured such that $N-1$ of the traces will only communicate with their “neighbours” and a master trace. In such computations the time and space complexity will be amortized constant.

There is no specific precedence test for Fowler/Zwaenepoel timestamps as their intended application was causal distributed breakpoints. In that application, what is determined is the set of greatest predecessors within each trace. This is done by recursive search through the events

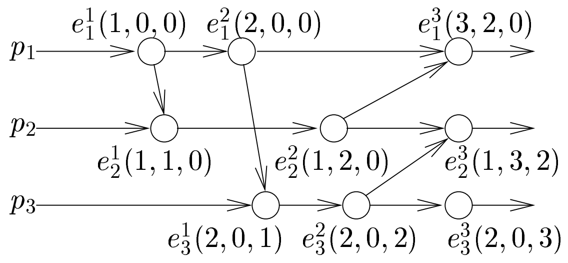


Figure 6.3: Fowler/Zwaenepoel Timestamps

referenced in the timestamp. Each branch of the search is terminated if its timestamp does not indicate a greater predecessor than is currently known. Thus, in Figure 6.3, which shows an example of Fowler/Zwaenepoel timestamps, if we wished to determine the greatest predecessors of event e_2^3 in each trace, we would tentatively identify events e_1^1, e_2^2 and e_3^2. We would visit these events in turn. The first two would turn up no additional information, but visiting event e_3^2 would identify event e_1^2 as being a greater predecessor in the first trace than event e_1^1. We would then visit event e_1^2 at which point the search would terminate.

Given the ability to determine greatest predecessors by trace we can answer the precedence question. However, if we only wished to answer a precedence question we may be able to terminate the search earlier. For example, we can determine $e_1^1 \preceq_{\mathcal{E}} e_2^3$ just by the Fowler/Zwaenepoel timestamp of e_2^3. The cost of determining precedence is in the worst case linear in the number of messages, and in the best case constant time.

As with the Fidge/Mattern timestamp, we may implement the algorithm using either an associative array or a vector of size equal to the number of traces. In this case it is probably better to use the associative array, as the size of the array is proportional to the number of traces that directly communicate with the trace in which the event being timestamped occurs. The reason is that the Fowler/Zwaenepoel timestamp is only tracking direct dependencies

6.1.4 Jard/Jourdan Timestamps

Jard/Jourdan timestamps [70] are an extension of Fowler/Zwaenepoel timestamps that introduce the concepts of observability and pseudo-direct dependence. Not all events are considered to be worth observing, and so they are divided into observable and unobservable ones. However, the model of distributed computation is such that any event (unary events excepted, but then their existence is predicated on their usefulness), whether observable or not, can cause a transitive dependency that needs to be captured. For this reason, Jard/Jourdan timestamps introduce the notion of pseudo-direct dependency. An event e is pseudo-dependent on an event e^d (written $e^d \ll e$) if there is a path from e^d to e with no observable events on that path. Jard/Jourdan timestamps then create Fowler/Zwaenepoel-like timestamps for just the observable events. The only additional difference is that the Fowler/Zwaenepoel timestamp maintains information in its vector about dependencies in other traces even after intervening (observable) events. The Jard/Jourdan timestamp only maintains information about immediate pseudo-direct dependencies.

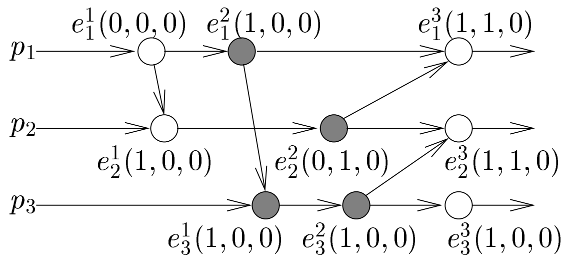


Figure 6.4: Jard/Jourdan Timestamps

As with the previous cases, each event e is assigned a vector timestamp $\mathcal{J}(e)$, indexed by TraceID, in terms of the events it covers. For event e^c we define $\mathcal{J}'(e^c)$ as follows. If e^c is observable then:

$$\mathcal{J}'(e^c)[p] = \begin{cases} \mathcal{J}(e^c)[p] + 1 & \text{if } p \in \phi(e^c) \\ 0 & \text{otherwise} \end{cases} \tag{6.10}$$

Otherwise:

$$\mathcal{J}'(e^c) = \mathcal{J}(e^c) \tag{6.11}$$

Then the Jard/Jourdan timestamp of e is:

$$\mathcal{J}(e) = \max_{e^c <: e} (\mathcal{J}'(e^c)) \tag{6.12}$$

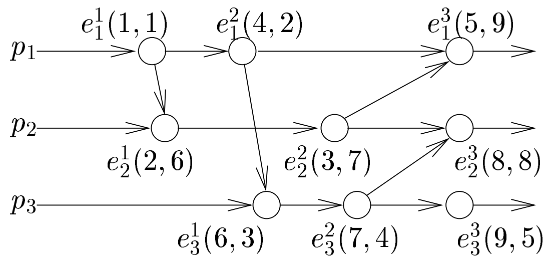
An observable virtual event, with vector timestamp 0, prior to the first event in each trace enables the correct timestamping of that first event. The algorithm is dynamic, since the timestamp for any event depends only on its immediate predecessors. The time and space complexity is similar to that of Fowler/Zwaenepoel, though it will generally be somewhat lower, as this method drops vector elements that are not pseudo-direct predecessors.

As with Fowler/Zwaenepoel, a recursive search is needed for the precedence test. Since only pseudo-direct dependency information is kept by the timestamp, the length of this search is likely to be notably longer that that of the Fowler/Zwaenepoel. This is especially true for computing the greatest predecessor set. There is no precedence test for unobservable events, nor is it clear what would be needed to create such a test. In the absence of such a test, this technique could not be used for our partial-order data structure.

We may implement the algorithm using either an associative array or a vector of size equal to the number of traces. As with Fowler/Zwaenepoel, it is probably better to use the associative array, for essentially the same reason.

An example of Jard/Jourdan timestamps is shown in Figure 6.4. The unshaded events are the observable events.

The Jard/Jourdan timestamp is then Fowler/Zwaenepoel-like with respect to the observable events. If all events are observable it degenerates to the Fowler/Zwaenepoel timestamp with one exception. The difference is that the Fowler/Zwaenepoel timestamp maintains information in its vector about dependencies in other traces even after intervening events. The Jard/Jourdan



Realizer: $< e_1^1, e_2^1, e_2^2, e_1^2, e_1^3, e_3^1, e_3^2, e_2^3, e_3^3 >< e_1^1, e_1^2, e_3^1, e_3^2, e_3^3, e_2^1, e_2^2, e_2^3, e_1^3 >$

Figure 6.5: Ore Timestamps

timestamp only maintains information about immediate direct dependencies. As a result, the Jard/Jourdan timestamp is likely to have a notably longer search time in determining precedence if all events are observable.

From the directed-graph viewpoint, the Jard/Jourdan timestamp represents a set of edges connecting each event e to the greatest observable event in a trace that has communicated pseudo-directly with the event e. As with Fowler/Zwaenepoel, these greatest events may not be the greatest predecessors of e, both because this timestamp technique does not capture transitive dependency and because it ignores unobservable events.

6.1.5 THE ORE TIMESTAMP

The Ore timestamp [114, 140] is unique among vector timestamps in that it does not take a trace view. Rather, it is based on having a realizer for the partial order. The linear extensions are arbitrarily ordered from 1 to d, where d is the number of extensions. Each event e in each extension l_i is assigned an integer $\mathrm{leid}(l_i, e)$ to indicate its position within that extension. The following equivalence must hold for this assignment:

$$e^j \prec_{l_i} e^k \iff \mathrm{leid}(l_i, e^j) < \mathrm{leid}(l_i, e^k) \tag{6.13}$$

The Ore timestamp $\mathcal{O}(e)$ for event e is then defined as:

$$\forall_{i:0<i\leq d}\, \mathcal{O}(e)[i] = \mathrm{leid}(l_i, e) \tag{6.14}$$

Precedence testing between two events, e^j and e^k, can be determined by the equivalence:

$$e^j \prec e^k \iff \forall_{i:0<i\leq d}\, \mathcal{O}(e^j)[i] < \mathcal{O}(e^k)[i] \tag{6.15}$$

An example of Ore timestamps is shown in Figure 6.5. Note that the vector size is equal to d, while the precedence test is $O(d)$. The value of d can be as large as $O(n!)$, where n is the number of events in the computation. For any reasonable implementation of a realizer it would be no larger than the width, which can be no larger than the number of traces. It can be as small as the

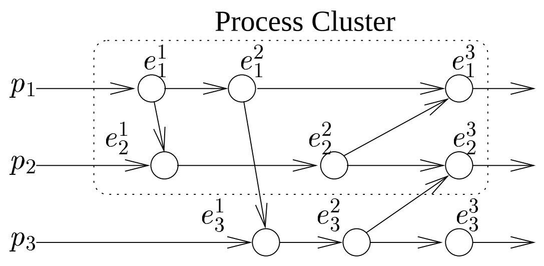


Figure 6.6: Summers Cluster Example

dimension of the partial order. While any algorithm that produces a realizer can be used for this timestamp, one that produced a large realizer (that is, one with a large value of d) would be of little value. Unfortunately, computing a minimal realizer is NP-hard [177]. Although, we know of no dynamic algorithm to build a realizer, we were able to adapt this timestamp technique as our first scalability solution, presented in Chapter 8.

6.1.6 The Summers Cluster Timestamp

The Summers cluster timestamp [140] was an attempt to increase the efficiency of timestamp generation for user-defined abstract trace clusters. The essence of the idea is the recognition that events within a cluster can only be causally dependent on events outside the cluster through receive events from transmissions that occurred outside the cluster.[2] Such events are called "cluster receives." For example, in Figure 6.6 event e_2^3 is a cluster receive. By identifying such cluster-receive events, it is possible to shorten the timestamp of all other events within the cluster to twice the number of traces in the cluster. Half of this timestamp determines precedence within the cluster. The other half identifies the greatest cluster-receive events in each trace in the cluster that precede the given event. The Summers algorithm is a two-step method for achieving this.

First, all events are timestamped using the Fidge/Mattern technique described in Section 6.1.2. During this process all cluster-receive events are identified and their timestamps recorded. The second step computes the cluster timestamp for all events within the cluster as follows. If the event is a cluster receive, the first half of the timestamp is set to the projection over the cluster traces of the Fidge/Mattern timestamp recorded for that event. Otherwise, the first half of the timestamp is computed using the Fidge/Mattern technique, but constrained to the cluster traces. To compute the second half of the timestamp the algorithm maintains a cluster-receive vector that is analogous to a Fidge/Mattern vector. The difference is that it is incremented only on the occurrence of a cluster receive. The second half of the timestamp is first set to the maximum of the second half of the timestamps of all events covered. If the event is a cluster receive, then the position corresponding to the trace in which the cluster receive occurred is incremented.

[2]This is not strictly true. Events within a cluster can also be causally dependent on events outside the cluster through synchronous events which have homes both within and without the cluster. However, Summers developed the concept in an asynchronous context, and thus did not have to deal with synchronous events.

The precedence test for Summers cluster timestamps depends on whether or not both events being compared are within the cluster. If they are, then the test is identical to the Fidge/Mattern test of Equation 6.5. Summers does not give a test for one event within the cluster and one outside, since his purpose was the timestamping of cluster events. However, he does give a mechanism for the generation of a Fidge/Mattern timestamp from the cluster timestamp, since he wanted the ability to alter the set of traces within a cluster. This mechanism is to take the maximum of all cluster-receive-event timestamps referenced in the second half of the cluster timestamp and then add in the cluster timestamp values in the vector locations where the cluster exists. Precedence testing then uses the Fidge/Mattern technique.

The precedence test is constant time for events within the cluster. For comparison to events outside the cluster, it is linear in the product of the number of traces and the cluster size, because of the need to recreate the Fidge/Mattern timestamp. The precedence test is also centralized, though as we have noted, this is not a significant issue. In addition, it is a static algorithm, which substantially limits its applicability to our application. While the space reduction is potentially useful, prior to our work we were not aware of any measurements that quantify that reduction. We were able to adapt this timestamp technique as our second scalability solution, presented in Chapter 9.

6.2 PRECEDENCE-RELATED EVENT SETS

The quick summary of our timestamp survey is that only the Fidge/Mattern timestamp is applicable to our problem. Specifically, it is the only timestamp that is dynamic and has an efficient precedence test. As such, this is the timestamp of choice in all existing distributed-system-observation tools of which we are aware. We can therefore turn to the second half of the precedence problem, determining precedence-related event sets, on the assumption that the Fidge/Mattern timestamp is used to compute `operator<()`. We first deal with predecessors and successors.

6.2.1 PREDECESSORS AND SUCCESSORS

There are four methods that fall under this banner, the predecessor, successor, greatest-predecessor, and least-successor sets. In terms of implementation, however, there are in effect only two. The predecessor set is simply the cut corresponding to the greatest-predecessor slice. Likewise, the least-successor slice marks the boundary of the successor set. We therefore only compute those slices.

The computation of the greatest-predecessor slice is particularly easy when Fidge/Mattern timestamps are employed. Specifically, the Fidge/Mattern timestamp directly encodes the slice of greatest-predecessor events. As such, the response to this query is simply to return a reference to this slice, which is constant time. That said, use of this slice will have a cost at least proportional to the number of elements used, reflecting the information-theoretic lower bound of the query of $O(N)$, where N is the number of traces.

The Fidge/Mattern timestamp does not, however, encode the least-successor slice. The typical manner in which this slice is computed is by searching along each trace from the event identified by the greatest-predecessor slice, since the least successor on a trace cannot be less than the greatest predecessor. As the traces are totally ordered, and the data structure keeps track of the current maximum event stored, we can perform a binary search. We propose optimizing this slightly, given that events exist in blocks. Rather than performing the usual divide-and-conquer binary search, we advocate a galloping search, that doubles the number of events skipped on each check, starting with one. Thus, if the greatest predecessor were at position i, we examine position $i + 1$, $i + 2$, $i + 4$, $i + 8$ and so forth, until we either reach the maximum event, or discover a successor. If we discover a successor, we perform a binary search in the, presumably reduced, range. While in either approach, the worst-case search time is $O(\log n_i)$, where n_i is the number of events on the trace between the greatest-predecessor event and the maximum event, we believe the galloping search is better in practice when the traces have large numbers of events. This issue requires experimental investigation.

Since the greatest predecessor may be the minimum event, n_i can be as large as the number of events on the trace. Thus, the worst case time required for the complete slice computation is $O(\Sigma_p \log n_p)$, where n_p is the number of events currently stored in trace p. If events are evenly distributed among traces (not, in general, a reasonable assumption, but convenient for these purposes), then that worst-case cost is $O(N \log(n/N))$, where N is the number of traces and n is the total number of events currently stored. We do not believe this worst case tends to occur in practice.

A static variation on Fidge/Mattern timestamps, called reverse vector time [7], achieves a constant time-bound for the least-successor computation. It does so by pretending the computation is run backwards, treating receive events as transmits, and *vice versa*, and computing Fidge/Mattern timestamps accordingly. Given its clearly static nature, it is not clear that this is useful in our application. It is conceivable that we might compute a variant of this, that would speed up the least-successor computations, though we have not investigated this possibility. In particular, such an approach would exacerbate the current scalability problems.

6.2.2 THINGS CONCURRENT

The least- and greatest-concurrent sets appear somewhat more expensive to compute, as the elements they contain must be mutually concurrent. Our initial, naive, algorithm [164] starts with the greatest-predecessor set and increments the elements. This results in a slice whose elements are either concurrent with, or successors of, the event. The algorithm then iterates through the elements of the slice, removing any that are successors of the event. This leaves only concurrent elements. The algorithm then performs a nested loop to remove any elements that are not mutually concurrent. The total execution cost is therefore $O(N^2)$. This is unacceptably high.

We now present a novel algorithm for computing the least-concurrent set in time $O(N)$ where the number of partner events is limited to one. To do so, we prove the following theorem.

Theorem 2 (Least Concurrent) *An event e^j is least concurrent to an event e if-and-only-if the* *two events are concurrent and the events covered by e^j are predecessors of e.*

$$e^j \in \mathrm{leastConcurrent}_{\mathcal{E}}(e) \Longleftrightarrow \left(e^j \parallel_{\mathcal{E}} e \wedge \forall_{e^k <: e^j}\, e^k \prec_{\mathcal{E}} e\right)$$

Proof:
($\Longrightarrow$) Since $e^j \in \mathrm{leastConcurrent}_{\mathcal{E}}(e)$ then, by the definition of the least-concurrent set (Equation 4.4), $\nexists_{e^k \prec_{\mathcal{E}} e^j}\, e^k \parallel_{\mathcal{E}} e$. Therefore $\nexists_{e^k <: e^j}\, e^k \parallel_{\mathcal{E}} e$. Therefore $\forall_{e^k <: e^j}\, e^k \prec_{\mathcal{E}} e \vee e \prec_{\mathcal{E}} e^k$. If $e \prec_{\mathcal{E}} e^k$, then $e \prec_{\mathcal{E}} e^j$, since $e^k <: e^j$. This contradicts $e \parallel_{\mathcal{E}} e^j$, which is true because $e^j \in \mathrm{leastConcurrent}_{\mathcal{E}}(e)$. Therefore $\forall_{e^k <: e^j}\, e^k \prec_{\mathcal{E}} e$.
($\Longleftarrow$) Since $e^j \parallel_{\mathcal{E}} e$, it suffices to show that $\nexists_{e^k} \left(e^k \parallel_{\mathcal{E}} e \wedge e^k \prec_{\mathcal{E}} e^j\right)$. Suppose there did; let that event be e^l. Then we have $e^l \parallel_{\mathcal{E}} e$ and $e^l \prec_{\mathcal{E}} e^j$. Since $e^l \prec_{\mathcal{E}} e^j$, then there exists some event $e^k <: e^j$ such that $e^l \prec_{\mathcal{E}} e^k$. Since $\forall_{e^k <: e^j}\, e^k \prec_{\mathcal{E}} e$, $e^l \prec_{\mathcal{E}} e$ by transitivity. This contradicts $e^l \parallel_{\mathcal{E}} e$. □

The algorithm is then a straightforward application of this theorem. We first compute the greatest-predecessor slice of the event, and increment it by one. Each element in that slice is then either concurrent with or a successor to the event. We then iterate over the slice, removing any element which is a successor to the event, or whose non-trace covered events are not predecessors to the event. What remains are the least-concurrent events. Note that we do not need to examine the covered event on the same trace, as it must be a predecessor.

This algorithm works correctly whether or not the data structure is limited to a single partner. However, observe that when we are limited to a single partner the algorithm is not only $O(N)$, but it has a very low constant. Its performs only one iteration over the traces, and the contents of that iteration are only one event retrieval and two invocations of `operator<()`. In addition, note that this execution cost remains the same even if the environment allows multicast. That is, if transmit events can have multiple partners, but receive and synchronous events cannot, then no event will cover more than two events. Further, even in environments in which receive and synchronous events can have multiple partners, most events will have only one or no partners. As such, this is likely to be amortized $O(N)$ for those environments.

The greatest-concurrent set is computed in an analogous fashion, though starting with the least-successor set and decrementing it. The naive algorithm will likewise have an execution cost of $O(N^2)$. The more-sophisticated algorithm is based on the corresponding theorem.

Theorem 3 (Greatest Concurrent) *An event e^j is greatest concurrent to an event e if-and-only-if the two events are concurrent and the events that cover e^j are successors of e.*

$$e^j \in \mathrm{greatestConcurrent}_{\mathcal{E}}(e) \Longleftrightarrow \left(e^j \parallel_{\mathcal{E}} e \wedge \forall_{e^j <: e^k}\, e \prec_{\mathcal{E}} e^k\right)$$

Proof: The proof is analogous to that of the previous theorem, and so we omit it.□

The algorithm is likewise similar to the previous one, though starting with the least-successor set and decrementing it. The cost of the algorithm is, however, less attractive. The least-successor-set computation will dominate, and thus it is $O(N \log(n/N))$ for the single-partner case. If multicast is allowed, the cost will be higher, as any multicast transmit will have multiple covering events. Note, however, that if multicast is the only form of multiple partners possible, then the amortized cost will remain the same, as for every M-way multicast transmit, there will be M receivers that will have only a single covering event. Multi-receive will not increase the cost, as such events likewise have only one covering event.

Finally, we note that when computing the greatest-concurrent set for maximal events, a useful optimization is possible. In this case, we do not need to compute the least-successor set. Rather, we can just use the current slice of maximum events stored. The events in this slice cannot be successors, since the event we are computing the set for is maximal. This allows the computation to execute with the same efficiency as does the least-concurrent computation. This case is not artificial. In particular, it occurs when processing events in a linearization of the partial order.

The naive algorithms were initially developed for our offline dimension-bound-analysis algorithm [164]. We subsequently developed alternate techniques [165] that allowed us to avoid these sets. However, we found it sufficiently useful as a mode of thinking about partial-order algorithms as to include them within our structure, and to apply those techniques to the more-efficient computation of these sets.

We close this section with a general observation about the usage of precedence-related event sets. It is sometimes the case that the entire set is not examined. For example, when least-successor sets of different events are compared to determine if the events have partially, completely or non-overlapping causal futures (see Basten [7]), if the futures are only partially overlapping, then the determination may be made without examining all of the elements. In such a case it would not be necessary to compute all of the least-successor sets. Rather, a lazy-evaluation approach might be adopted, wherein elements are computed on demand. We are not aware of any work that has been done in this regard.

7 SCALABILITY PROBLEMS

Having described the current state-of-the-art techniques for implementing our proposed partial-order data structure, we now wish to look at what breaks as the system under observation gets larger. There are two, non-orthogonal, ways in which distributed systems can grow under our model: an increased number of events and an increased number of traces. An increase in the number of events is not, by and large, problematic. More precisely, it does not present novel problems that are not actively addressed in any data structure containing large numbers of elements. An increase in the number of traces, on the other hand, can be troublesome. As we observed in the previous chapter, precedence testing is dependent on Fidge/Mattern timestamps, which grow linearly with the number of traces. The cost of computing such timestamps is, in turn, proportional to their size. In this chapter we will demonstrate that the size of these timestamps is a fundamental scalability limitation. We will then show the theoretical foundation for the size of these timestamps, and indicate why this basis is not relevant in our circumstance. Finally, we close by quickly surveying various techniques that have been applied to reduce the timestamp size.

7.1 THE VECTOR-CLOCK-SIZE PROBLEM

Given that the only timestamp that currently satisfies the requirements for precedence determination in our data structure is the Fidge/Mattern, let us assume that we use it and observe the problems that occur as the number of traces grows. For this thought experiment it is not unreasonable to imagine something like a thousand traces, with an average of a thousand events on each trace. Thus we have about a million events, with around 40 bytes per event in fixed-size data, or about 40 megabytes in the data structure, prior to dealing with precedence. This is core data structure, and cannot be removed, to be recalculated as needed. This is in contrast with Fidge/Mattern data which can either be stored with the data structure, or computed as needed.

7.1.1 ONE BIG STRUCTURE

We now deal with the first of these possibilities. We will compute the Fidge/Mattern timestamps as events are stored, and include that timestamp with the event stored. A minor, but significant issue, is that we will augment the EventRef::Data class with a pointer to the Fidge/Mattern vector, rather than with the vector itself. While this does incur an additional indirection, various applications will have no need of the timestamp, and their caching performance would suffer in the presence of such a large vector.

Since there are a million events, each storing a thousand-element vector, the memory consumption for the data structure goes up by two or four gigabytes, depending on whether the elements are encoded as shorts or integers. Most machines do not have that much physical memory, and so the data structure will reside, in part, in virtual memory. Note that the Moore's Law growth of memory [108] will not rescue us, since Parkinson's Law of Data [121] indicates that memory usage grows by an equal amount in the given time. Simply put, if we have more memory, we will want to tackle bigger problems.

We now determine the implications of such a sizable data structure. To do so, we enumerate the uses of the timestamp. They are

1. precedence testing
2. greatest-predecessor determination
3. causal-history comparison
4. computing new timestamps.

For the last three of these items, the entire timestamp vector will be used. They therefore, potentially, operate well with respect to both virtual memory and cache, since spatial locality is good. Note, however, that the last two items in the list are somewhat fictitious. We only need to compute new Fidge/Mattern timestamps because we use Fidge/Mattern timestamps. Further, our assumption at this stage is that the timestamps are computed and stored. There is, thus, little value in this advantage. As for causal-history comparison, we know of no tool that provides or uses this technique.

With regard to greatest-predecessor determination, while it is true that the entire timestamp is used, if the computation that uses the individual elements of the greatest-predecessor is sufficiently involved, the benefit of proximity between the elements may be lost. In that instance, we would lose nothing by computing the greatest-predecessor components as they are required.

The most significant issues, however, occur when dealing with precedence testing. Of the thousand elements in the vector, only one is used, or two if concurrency is being determined. Assuming we used integers to encode the vector elements, we have a 4 KB vector, which is the page size in a typical virtual memory. (Linux on Alpha AXP systems uses 8 KB pages and on Intel x86 systems it uses 4 KB pages [126]. AIX uses 4 KB pages [66]. Windows NT, like Linux, uses 4 KB on x86 processors and 8 KB on Alpha processors [127]. In all instances this is driven by the memory management unit which is now part of the processor.) Therefore, for precedence testing, the virtual-memory subsystem will bring in a page for one or two values. Worse, it will evict some other page, which may still be of value, for just this purpose. We do not believe this will work well because there is no spatial locality.

To make this more concrete, consider computing the greatest-concurrent set for a non-maximal event. To do this, we must first compute the least successor. This will require, in the worst case, precedence tests against $\log_2 1000$ events per trace. This corresponds to about ten-thousand pages of virtual memory. However, we are not done yet. This has simply computed the least successor. We must now examine the preceding events of this slice. We must compare those events, and their non-trace covering events, to our given event. Assuming the one-partner environment, this will require another two-thousand pages of virtual memory. In other words, we require about 48 megabytes of memory just to compute the greatest-concurrent elements of an event.

We now consider the cache implications of the Fidge/Mattern precedence test. Processor cache, like virtual memory, presumes spatial and temporal locality. The cache's existence is predicated on temporal locality. Spatial locality is supported by bringing into the cache a line of up to 32 words, rather than just the word requested. Some specific examples are the Pentium III with a level-one cache size of 1000 lines, each of 32 bytes [102], the Athlon with a level-one cache size of 2000 lines, each of 64 bytes [102], and the Pentium IV with a level-two cache of 2000 lines, each of 128 bytes [103]. In no case does the entire timestamp vector fit into a cache line, nor do we advocate that it should. However, this means that even when we are determining concurrency, and so need two elements of the vector, there is less than a 3% (32/1000) chance that the other element would have been brought in by spatial locality.

Consider again the computation of the greatest-concurrent set for a given non-maximal event e. The vector element $\mathcal{FM}$[e.eventID.TraceID()] will experience good temporal locality, being used up to ten-thousand times to compute the least-successor set. However, none of the events being compared with will experience any spatial or temporal locality at all. The entire data cache will in effect be flushed between five and ten times, with the resulting cache containing data of no value whatsoever. The effect will be as if this entire computation were run from main memory. We now require about a thousand tests against the remaining elements of $\mathcal{FM}(e)$ interleaved with an equal number of tests with non-trace events covering the potentially greatest-concurrent events. Here we will achieve some spatial and temporal locality. Specifically, the vector timestamp of our given event should have good spatial locality. Any covering events that are concurrent appear to have some temporal locality, since they may also be tested elsewhere in the loop for precedence against our given event. However, that locality appears to be illusory. The reason is that the two tests require different vector elements. Thus, as above, there is only about a 3% chance that the correct vector element is present.

All of this analysis of course ignores the fact that we cannot really dedicate the entire cache to vector timestamps. Indeed, the fixed-size portion of any event must first be examined to acquire the timestamp, and that portion will require (and nicely fit in) a cache line. Therefore the effective cache size, as regards the vector timestamps, is really half the actual size.

The cost estimation for a given precedence test depends on where the relevant vector elements are located. We believe we have fairly thoroughly demonstrated that locality in the cache is very limited, and this is likely the case for main memory for essentially the same reasons. The cost is then going to be a function of the size of main memory. If we assume that our main memory is even half the size of our data structure (observe that this implies one to two gigabytes of main memory, dedicated just to the vector timestamps), and we assume precedence tests between random events, then there is a 75% chance that one of the timestamps will currently be swapped out. Whenever we need to read in a page of virtual memory, we can assume about six orders-of-magnitude delay over main-memory operation. This implies that the cost of precedence testing will be about 75,000 times worse amortized than if the data structure fitted in main memory. Even if we assume a 95% hit rate on the main memory portion of the data structure, we will still be nearly 10,000 times slower amortized than running entirely out of main memory.

In summary, virtual memory and processor caching do not appear to work well with Fidge-Mattern timestamps when they are used in this mode. We note in passing that Ore timestamps would appear to have very good spatial locality for precedence testing. In particular, if the Ore timestamp is no bigger than the cache-line size (in the case of the Pentium IV, that implies a realizer with up to 32 extensions), then the entire precedence-test can take place in cache. Since loading the cache is a significant portion of the cost, this implies that the test would not in practice be d-times as slow as the Fidge/Mattern test, where d is the size of the timestamp. We will elaborate on this point in Chapter 8. The Summers cluster timestamp can likewise benefit from this locality. We will explore this point further in Chapter 9.

Finally, we note that we have provided a thought experiment, but no actual data. This is, at least in part, because we know of no system that uses this technique and is capable of processing the size of data suggested in our experiment. However, it would useful to record the use of tools such as POET to see the sequence of operations, and determine the implications of those operations. This is currently a hard problem, as the code is not set up per our partial-order data structure. It is therefore non-trivial to discern what high-level operations are performed at any given point. Such records might be useful to explore the concept of a data-trace cache, analogous to an instruction-trace cache [117].

7.1.2 CALCULATE ON DEMAND

We now turn to the alternate possibility of calculating the Fidge/Mattern timestamp on demand. Clearly we would not wish to calculate the entire set of timestamps from the beginning of the computation up to the required timestamps on every occasion that a timestamp was required. This implies that we would develop our own caching scheme. There are at least three ways in which this might be done. We could record every k^{th} timestamp, we could strategically record key timestamps, or we could simply cache timestamps as they are required, on the assumption of temporal locality.

Recording every k^{th} timestamps on each trace provides a clear time/space tradeoff, reducing the required space by a factor of k. However, the larger the value of k, the greater the number of timestamps that must be computed to determine our required timestamp. For the single-partner environment, with $k > 2$, the worst-case is unbounded. That is, we may have to compute a timestamp for every event that is causally prior to our given event, other than those for which timestamps are recorded. The probability of this worst case occurring is a function of the size of k. The larger the value of k, the greater the likelihood of its happening. We have no data, and are aware of no study, as to what these probabilities are for a given value of k. It is possible that such data can be gleaned from work on distributed checkpointing, which, for certain algorithms, faces a similar problem [111].

The average number of timestamps that must be computed for a computation consisting entirely of unary events is $k/2$. Since each timestamp computation costs $O(N)$, where N is the number of traces in the computation, the cost of a precedence test would then become $O(kN)$. While we recognize that if we have a sequence of unary events we could compute the precedence test in constant time, as only one vector element will differ between the recorded and the computed timestamps, the scenario is clearly artificial, and intended to provide an approximate lower bound on the average behaviour. Specifically, no such optimization would be available when receive and synchronous events were present. Further, the average number of timestamps that would need to be computed would only increase from this unary-only scenario.

If we wish to avoid an unbounded timestamp computation, we must strategically record key timestamps. This is difficult if we wish to significantly reduce the number of cached timestamps. The obvious strategy of caching every receive- and synchronous-event timestamp will only produce a factor of two or three in space saving. This is based on the observation that for every receive there must be a corresponding transmit, and unary events are unlikely to dominate. This would reduce the space-consumption of our thought experiment to something like a gigabyte. While useful, it is hardly going to solve the problem.

A more-sophisticated strategy requires recording timestamps for consistent cuts of the partial order in which no messages are outstanding. This is the same requirement as is imposed on no-message-logging checkpointing (Section 3.6). Unfortunately, there is no guarantee that such cuts exist in the partial order. We know of no tool that has taken this approach, possibly in part because of the difficulty of detecting the requisite cuts and the lack of guarantee of their existence. This approach has the same lower-bound cost, $O(kN)$, on average cases as the previous example. It should have better average cost, if the appropriate cuts are present in the partial order.

The POET system takes a somewhat different approach, that merges these two techniques. Rather than caching the timestamps themselves, it caches the state of the timestamper. This state is written to disk at regular intervals (defined by the CHKPT_INTVL parameter). When an event needs to be timestamped for which no appropriate information is currently available in memory, a checkpoint of the timestamp state near, but prior, to the event is read from disk. Timestamping then continues from that state until the required event is reached and timestamped. This prevents the unbounded-computation problem, though at the expense of various disk accesses to seek the requisite timestamper state. Note that this is not strictly a cache, since is written to disk. However, such an approach could be used as a caching strategy.

The third approach is to cache on use. While this suffers from the same potential unbounded-timestamp computation problem as recording the timestamp of every k^{th} event, it has the possible advantage of temporal locality. Based on our thought experiments, we are unconvinced that significant temporal locality exists. This may then become a liability rather than an advantage, as cache entries are flushed to make way for new timestamps that may well not be used more than once. That said, some temporal locality does exist, particularly when calculating precedence-related event sets. It is useful to exploit that locality when building a system that does not incorporate such capabilities, but rather operates at the event-level with Fidge/Mattern timestamps. The cost of precedence tests under the cache-on-use policy is $O(N)$, since it too must calculate the requisite timestamps. As is always the case with caching, the size of the constant will depend on the size of the cache and the previous behaviour.

7.1.2.1 CASE STUDY: POET

Having examined the behaviour of Fidge/Mattern timestamps through thought experiments, we now study some actual cases in the POET system. We first describe the POET caching mechanism. We then detail the experiments we ran. Finally, we provide results and analysis.

The POET dbg_session process has a cache-on-use policy, resorting to reading timestamp state from disk only when the cache information is insufficient. It maintains NUM_CACHE caches (default, five), each containing a timestamp state and the last NUM_EVENTS timestamped. A timestamp state is a set of Fidge/Mattern timestamped events, with at least one in each trace, such that any successors to those events may have Fidge/Mattern timestamps computed *per* Equation 6.4. Whenever a timestamp state is restored from disk, it will clear the least-recently-used cache and use that cache for timestamps generated thereafter. This means that no single timestamp state dominates the cache, as would be likely with a single cache. This in turn allows temporal locality to exist across a larger interval. The number of caches is akin to the associativity level of a processor cache. The events stored within the cache are the last NUM_EVENTS timestamped, accessed first by TraceID, and then in a linked list within each trace.

When a request is made for a timestamped event, each cache is first checked. This takes O(NUM_CACHE) steps. If the event is present in one of the caches, it will then be found in time proportional to the number of events cached for that particular trace. If events are evenly spread over the traces, this will be O(NUM_EVENTS/num_proc), where num_proc is the number of active traces in the computation. The default value for NUM_EVENTS is 250, and for such a small cache, linked-lists are probably reasonable. If the event is not present in any cache, then the cached timestamp-state that is closest, and prior, to the event is selected for use in computing the timestamp of the event. However, if the timestamper can skip more than one event per process, on average, it will not use this selected state, but will restore one from disk. Likewise, if all cached states of the timestamper are beyond the sought event, state must be restored from disk. The timestamper is then allowed to run until the required event is timestamped. The expected execution time is then $O(N)$ amortized.

The purpose of this case study is then to provide some idea of the constants that are involved in the caching operations and to demonstrate the proportionality between execution time and timestamp size. Two different machines were used for the experiments, both running RedHat Linux 7.1, with the 2.4.2-2 kernel. The first machine was a ThinkPad 390E, with a 333 MHz Mobile Pentium II processor, 64 MB of memory and a 6.4 GB fixed disk, of which about 3.9 GB were dedicated to the ext2 file-system and 128 MB were used for swap space. The second machine was a ThinkPad A22m with an 800 MHz Pentium III (Coppermine) processor, 256 kB of cache, 192 MB of memory, and a 10 GB fixed disk, of which about 3.5 GB were dedicated to the ext2 file-system and 256 MB were used for swap space.

For the experiments three different execution histories were examined: a PVM implementation of Life, a Java Dither program, and a Java web server. The implementation of Life was over a torus, which was divided equally in fixed bands to the various processes. Each process was given initial information from the master process. They then communicated with their neighbouring-band processes in each round of Life. After all rounds were executed the results were returned to the master. There were a total of 39 rounds, executed by 128 processes, resulting in 31,098 events, all asynchronous or unary. The Java language was instrumented such that each object was represented by its own trace. The Dither program, from the Java sample suite, required 297 objects (thus, traces) and produced 102,371 events, all synchronous or unary. The web server required 175 objects and produced 9,157 events, mostly synchronous, but some asynchronous.

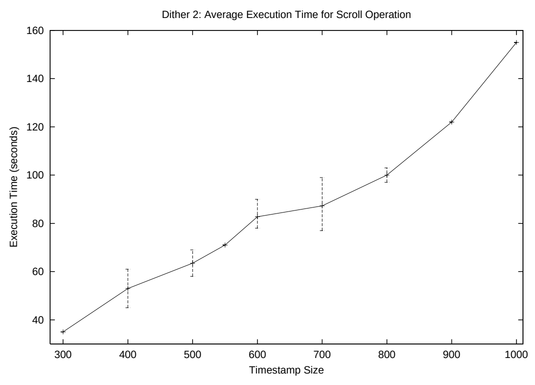
The experiments performed were the following. The execution history was loaded into POET and the system was allowed to quiesce. This is important, as, especially for the larger histories, the system would take some time (on the order of minutes) to finish timestamping the history and writing checkpoints to disk. After this, the statistics were printed. The POET statistics give an indication of the number of timestamped and raw events requested, and how many are satisfied by cache. The first event on the last trace displayed was then left-clicked (in the case of Life there was no first event shown, so the chevron was left-clicked). This input causes POET to initiate a scrolling operation, dragging as many events as can be displayed on that last trace as possible, subject to the constraints of the partial order [143]. The purpose of this particular choice of operation is that it requires a significant number of precedence tests and the computation of various greatest-predecessor and least-successor sets. It thus represents the closest thing that POET has to a plausible sequence of partial-order operations. After the operation was completed, the statistics were printed again.

The following parameters were then varied. First, we varied NUM_PROC, the maximum number of traces POET will accept, from 300 to 1000, and measured the wall-clock execution time of the scrolling operation. We note at this point that POET uses a fixed-size vector, of size NUM_PROC, though with caveats. Most code on that vector will operate only over the range of active traces, define by num_proc. A very small fraction, however, must use the entire vector. In particular, timestamping events has to operate over the entire vector, if only to zero out those portions that are not used. This experiment is then intended to capture some of the effects of larger timestamps, though clearly without capturing the full effects. The parameter changes were performed under the compilation options -g, -pg, -O1, and -O2. When capturing profiling information, our null operation was to load the execution history and then exit. This allows us to look at the execution profile of just the scrolling operation.

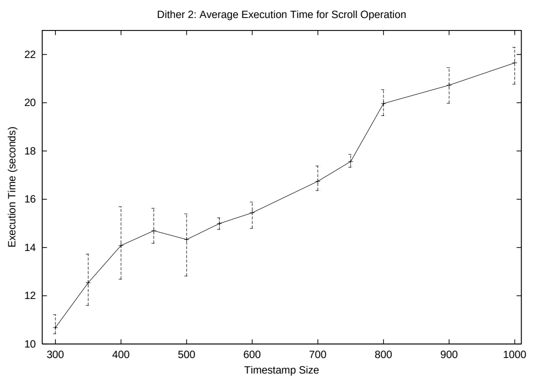
We then varied NUM_EVENTS from 10 to 10,000, running only under the -pg option. We were primarily interested in the change in the statistics and profile for such operations. This was performed with NUM_PROC set to 300 for Dither and 150 for Life. We then examined the effect of raising NUM_PROC in this case.

We now present the results for these various experiments. We start with a graphical display of the execution time versus vector size, shown in Figure 7.1. The graphs on the left are for the 333 MHz machine, while those on the right are for the 800 MHz machine. The horizontal axis is vector size, while the vertical axis is wall-clock execution time (measured with an accuracy of around 200 ms). Note that the scales vary by graph. The particular results shown are based on using the -g compilation option. Use of different compilation options varied the numbers on the vertical axis, but not the essential shape of the graph. While we would not wish to argue that these graphs display a strictly linear relationship, a linear regression analysis does yield correlation coefficients of between 0.97 and 1.00. We can therefore reasonably assert that the proportionality of execution time to vector size holds in real systems.

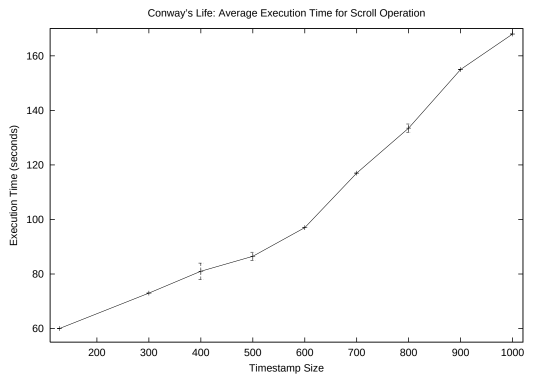
To put the execution times in context, we ran the timestamper over all events. This provides some form of estimation of the total execution time if the events could be timestamped and held in memory. These experiments were performed only on the 800 MHz machine, and only for



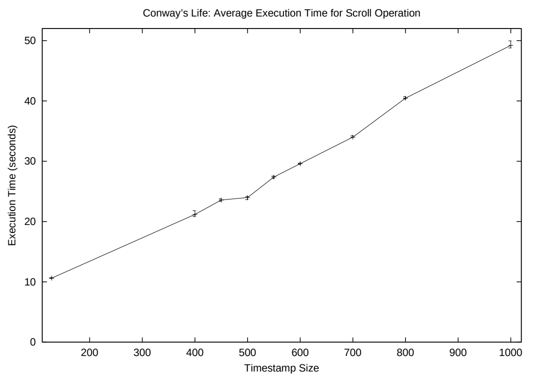
(a) Dither 333



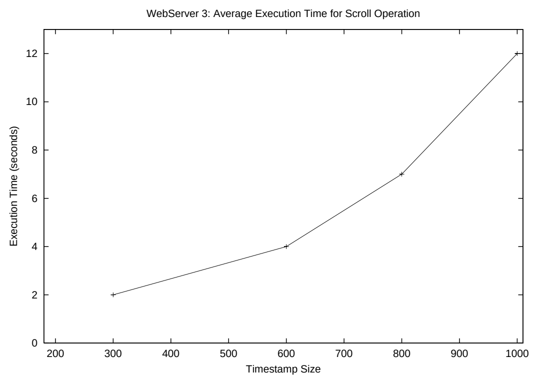
(b) Dither 800



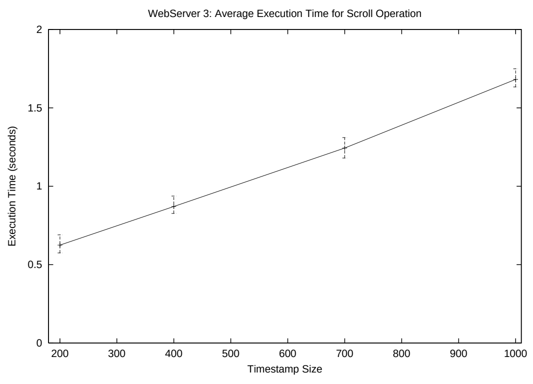
(c) Life 333



(d) Life 800



(e) Web 333



(f) Web 800

Figure 7.1: Execution Time v. Vector Size (Note: Scales Differ)

NUM_PROC	Dither Time (s)	Life Time (s)
300	1.28	0.213
600	1.43	0.293
900	1.56	0.392

Table 7.1: Cost of Linear Timestamping

Dither and Life. The results are shown in the Table 7.1. Note that there appears to be little correlation between these execution times and the scrolling time. That is, while both increase as the vector size increases, the time required to timestamp Life's thirty-thousand events is around a fifth of that required to timestamp Dither's one-hundred-thousand events, while the scrolling time required for Life can be more than twice that of Dither. This is because the scrolling operation for Life requires more timestamps than does that for Dither, and since those timestamps have to be computed by POET if they are not in cache, it takes longer.

Note also that the time per event timestamped is between 20% and 80% higher for Dither than Life. This is in part because the maximum in the Fidge/Mattern operation is only over actual traces, of which there are 128 in Life and 297 in Dither. It is also partly a result of a different mixture of event types, which will require somewhat different execution time to timestamp (for example, the unary is simply a copy of an existing timestamp with one entry incremented).

The most significant aspect of this table, however, is that all of the numbers are one or two orders-of-magnitude smaller than the scrolling time. Profile analysis of the scrolling experiments confirms that the majority of their execution time (greater than 90%) was taking place in timestamping-related functions. This shows why it would be desirable to maintain the entire data structure in memory, and how significant the cost is of not being able to do so.

These experiments alone demonstrate that Fidge/Mattern timestamps, at least as they are used in POET have a scalability problem as the number of traces increases. Further, they demonstrate that that scalability problem is not simply a size issue, but that it affects the execution performance of the POET system. It is also reasonable to believe, based on these results and the fact that POET can handle a larger partial order than any other extant system, that this problem extends to the Fidge/Mattern timestamps themselves, and is not an artifact of the POET design or implementation.

The question then arises as to whether larger caches will help and, if so, to what degree. The quick summary is that our caching experiments indicated that it can help, and in some cases it can help significantly. However, it cannot remove the proportionality and, as the timestamp becomes larger, the effective cache size shrinks. Our experiments were as follows. First, we varied the value of NUM_EVENTS from 10 to 45,000, holding NUM_PROC at 300 (150 for Life). The execution times for the scrolling operations are shown in Table 7.2 and Figure 7.2. The Dither tests were performed on the 800 MHz machine and the Life ones on the 333 MHz machine.

Both the table and the graph illustrate that the cache size is very significant, but that there is a point beyond which no further benefit accrues, and things can actually get worse. This is not remarkable. First, no system can cache what it does not yet have. In the case of POET caching, some timestamps must be computed. Second, at some point the memory consumption of the

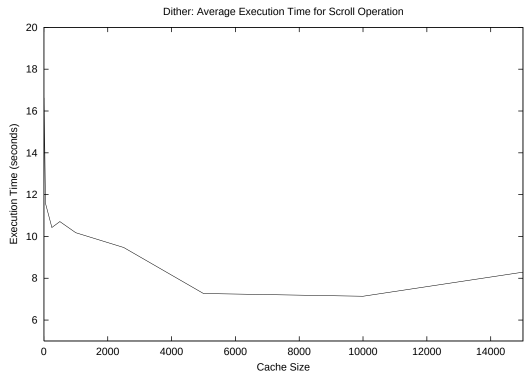
NUM_EVENTS	Dither Time (s)	Life Time (s)
10	16.63	122.6
50	11.59	112.3
250	10.43	27.23
500	10.71	19.12
1000	10.18	18.19
1500		11.21
2000		7.53
2500	9.47	6.68
3000		6.58
5000	7.27	6.21
6000		4.21
10000	7.14	2.52
15000	8.29	
25000	11.85	1.84
35000		1.67
45000		2.45

Table 7.2: Benefits of Caching

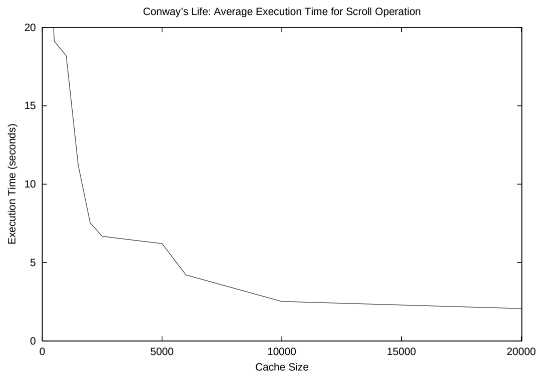
cache begin to conflict with the memory requirements of the remainder of the system, and paging begins to happen. This is where the problems now occur. As the vector size increases, for a given size of memory, the number of events that can be stored productively in the cache decreases. More simply put, the reason the Life results are so good at a vector size of 10,000 is that the value of NUM_PROC is 150. It thus consumes only 30 MB of memory. When we increase that to the default 300, the execution time increases to 4.61 seconds, and at 500 it is 28.35 seconds (with significant disk activity for the duration). Again, this is not surprising, as what is supposed to be cache is now executing in virtual memory. The cache alone consumes about 100 MB, on a 64 MB machine.

Examination of the statistics information allows us to estimate the number of additional timestamps that must be computed per event timestamped. This can be determined as essentially all raw-event requests will be made by the timestamper. We can then compute two different numbers. First, we can determine the average number of events that must be timestamped to timestamp an event that is not in the cache. This varies in Life from 250 with a cache size of 10, to 238 with a cache size of 1,000, to 320, with a cache size of 10,000, and 640 with a cache size of 45,000. At the higher end of the range the numbers are invalid, as the raw events are being requested for other purposes. In that range, few (less than 100) events are being timestamped, as opposed to being found in cache, and so the results are skewed. This then corresponds to about 2 events per trace that need to be timestamped. For Dither, it varied from 1643 at cache-size 10 to 1302 at cache-size 1,000. This corresponds to 4 to 5 events per trace that need to be timestamped.

The alternate, and perhaps more meaningful calculation, is to determine the average number of events that need to be timestamped per timestamp-event request, whether or not the timestamp



(a) Dither 800



(b) Life 333

Figure 7.2: Execution Time v. Cache Size (Note: Scales Differ)

is found in the cache. This gives us an amortized cost for timestamp computations. In this case, Life varies from 227 with a cache size of 10, to 19 with a cache size of 1,000, to 5.8 with a cache size of 10,000 (though we suspect that it is actually lower, per the point above regarding raw-event requests). For Dither, the results varied from 1,439, at cache-size 10 to 668 at cache-size 1,000 to 464 at cache-size 10,000. The cache hit-rate reached 97% for Life but only 61% for Dither, both at a cache-size of 10,000. This, in large part, explains the substantially better performance produced by the cache-size increase for the Life results versus the Dither results.

Finally, we looked briefly at the effect of changing the checkpoint interval from the default of 200 to 1,000 and then to 10,000. The primary effect was to substantially reduce the size of the checkpoint file, and thus substantially speed up the point at which quiescence was reached. This is not surprising. The significant effect from the caching perspective, is that it should, and did, result in a greater number of timestamps that must be computed to timestamp a given event. Thus, for example, when the checkpoint interval and cache-size were 10,000, Life required an average of 415 events to be timestamped per timestamp request, where it was only 320 at the default checkpoint interval of 200. The execution time increased to 3.2 seconds as a result. Similar results occurred across the range of options that we evaluated.

7.2 Theoretical Objections

Having provided both theoretical and empirical justification that Fidge/Mattern timestamps do not scale well as the number of traces increases, we now consider their theoretical justification. Specifically, it has been argued that these timestamps are optimal [16], and therefore, whatever their scalability properties, we cannot achieve better. We now address these points in turn.

There are two justifications for vectors of size equal to the number of traces. First, to characterize causality in a partial order it is necessary to have a vector (or equivalent) of size equal to the dimension of that partial order [114]. Further, the dimension of a partial order can be as

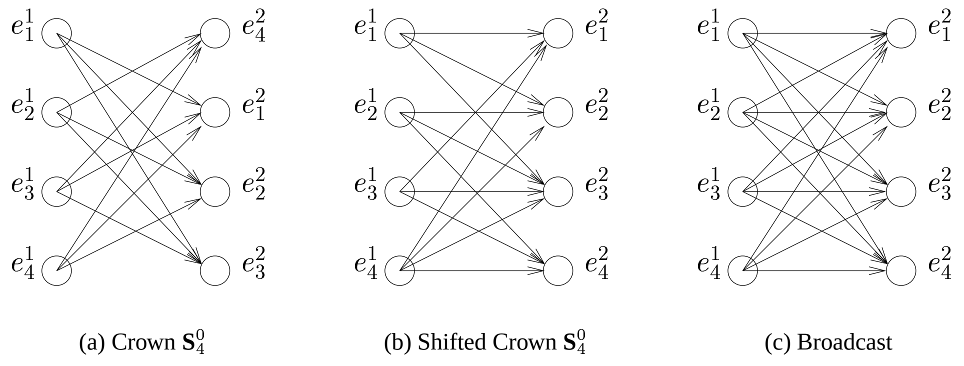


Figure 7.3: Crown $\mathbf{S}_4^0$and broadcast partial orders

large as the width [155]. Crown $\mathbf{S}_N^0$ is the standard example of such a partial order, and is shown for $N = 4$ in Figure 7.3(a). By rotationally shifting the rightmost elements ($\forall_i\ e_i^2$) of this partial order up by one position, we create the distributed computation shown in Figure 7.3(b). Such a computation has dimension equal to its width, and thus requires vectors of size equal to its width to capture causality.

This is an unusual computation in that it requires both multicast and multi-receive. While this computation does not violate our model of distributed computation, and both of these operations do have real systems counterparts, it is important to emphasize that neither are necessary. Charron-Bost [16] translated crown $\mathbf{S}_N^0$ into a point-to-point distributed computation, and demonstrated that this computation also has dimension N. Given this, Fidge/Mattern timestamps appear to be optimal.

There is, however, a very significant limitation in this proof. The fact that some distributed computation exists with dimension equal to the number of traces, does not imply that all, or even most, distributed computations will have such a dimension. Indeed, both crown $\mathbf{S}_N^0$ and the Charron-Bost computation correspond to all processes sending a message to all other processes, with the exception of their left neighbour. This is not a realistic distributed computation. The more likely computation is for each process to broadcast a message to all other processes, as shown in Figure 7.3(c). Indeed, even if it were desired that each process send a message to all other processes except their left neighbour, it is likely that broadcast would be used, together with a note that the left neighbour should ignore the message, since that is likely more efficient. The dimension of such a broadcast is two, regardless the number of traces. As we shall show in the next chapter, and have demonstrated elsewhere [164, 165], a significant number of distributed computations, with up to 300 traces, have a dimension much smaller than the number of traces. We thus exploited this limitation in our first solution.

The second justification is more fundamental. Garg and Skawratananond have recently shown that to compare consistent cuts you need vector clocks of size N for all distributed computations [52]. This then begs an important question: what does it mean to compare cuts? More fundamentally: what does it mean to characterize causality? What it appears to mean is that there is some entity associated with an event (respectively, cut) that can be compared with that same entity associated with another event (cut) to determine the causal relationship between the two. No intermediary needs to be involved in the calculation. This is an important property in a distributed computation, where no intermediary would be accessible without additional communication. However, our data structure does not exist within the distributed computation. It is centralized in the monitoring entity. It is therefore not unreasonable to access other events as required to determine causality. Indeed, the most-primitive precedence algorithm that we discussed was the graph searching, which must access every event on the path between the two being compared, and likely several others. We exploited this capability in both of our solutions.

7.3 CURRENT TECHNIQUES FOR SPACE-REDUCTION

We close this chapter with a brief survey of various techniques for reducing the space-consumption required by vector timestamps, and a short justification of their inapplicability to our problem. It should be recollected that much of this work on logical-time algorithms presupposed their use in distributed computations, and as such they are either not directly applicable or inefficient in our environment. In particular, while many information-compression techniques are applicable, or at least worth studying, in that distributed-timestamp context, they will not in general be applicable to the creation of a partial-order data structure.

The most obvious approach, and much studied, is to perform differential encoding of the timestamps. In the case of a unary or a transmit event, the Fidge/Mattern algorithm changes only a single vector element. It is clearly possible to look at storing just that change, rather than the entire vector. Singhal and Kshemkalyani [123, 136] have a solution for the distributed environment, but it is not directly applicable as it requires an $O(N^2)$ matrix at each process to determine what has changed since the last communication, and thus what to transmit. Demaine performed a number of experiments on execution histories, exploring a variety of differential encoding techniques for the POET data structure, including storing the difference between timestamps, and the difference of the difference [32]. His conclusion was that it would achieve at most a factor of two or three in space saving. This is consistent with the observation that for every transmit, there must be a receive, and receive events will not experience significant space-saving under differential encoding. Indeed, if associative arrays are used, their space-consumption may increase.

Most of the remaining work that focuses on timestamps tends to be more specialized. Golden studied the problem of dealing with dynamic process creation and termination [124]. In the Fidge/Mattern approach, processes that cease to exist will maintain their vector location indefinitely. This is undesirable, especially when we move from processes to traces, and model languages such as Java with one trace per object. In these cases, many traces will exist over the course of a computation, but most will be short-lived. Unfortunately, Golden's solution is premised on using the timestamps within the system, and it is not clear how, or if, his solution might be adapted.

In very recent work, Christiaens and De Bosschere also address this issue by developing accordion clocks [22], which grow or shrink as required. However, their domain was data-race detection in multi-threaded Java applications. Their algorithm removes timestamp locations associated with terminated threads when it is determined that no events in the thread can be involved in a data race. More precisely, once the last event in a terminated thread is causally prior to all future events, the vector location for that thread is no longer necessary. The results for their particular application show a reduction in size to a constant for two of their three target programs, and a substantial reduction in the third.

The formal condition for accordion clocks can, in principle, be readily adapted to the POET data structure. Any trace which has terminated, and this must be recognized as more than simple silence on the part of the trace, can have its vector location used by a new trace at such time, if ever, as the last event is causally prior to all extant maximal events. Unfortunately, the "if ever" is likely to be never. The POET target environment that would most benefit from this is Java, and it has an asynchronous garbage collection thread. As instrumented, this thread contains events that are concurrent to all other threads. The condition could therefore never hold. Further, it must be emphasized that even if we are able to create a solution that more efficiently handles trace termination, we still require a system that can scale to large numbers of traces, since there are environments for which the traces are both large in number and long-lived.

A number of approaches have been developed that, like Lamport time, are space-conserving, but cannot completely determine precedence. These include plausible clocks [152, 153], and statistical or formulaic encoding of the data [174]. While useful for their chosen domains, they are inapplicable in our environment as it is not currently clear what the implications are of distributed-system-observation tools that did not completely capture precedence. We believe, however, that this might be an area worth exploring, as it is clear that no monitoring system can capture all precedence channels [19].

Frumkin, Hood, and Yan claim that entropy-efficient encoding of program traces can result in a significant reduction of trace sizes [48] (a factor of five), though this appears to be a theoretical analysis, rather than an actual tool, and is focused on the disk-storage requirement, not the in-memory data structure. Yan, Sarukkai, and Mehra also looked at trace-file size reduction [175].

Meldal, Sankar, and Vera developed a mechanism for determining temporal ordering of messages arriving at the same process, where the process interconnection is static and known ahead of time [105]. This is sufficiently restrictive that it is doubtful that it would be of value in our work.

Finally, the graph-theory community has developed various algorithms for maintaining the dynamic transitive closure of digraphs [1, 77, 78, 178]. There are several differences from our work. The primary difference is that they wish to be able to insert or delete edges in arbitrary locations in the digraph. For our purpose, it is sufficient to limit edge insertions to sink-only vertices and deletions to source-only vertices. This corresponds to inserting only maximal elements in the partial order and deleting only minimal elements. In other words, we can restrict the processing order to be a linearization of the partial order. A second important difference is that they wish to be able to answer the precedence query in constant time. We are willing to accept a tradeoff on this if it enables scalability. As a result of these differences, their best algorithms require space equal to that of the Fidge/Mattern timestamp, and insertion time proportional to the number of elements in the partial order. This insertion-time alone is unacceptable for our purpose.

PART III

SOLUTIONS

8 DIMENSION-BOUND TIMESTAMPS

We have demonstrated that Fidge/Mattern timestamps do not scale well for our application, and that there is no theoretical requirement that demands their use. We have not yet, however, offered an alternative. We now provide two possible alternatives, significant variants of the Ore and Summers cluster timestamps respectively. Both satisfy our requirements, as identified in Chapter 6. Both require centralized creation and a somewhat centralized precedence test. By "somewhat centralized" we means that there are instances when the partial-order data structure will have to be consulted, and other instances in which comparison between the timestamps alone is sufficient.

We start with the Ore-variant, which we have termed dimension-bound timestamps as their size will depend not on the number of traces, but on the dimension-bound that the algorithm is able to achieve, which can be as low as the dimension of the partial order. To justify a dynamic variant on the Ore timestamp, we must first demonstrate that it will be of value. Specifically, we must show that a significant number of distributed computations have a low dimension relative to their width (that is, their number of traces). If this were not the case there would be no value in an Ore timestamp. We provide such evidence in Section 8.1. We then present a three-phase dynamic-Ore algorithm, where the phases are interleaved. We provide solutions for each phase, though we have not implemented the complete scheme. Finally, we analyse the resulting timestamp.

8.1 BOUNDING THE DIMENSION

In this section we will describe the algorithms we have developed to compute an upper-bound on the dimension of the partial order. This is done in a two-phase process. We first compute the critical pairs of the partial order, and then we create a set of extensions that reverses those pairs. We will first describe the formal justification for this approach, and then describe the two phases. Finally we will provide some analysis of the algorithm. This work comes from our offline and online dimension-bound-analysis papers [164, 165].

Computing the exact dimension of a partial order is known to be NP-hard for any partial order of dimension greater than two [177]. We therefore approached the problem by attempting to simply bound the dimension. For our purposes, an order-of-magnitude difference between the dimension bound and the number of processes would be sufficient to justify proceeding. It was then necessary to develop an algorithm to achieve a reasonable bound in a reasonable amount of time. Rather than take the direct approach of generating linear extensions and then determining if they formed a realizer we chose an indirect route based on critical pairs (Definition 9, page 11). As we know from Theorem 1, the dimension of a partial order is equal to the least number of subextensions of that partial order required to reverse all of its critical pairs.

Three observations must be made about this. First, it is sufficient to simply reverse the critical-pair events. Specifically, all events that are not part of a critical pair may be ignored. Second, not every subextension need contain all critical pairs. Third, it is not necessary for the subextensions

```
1: $\forall_{e^j}\ e^j \in \mathcal{E}$ {
2:     $l_C \leftarrow \mathrm{leastConcurrent}_{\mathcal{E}}(e^j)$
3:     $\forall_{e^i}\ e^i \in l_C$ {
4:         $g_C \leftarrow \mathrm{greatestConcurrent}_{\mathcal{E}}(e^i)$
5:         if $(e^j \in g_C)$ {
6:             $(e^i, e^j) \in \mathrm{CP}$
7:         }
8:     }
9: }
```

Figure 8.1: Critical-Pair Computation

to be linear. They merely have to reverse the critical pairs. Given this, the approach we have taken is to first compute all of the critical pairs of the partial order (a polynomial-time problem) and then create extensions that reverse these critical pairs (an NP-hard problem).

8.1.1 Computing Critical Pairs

While it is possible to use the given definition of critical pairs to compute the set of all critical pairs, any such algorithm would likely be very inefficient. To achieve reasonable performance in the computation it is necessary to develop an association of critical pairs with the relations that hold for the partial order. To this end, we defined the sets $\mathrm{leastConcurrent}_{\mathcal{E}}(e)$ and $\mathrm{greatestConcurrent}_{\mathcal{E}}(e)$ (see Equations 4.4 and 4.5 in Section 4.3) and discovered their relationship to critical pairs, as expressed in the following theorem.

Theorem 4 (e^i, e^j) *form a critical pair if-and-only-if*

$$e^i \in \mathrm{leastConcurrent}_{\mathcal{E}}(e^j) \wedge e^j \in \mathrm{greatestConcurrent}_{\mathcal{E}}(e^i)$$

Proof: Necessary: Assume $(e^i, e^j) \in \mathrm{CP}_{\mathcal{E}}$. If $e^i \notin \mathrm{leastConcurrent}_{\mathcal{E}}(e^j)$ then, because $e^i \parallel_{\mathcal{E}} e^j$, $\exists_{e^k}\ e^k \prec_{\mathcal{E}} e^i \ \wedge\ e^k \parallel_{\mathcal{E}} e^j$. This implies that $\exists_{e^k}\ e^k \prec_{\mathcal{E}} e^i \ \wedge\ e^k \not\prec_{\mathcal{E}} e^j$ which contradicts $(e^i, e^j) \in \mathrm{CP}_{\mathcal{E}}$. Likewise, if $e^j \notin \mathrm{greatestConcurrent}_{\mathcal{E}}(e^i)$ then $\exists_{e^k}\ e^j \prec_{\mathcal{E}} e^k \wedge e^k \parallel_{\mathcal{E}} e^i$. This then implies that $\exists_{e^k}\ e^j \prec_{\mathcal{E}} e^k \ \wedge\ e^i \not\prec_{\mathcal{E}} e^k$ which again contradicts $(e^i, e^j) \in \mathrm{CP}_{\mathcal{E}}$.
Sufficient: Assume $e^i \in \mathrm{leastConcurrent}_{\mathcal{E}}(e^j) \ \wedge\ e^j \in \mathrm{greatestConcurrent}_{\mathcal{E}}(e^i)$. Show $\forall_{e^k}\ e^k \prec_{\mathcal{E}} e^i \Rightarrow e^k \prec_{\mathcal{E}} e^j$. If $e^k \prec_{\mathcal{E}} e^i$ then $e^k \not\parallel_{\mathcal{E}} e^j$ because $e^i \in \mathrm{leastConcurrent}_{\mathcal{E}}(e^j)$. Also, if $e^k \prec_{\mathcal{E}} e^i$ then $e^j \not\prec_{\mathcal{E}} e^k$ since if it did, by transitivity $e^j \prec_{\mathcal{E}} e^i$ which contradicts $e^i \in \mathrm{leastConcurrent}_{\mathcal{E}}(e^j)$. Therefore $\forall_{e^k}\ e^k \prec_{\mathcal{E}} e^i \Rightarrow e^k \prec_{\mathcal{E}} e^j$. Likewise we may show $e^j \prec_{\mathcal{E}} e^k \Rightarrow e^i \prec_{\mathcal{E}} e^k$. If $e^j \prec_{\mathcal{E}} e^k$ then $e^i \not\parallel_{\mathcal{E}} e^k$ because $e^j \in \mathrm{greatestConcurrent}_{\mathcal{E}}(e^i)$. Also, if $e^j \prec_{\mathcal{E}} e^k$ then $e^k \not\prec_{\mathcal{E}} e^i$ since if it did, by transitivity $e^j \prec_{\mathcal{E}} e^i$ which contradicts $e^j \in \mathrm{greatestConcurrent}_{\mathcal{E}}(e^i)$. Therefore $\forall_{e^k}\ e^j \prec_{\mathcal{E}} e^k \Rightarrow e^i \prec_{\mathcal{E}} e^k$. $\Box$

This theorem, in turn, enabled the development of the critical-pair computation algorithm shown in Figure 8.1. Some comments should be made about this algorithm. First, it is not obvious from the above why we perform the computation in what appears to be the reverse order. That is, we take each event as a possible second element of a critical pair, rather than a first element. The reason has to do with the relative cheapness with which we can compute the leastConcurrent set versus the comparative expense of computing the $\text{greatestConcurrent}$ set. We implemented our algorithm as a client of the POET server using the algorithms described in Section 6.2.2. The offline paper [164] used the naive algorithm, while the online paper [165] developed the technique that led to the improved algorithm described in that section. Per those algorithms, when an event is non-maximal it is cheaper to compute the leastConcurrent set.

This algorithm is still not ideal, however, as we have to compute the entire greatest-concurrent set for each element of the least-concurrent set. This makes the computations of the critical pairs $O(N^3)$, even under the improved least- and greatest-concurrent algorithms. However, the theorem used to create the optimal greatest-concurrent algorithm has a suitable corollary.

Corollary 1 (Critical-Pair Calculation:) *If two events, e^i and e^j, are concurrent then e^j is an element of the greatest concurrent set of e^i if-and-only-if e^i precedes some immediate successor of e^j.*

$$e^i \parallel_{\mathcal{E}} e^j \Longrightarrow \left(e^j \in \text{greatestConcurrent}_{\mathcal{E}}(e^i) \Leftrightarrow \forall_{e^k}\, e^j <: e^k \Rightarrow e^i \prec_{\mathcal{E}} e^k\right)$$

Proof: Follows directly from Theorem 3.□

Since the algorithm does not compute $\text{greatestConcurrent}_{\mathcal{E}}(e^i)$ until it has already computed $\text{leastConcurrent}_{\mathcal{E}}(e^j)$ and concluded that $e^i \in \text{leastConcurrent}_{\mathcal{E}}(e^j)$, and thus $e^i \parallel_{\mathcal{E}} e^j$ it becomes unnecessary to compute $\text{greatestConcurrent}_{\mathcal{E}}(e^i)$. Rather, a simple check of whether e^i is a predecessor of all immediate successors of e^j is sufficient. If it is, then (e^i, e^j) is a critical pair. Thus lines 4 and 5 of the algorithm change to

```
4:    $l_S \leftarrow \left\{e^k \mid e^j <: e^k\right\}$
5:    if $(\forall_{e^k}\, e^k \in l_S \Rightarrow e^i \prec_{\mathcal{E}} e^k)$ {
```

In the single-partner case, the set of events that are immediate successors to an event has cardinality less than 3. This means that the cost to compute the critical pairs is $O(N^2)$.

8.1.2 Reversing Critical Pairs

Building the minimum number of extensions that reverses the critical pairs of a partial order is NP-hard for dimension greater than two [177]. Instead we propose a reasonably efficient algorithm that will not give an optimal solution, but will provide an upper bound on the minimum number of extensions necessary (that is, the dimension). Insofar as the dimension-bound we compute is small relative to the number of traces, it is a satisfactory tradeoff. To achieve this we developed a two-step algorithm. First we select the desired extension in which we will reverse the current critical pair and then we insert it into that extension.

In accordance with Theorem 1, it is sufficient to develop subextensions that reverse the critical pairs of the partial order. We do not have to insert all critical pairs in all subextensions. We merely have to find one subextension that will reverse the critical pair. Each subextension then is composed of some reversed critical pair events and nothing else. Note that this approach is not sufficient for generating Ore timestamps. It is insufficient as each subextension contains just a subset of the events of the partial order, not all of the events. Further, Ore timestamps require that the extensions be linear. Note also that using subextensions, rather than extensions, has implications on how we can insert critical pairs into the subextension.

To understand the algorithm we must define what it means to insert a critical pair into an subextension. It means that we can add the events of the critical pair, such that they are reversed, and that it violates neither the partial order, nor the additional constraints that the reversal requires. It may also be the case, if the insertion algorithm is not optimal, that a subextension rejects the critical pair even though it did not violate these conditions, but rather violated some aspect of the structure in which the extension was kept.

It is, perhaps, helpful to consider a simple example. Suppose we have a partial order consisting solely of two concurrent events, e^i and e^j. It has critical pairs (e^i, e^j) and (e^j, e^i). Once (e^i, e^j) has been inserted into a subextension, x, that subextension must reflect the constraint $e^j \prec_x e^i$. As such (e^j, e^i) cannot be inserted into that subextension, since it would require the subextension to reflect $e^i \prec_x e^j$.

We say that a subextension *accepts* a critical pair if the critical pair may be inserted into that subextension. A subextension *rejects* a critical pair if it does not accept it. Since a subextension may reject a critical pair, insertion into a subextension may fail. Therefore the first step of the algorithm must have a strategy for selecting an alternate subextension in which to place the critical pair. We can now describe the specific algorithms used for the two steps.

The algorithm we used for the extension-selection step is a simple greedy one. We insert the current critical-pair events into the first extension that will accept the pair. In the case that all current extensions reject the critical pair, we create a new extension, containing no events, that must, by definition, accept the critical pair. The critical pair is inserted into the new extension. Thus, the first-step algorithm is shown in Figure 8.2.

8.1.2.1 Subextension Insertion

For the second step of the algorithm, we had to define a method for inserting critical pairs into a subextension. The initial algorithm that we developed was a greedy one that worked on the principle "place the event before the first event it must precede." While this approach produces some promising results, it also produces some spectacularly bad ones.

We therefore decided to develop an optimal solution to this second step. We maintain a directed acyclic graph for each subextension. To add a critical pair we add the two events in turn and then determine if the graph is still acyclic. If it is acyclic we have accepted and inserted the critical pair. If it is not, we reject the critical pair, and remove the evidence of the addition. This method proved to be acceptable, as we can see in Section 8.1.3.

The data structure for a given node of the DAG representing event e maintains the following

```
insert($e^i$,$e^j$) {
    for (i = 0; i < numberExtensions; ++i) {
        if (insert(extension[i], $e^i$, $e^j$)) {
            return
        }
    }
    create(extension[numberExtensions])
    insert(extension[numberExtensions], $e^i$, $e^j$)
    ++numberExtensions
    return;
}
```

Figure 8.2: Extension Insertion

information: the vector timestamp of e, the sets $\{\lambda \mid (\lambda, e) \in \mathrm{CP}_S\}$ and $\{\rho \mid (e, \rho) \in \mathrm{CP}_S\}$ (where $\mathrm{CP}_S \subseteq \mathrm{CP}$ is the subset of the critical pairs that this subextension S has reversed), and a set of pointers to some of the successors to e in the DAG. The actual successors pointed to will depend on the order in which events are inserted. It is not typically the minimum set of successors needed, but neither is it the full transitive closure.

We now define event precedence between two DAG events as follows.

$$e^i \prec_x e^j \iff e^i \prec_{\mathcal{E}} e^j \ \vee\ (e^j, e^i) \in \mathrm{CP}_x$$

where CP_x is the set of critical pairs that are reversed by subextension x, as defined above. Event e^i precedes e^j in subextension x if-and-only-if e^i precedes e^j in the partial order or (e^j, e^i) is a critical pair that is reversed by this subextension. Note that if events e^i and e^j are in a subextension x and (e^i, e^j) form a critical pair this does not necessarily imply that $(e^i, e^j) \in \mathrm{CP}_x$. The pair (e^i, e^j) is only in CP_x if the subextension x has accepted, that is, reversed, it. A simple example of this case is if the subextension x already contains the critical pair (e^j, e^i).

A second significant aspect of this definition is that it does not capture transitivity. Thus if e^k is an immediate successor to e^i and both are concurrent to e^j, with (e^i, e^j) forming a critical pair, then $e^j \prec_x e^i$ and $e^i \prec_x e^k$ but $e^j \not\prec_x e^k$. Appropriate transitivity can only be captured by traversing the DAG.

We designate a special node `root` with the property that $\forall_{e^i}$ `root`$\prec_x e^i$. The root node enables us to enter the DAG at a single point, rather than, potentially, multiple concurrent points.

The insertion algorithm is then as follows. We traverse the DAG in a depth-first search order, starting at the root, comparing the event being inserted with the current node. We determine if the event equals, succeeds, precedes or is concurrent with the current node. Only one of these must be the case, and if more than one has occurred, we abort the insertion. It has failed. If the event precedes or equals the current node, then it must not succeed or equal any successors to the current node. We therefore set a `mustNotSucceedOrEqual` variable to this effect. If this variable has been set before, and the event succeeds or equals the current node we will abort the insertion.

If the event succeeds the current node, then we may have to add successor pointers to the event from the current node. We have to add a successor pointer from the current node to the event if it precedes any of the successors to the current node, or if it is concurrent with all of the successors. In the former case we will also add a pointer from the event to the successor node that it precedes, and remove the pointer from the current node to the successor node, since there is a path through the event.

We repeat these operations for the successor nodes of the current node, in depth-first order. The `mustNotSucceedOrEqual` variable is kept in the depth-first search stack.

To traverse the DAG there is a `mark` integer associated with each node. Before each traversal we increment the mark value in the root. As we visit a node we set the value of mark at that node to value of the mark at the root. We only visit a node if the mark is less than the root mark value. Since we simply use an integer for this mark, if we ever reach about half-a-billion critical pairs this value will wrap. This defect can be easily fixed when the need arises.

To enable us to undo any changes we make, before we make any change we create a copy of the DAG node. We only do this if we do not yet have a copy. After successful insertion of both events of the critical pair we traverse the DAG completely to commit the changes. That is, we traverse the DAG and delete the copy at any node that has one. If the insertion of the events is unsuccessful, we traverse the DAG and abort the changes. That is, we traverse the DAG and, wherever there was a copy made we restore that copy over the existing node.

This algorithm requires a complete DAG traversal for each critical pair reversed. We therefore subsequently developed a more efficient subextension-insertion algorithm for our online paper [164]. The approach we took is to maintain vector clocks in the subextension, and update them after every critical-pair reversal. The value in this is that we can use the vector clocks to determine if the insertion of the reversal of a critical pair would cause a cycle in the extension, and thus must be rejected. Specifically, if we wish to add critical pair (e^i, e^j), we first determine if the extension implies $e^j \prec_x e^i$. This may be done using the standard Fidge/Mattern precedence test in constant time. As such, before we change anything, we know whether or not the critical pair will be accepted.

In the case that the critical-pair reversal is accepted, we update the vector clocks of any events affected by this new edge in the DAG. The vector-clock update is performed according to the standard Fidge/Mattern algorithm. If (e^i, e^j) is the critical pair that is being reversed, then the set of events that need to have their vector clock updated is $\{e^k \mid e^i \prec_x e^k \ \wedge \ e^j \not\prec_x e^k\}$. The justification for this set should be clear, since these are precisely those events for which the existence of e^j was not known in their vector clocks.

Some observations should be made regarding the size of this set. For a given critical-pair reversal, it can be as large as $n - 1$, where n is the number of events in the extension. This would be the case when the partial order consisted of a chain and one additional event, concurrent to all events in the chain. If the events in the chain are processed first, they will form a single extension, since they will have no critical pairs. The processing of the concurrent event will then require a vector clock update for all of these events. Note that in this case, although $n - 1$ vector clock updates are required for that final event insertion, the previous $n - 1$ insertions required no update

Number of Events	Number of Processes	Number of Critical Pairs	Dimension Bound
15	7	38	2
2687	59	2360	4
3252	66	3044	4
2612	66	2032	3
49791	95	6622	3
3272	96	5906	4
7426	109	10019	6
4028	110	8439	6
7928	112	6969	3
35266	112	6675	4
30048	120	7999	3
9826	178	18464	6

Table 8.1: Dimension bounds for Java

at all, implying an $O(1)$ amortized cost for this case. It is our experience that this type of effect is typical. While the set size can be large for some particular critical-pair reversal, it is small over all events, and can be approximated by a constant. Since acceptance is determined in constant time, critical-pair reversal, and thus sub-extension insertion, will be $O(N)$ amortized.

8.1.3 Results and Observations

We now examine the output of the algorithm, which is our primary interest in it. There are two significant aspects. We wish to look at the dimension bounds produced, and we would like to know how tight those bounds are, given that the algorithm does not produce the optimal bound.

We have executed our dimension-bound algorithm over several dozen distributed computations covering over half-a-dozen different parallel, concurrent and distributed environments and a range of 3 to 300 traces. The environment types are the Open Software Foundation Distributed Computing Environment [44], the μC++ shared-memory concurrent programming language [14], the Hermes distributed programming language [139], the Parallel Virtual Machine [53] and the Java programming language [55]. The PVM programs were a subset Cowichan problem set [171, 172], which conveniently represent many common communication patterns that are seen in scientific computations. They frequently exhibited close-neighbour communication and scatter-gather patterns. The Java programs were a variety of web-like applications, including various web-server executions. The DCE programs were sample business application code. The programs for the remaining environments consisted of a mixture of student-written and sample code with no particular common theme. Various of the raw results are shown in Tables 8.2 through 8.5.

The initial results, as reported in our offline and online analysis papers [164, 165], were very promising. The quick summary is that the dimension bound that we discovered over this range of computations and environments was always 10 or less. For computations with a trace-count

Number of Events	Number of Processes	Number of Critical Pairs	Dimension Bound
45	5	12	3
90	19	27	2
121	20	61	4
249	40	124	3
291	42	164	3
467	42	183	4
297	44	237	4
499	70	443	5
501	72	496	6
833	110	1490	9
817	112	1378	8
928	114	1738	7
902	115	1402	8
1560	159	3579	10

Table 8.2: Dimension bounds for OSF DCE

Number of Events	Number of Processes	Number of Critical Pairs	Dimension Bound
360	12	156	2
1750	12	853	5

Table 8.3: Dimension bounds for μC++

Number of Events	Number of Processes	Number of Critical Pairs	Dimension Bound
1888	125	1323	5
1944	127	1429	5
4164	267	4403	7
l4086	297	21401	6

Table 8.4: Dimension bounds for Hermes

Number of Events	Number of Processes	Number of Critical Pairs	Dimension Bound
138	16	270	3
1338	64	4782	5
2682	128	17759	5
16122	256	93102	9

Table 8.5: Dimension bounds for PVM

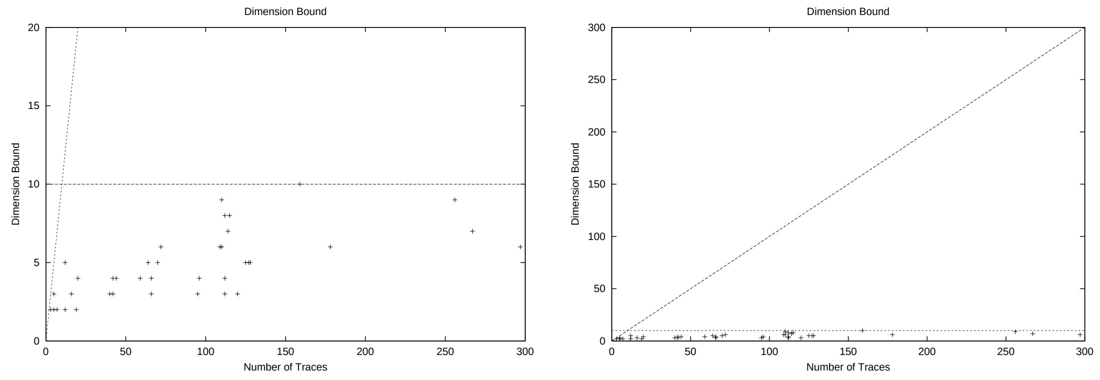


Figure 8.3: Dimension-Bound v. Number of Traces (Note: Scales Differ)

greater than 20 there is a minimum of an order-of-magnitude difference between the dimension and the number of traces. When the number of traces is greater than 100, it is usually a factor of 15 or greater. To help visualize what these results imply, we created a graph, shown in Figure 8.3, which plots dimension-bound as a function of the number of traces. The two graphs shown are the same, though with with differing scales. The horizontal axis is the number of traces, while the vertical axis is the dimension bound. We also plot two additional lines. First we show the "dimension = 10" line, as all results were less than or equal to that value. Second, we show the "dimension = width" line, which illustrates the increase in Fidge/Mattern vector-clock size as the number of traces increases.

In addition to testing with distributed computations, we created a series of broadcast patterns of varying sizes and crown patterns. The results from our program for these patterns was dimension bound N for the crown patterns and 3 for the broadcasts, regardless the number of processes involved in the broadcast. The dimension of all of these cases is optimal.

However, it should by no means be inferred that the algorithm is in general producing optimal results. All we can infer from the current data is that the dimension bounds achieved for a reasonable number and variety of distributed computations are substantially better than the assumed default value of the number of traces involved.

Our experimental results showed little difference in the dimension bound produced when using either the offline or online algorithms [164]. In some instances the bound was higher by one, in others lower by one, in most it was the same.

Then we analysed Life. More precisely, we observed some anomalies in the dimension bounds produced for the execution history of the PVM implementation of Conway's Life (see Section 7.1.2.1 for a description of the PVM implementation of Life). As a result we did a more detailed investigation, determining the dimension bound as a function of the number of iterations in the Life program. We examined executions running on 128 processes, with the number of iterations varying from 9 to 69. Two things became immediately apparent. First, as is shown in Figure 8.4, the dimension bound produced can and does exceed 10 for real programs. While we

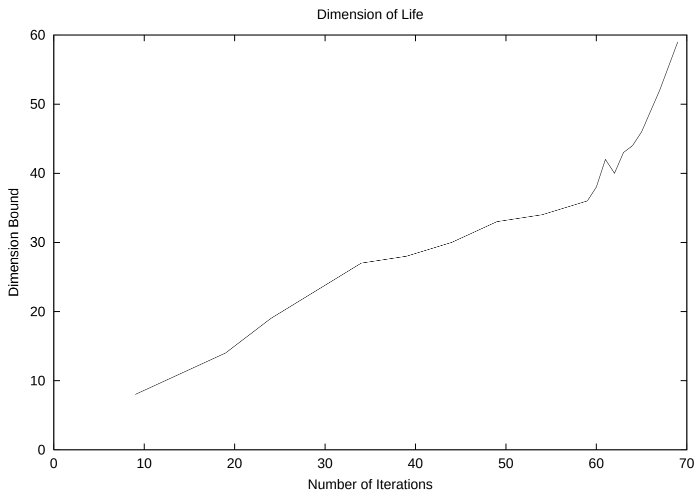


Figure 8.4: The Dimension-Bound of Life

knew this was possible, it was not comforting that this should be the case with such a basic communication paradigm. Second, the online algorithm produced both significantly worse results than the offline algorithm, and for a non-trivial number of cases those results exceeded the number of traces in the computation. Prior to this we did not believe that this was possible. In spite of these issues, we still do not know what the dimension of Life is. Specifically, the algorithm produces a bound, not the actual dimension, as that is NP-hard.

This brings us to the second aspect of the bounding algorithm that we wished to examine: the tightness of the bound. We do not currently have a bound, and had not felt that this was a significant issue, since the results were sufficiently low that it was not relevant to whether or not a dynamic Ore timestamp would be a good idea, if feasible. Based on the results for the Life execution history, however, it is now clear that such bounds would be desirable. In particular, we are not yet convinced that Life does have a high dimension, or that it is inherently impossible to achieve a lower bound should it have a lower dimension. What is clear, however, is that the problem with the algorithm, assuming Life does indeed have a lower dimension than that reported by our algorithm, occurs in the subextension-selection phase of the algorithm, since all other portions of the algorithm are optimal. Dealing with such a problem is hard, since we have no clear basis for deciding which subextension to select for any particular critical-pair insertion.

Further to this problem, it is apparent that we have no clear characterization of low dimension-bound programs. As such, we cannot easily determine when a dimension-bound timestamp would be applicable and when it would not. That said, we still feel that the evidence strongly supports investigating such a timestamp. It was never believed that it would be applicable for all computations, and it never will be applicable for those with high dimension. However, we have demonstrated that a significant number of programs do produce execution histories with a low dimension bound, and for those cases, such a timestamp would be beneficial.

```
1: timestamp($e$) {
2:     $C_e \leftarrow$ criticalPairs($e$);
3:     insert($e$, $C_e$, $R$);
4:     assignVector($e$, $R$);
5: }
```

Figure 8.5: Dynamic-Ore Timestamp Algorithm

8.2 Dynamic-Ore Timestamp Algorithm

In this section we present our dynamic variant of the Ore timestamp. A preliminary version of this work was presented in our framework-algorithm paper [166].

Before we begin, we must deal with a fairly fundamental problem. All of the terminology and theory of partial orders that we have thus far discussed, and that we have found in the literature, pertains to static entities. That is, the partial order exists in entirety. There is no notion of dynamic partial orders. Our solution to this problem was to treat the set of events and their interaction at any given instant as the complete partial order. The arrival of a new event (that is, information pertaining to a previously unknown event) then creates a new partial order. To deal with this dynamic nature, we develop theorems regarding the relationship between these two distinct partial orders. It is at this point that it becomes clear why it has been necessary to make explicit which partial order various relations refer to.

Our high-level algorithm is as shown in Figure 8.5. For each new event, the algorithm determines the critical pairs implied by that event, and uses those as a constraint when adding the new event to the pseudo-realizer. After the event is added to the pseudo-realizer, its position in the various extensions is used to determine its vector timestamp. Note that it is a pseudo-realizer, and not a realizer, because the extensions that form it are not linear. It is, however, sufficient per Theorem 1 (page 11). We will now describe our algorithms for each of these steps.

8.2.1 Incremental Computation of Critical Pairs

The requirements for the incremental computation of critical pairs are as follows. First, we need a time-efficient algorithm. Specifically, we must have an algorithm that is $O(1)$ with respect to the number of events in the computation, or the algorithm could not be considered to be dynamic. Also, we prefer that it is no more than $O(N)$ with respect to the number of traces, as this is the cost of the Fidge/Mattern algorithm. In addition to these time requirements, we also constrain the space consumption per event to be no more than $O(d)$ amortized, where d is the number of extensions in the pseudo-realizer. Without the space saving, the algorithm would be of no value. Finally, the critical pairs must be computed dynamically. Only the first of these requirements is satisfied by the algorithm of Figure 8.1. Further, the theorem that algorithm rests on does not satisfy the dynamic requirement.

To compute the set of critical pairs dynamically, we first recall Theorem 4: (e^{i}, e^{j}) form a critical pair if-and-only-if $e^{i} \in \mathrm{leastConcurrent}_{\mathcal{E}}(e^{j})$ and $e^{j} \in \mathrm{greatestConcurrent}_{\mathcal{E}}(e^{i})$. In this dynamic algorithm, the set of events that are least or greatest concurrent to a given event may change as new events arrive. As a result, we require two additional theorems: one to indicate which critical pairs are no longer valid in the presence of a new event, and one to indicate which new critical pairs are present as a result of the new event.

In order to be able to efficiently determine the change in the critical-pair set, we must put a restriction on the order in which new events are processed. We require that the events be processed in some linear extension of the partial order. The effect of this condition is that we are now dealing with additions to the partial order of new maximal elements, and of no other kind. This enables the development of our required theorems. We start with a lemma that determines the effect of a new maximal event on the least concurrent set of existing events.

Lemma 1 (Consistency of Least Concurrent Sets) *Elements in the least-concurrent set of an event remain in that set under the addition of new maximal elements to the partial order.*

$$e^i, e^j \in \mathcal{E} \Longrightarrow \big(e^i \in \mathrm{leastConcurrent}_{\mathcal{E}}(e^j) \Leftrightarrow e^i \in \mathrm{leastConcurrent}_{\mathcal{E}\cup\{e\}}(e^j)\big)$$

Proof: Suppose $e^i \in \mathrm{leastConcurrent}_{\mathcal{E}}(e^j)$. Then $e^i \parallel_{\mathcal{E}} e^j$ and $\nexists_{e^k}\ e^k \prec_{\mathcal{E}} e^i \ \wedge\ e^k \parallel_{\mathcal{E}} e^j$. Since e is maximal, $e \not\prec_{\mathcal{E}\cup\{e\}} e^j$ and $e \not\prec_{\mathcal{E}\cup\{e\}} e^i$. Therefore $e^i \parallel_{\mathcal{E}\cup\{e\}} e^j$. For the same reason, $\nexists_{e^k} e^k \prec_{\mathcal{E}\cup\{e\}} e^i \ \wedge\ e^k \parallel_{\mathcal{E}\cup\{e\}} e^j$. Therefore $e^i \in \mathrm{leastConcurrent}_{\mathcal{E}\cup\{e\}}(e^j)$. On the other hand, suppose $e^i \in \mathrm{leastConcurrent}_{\mathcal{E}\cup\{e\}}(e^j)$. Then $e^i \parallel_{\mathcal{E}\cup\{e\}} e^j \wedge\ \nexists_{e^k} e^k \prec_{\mathcal{E}\cup\{e\}} e^i \ \wedge\ e^k \parallel_{\mathcal{E}\cup\{e\}} e^j$. Since both e^i and e^j are in $\mathcal{E}$ and e is maximal, $e^i \parallel_{\mathcal{E}} e^j \wedge\ \nexists_{e^k} e^k \prec_{\mathcal{E}} e^i \ \wedge\ e^k \parallel_{\mathcal{E}} e^j$. Therefore $e^i \in \mathrm{leastConcurrent}_{\mathcal{E}}(e^j)$. □

We now state and prove the theorem that determines which critical pairs are no longer valid under the addition to set $\mathcal{E}$ of maximal element e.

Theorem 5 (Remaining Critical Pairs) *The critical pair (e^i, e^j) of the partial order $(\mathcal{E}, \prec_{\mathcal{E}})$ is not in the set of critical pairs of the partial order $(\mathcal{E} \cup \{e\}, \prec_{\mathcal{E}\cup\{e\}})$ if and only if e^j is covered by e and e^i is concurrent with e.*

$$(e^i, e^j) \in \mathrm{CP}_{\mathcal{E}} \Longrightarrow (e^j <: e \ \wedge\ e^i \parallel_{\mathcal{E}\cup\{e\}} e) \Leftrightarrow (e^i, e^j) \notin \mathrm{CP}_{\mathcal{E}\cup\{e\}}$$

Proof: (Necessary:) Suppose $(e^j <: e \ \wedge\ e^i \parallel_{\mathcal{E}\cup\{e\}} e)$. Thus $e^j \notin \mathrm{greatestConcurrent}_{\mathcal{E}\cup\{e\}}(e^i)$. Therefore $(e^i, e^j) \notin \mathrm{CP}_{\mathcal{E}\cup\{e\}}$.
(Sufficient:) Suppose that $(e^i, e^j) \notin \mathrm{CP}_{\mathcal{E}\cup\{e\}}$. Then either $e^i \notin \mathrm{leastConcurrent}_{\mathcal{E}\cup\{e\}}(e^j)$ or $e^j \notin \mathrm{greatestConcurrent}_{\mathcal{E}\cup\{e\}}(e^i)$. Since $(e^i, e^j) \in \mathrm{CP}_{\mathcal{E}}$ then $e^i \in \mathrm{leastConcurrent}_{\mathcal{E}}(e^j)$ and $e^j \in \mathrm{greatestConcurrent}_{\mathcal{E}}(e^i)$. By Lemma 1, $e^i \in \mathrm{leastConcurrent}_{\mathcal{E}\cup\{e\}}(e^j)$. Therefore $e^j \notin \mathrm{greatestConcurrent}_{\mathcal{E}\cup\{e\}}(e^i)$. Therefore $\exists_{e^k} e^k \parallel_{\mathcal{E}\cup\{e\}} e^i \ \wedge\ e^j \prec_{\mathcal{E}\cup\{e\}} e^k$. Since there is no such $e^k \in \mathcal{E}$, the only such e^k is e. Therefore $e \parallel_{\mathcal{E}\cup\{e\}} e^i \ \wedge\ e^j \prec_{\mathcal{E}\cup\{e\}} e$. Now we have to show that $e^j <: e$. Suppose not. Then $\exists_{e^l} e^j \prec_{\mathcal{E}\cup\{e\}} e^l \wedge e^l \prec_{\mathcal{E}\cup\{e\}} e$. Since $e^i \parallel_{\mathcal{E}\cup\{e\}} e$ and $e^i \parallel_{\mathcal{E}\cup\{e\}} e^j$, then $e^i \parallel_{\mathcal{E}\cup\{e\}} e^l$. Also $e^l \in \mathcal{E}$. Therefore $\exists_{e^l} e^l \parallel_{\mathcal{E}} e^i \wedge e^j \prec e^l$. Therefore $e^j \notin greatestConcurrent_{\mathcal{E}}(e^i)$ which is a contradiction. □

Since e is maximal $e \not\prec_{\mathcal{E}\cup\{e\}} e^i$. Therefore the concurrency of e^i with e can be verified or refuted by merely checking $e^i \prec_{\mathcal{E}\cup\{e\}} e$.

Our second theorem defines the entire set of additional critical pairs that may result from the addition of e.

Theorem 6 (Additional Critical Pairs) *The only new critical pairs formed by the presence of a new event e are those of the form (e^i, e) or (e, e^j) that satisfy Theorem 4.*

$$(e^i, e^j) \in \mathrm{CP}_{\mathcal{E}\cup\{e\}} - \mathrm{CP}_{\mathcal{E}} \iff \begin{array}{l}(e^i = e \vee e^j = e) \wedge e^i \in \mathrm{leastConcurrent}_{\mathcal{E}\cup\{e\}}(e^j) \wedge \\ \qquad e^j \in \mathrm{greatestConcurrent}_{\mathcal{E}\cup\{e\}}(e^i)\end{array}$$

Proof:
($\Longleftarrow$): $(e^i = e \vee e^j = e) \wedge e^i \in \mathrm{leastConcurrent}_{\mathcal{E}\cup\{e\}}(e^j) \wedge e^j \in \mathrm{greatestConcurrent}_{\mathcal{E}\cup\{e\}}(e^i)$. Then $(e^i, e^j) \in \mathrm{CP}_{\mathcal{E}\cup\{e\}}$. Since, $e \notin \mathcal{E}$ by definition and $(e^i = e \vee e^j = e)$, then $(e^i, e^j) \notin \mathrm{CP}_{\mathcal{E}}$. Therefore $(e^i, e^j) \in \mathrm{CP}_{\mathcal{E}\cup\{e\}} - \mathrm{CP}_{\mathcal{E}}$.
($\Longrightarrow$): $(e^i, e^j) \in \mathrm{CP}_{\mathcal{E}\cup\{e\}} - \mathrm{CP}_{\mathcal{E}}$. Therefore $(e^i, e^j) \in \mathrm{CP}_{\mathcal{E}\cup\{e\}}$ and $(e^i, e^j) \notin \mathrm{CP}_{\mathcal{E}}$. This then implies that $e^i \in \mathrm{leastConcurrent}_{\mathcal{E}\cup\{e\}}(e^j)$ and $e^j \in \mathrm{greatestConcurrent}_{\mathcal{E}\cup\{e\}}(e^i)$ and $e^i \notin \mathrm{leastConcurrent}_{\mathcal{E}}(e^j) \vee e^j \notin \mathrm{greatestConcurrent}_{\mathcal{E}}(e^i)$. We now have to show that this implies that $(e^i = e \vee e^j = e)$. Assume $e^i \neq e \wedge e^j \neq e$. Therefore, by Lemma 1, $e^i \in \mathrm{leastConcurrent}_{\mathcal{E}}(e^j)$. Thus $e^j \notin \mathrm{greatestConcurrent}_{\mathcal{E}}(e^i)$. This then implies that $\exists_{e^k}\ e^k \parallel_{\mathcal{E}} e^i \wedge e^j \prec_{\mathcal{E}} e^k$. Since $e^i, e^j \in \mathcal{E}$ and e is maximal $e^k \parallel_{\mathcal{E}\cup\{e\}} e^i \wedge e^j \prec_{\mathcal{E}\cup\{e\}} e^k$. This contradicts $e^j \in \mathrm{greatestConcurrent}_{\mathcal{E}\cup\{e\}}(e^i)$. $\Box$

The only new critical pairs are those that are formed with the event e as one of the elements of the pair. Furthermore, the combination of these two theorems clearly indicates that if (e^i, e^j) is a critical pair but ceases to be after the addition of e, then (e^i, e) will be a critical pair. We can also observe that in such a case any extension of the partial order that reverses (e^i, e) will also reverse (e^i, e^j) since $e^j \prec_{\mathcal{E}\cup\{e\}} e$. We therefore do not need to concern ourselves with the computation of which critical pairs cease to be critical pairs on the addition of a new event. Rather, we solely calculate the new pairs implied by the new event, and use those pairs, in conjunction with the partial-order relationships, as our constraints on inserting the event into the pseudo-realizer.

Using these theorems, Corollary 1, and the optimization for computing the greatest-concurrent set for maximal events described in Section 6.2.2, we developed a complete incremental critical-pair algorithm that satisfies our requirements. It is shown in Figure 8.6.

The algorithm must perform four things for each new event arrival. First, it calculates the Fidge/Mattern timestamp for the event (line 3). The Fidge/Mattern timestamp is necessary to determine various precedence relationships between the new event and existing events. Given that it is needed, it is also needed to compute further Fidge/Mattern timestamps.

Second, the algorithm removes Fidge/Mattern timestamps for any events for which the new event is the last covering event. Although we require Fidge/Mattern timestamps, we must remove them at the first opportunity to satisfy our size-bound requirement. Per Equations 6.3 and 6.4,

```
 1: criticalPairs($e$) {
 2:    $C_e \leftarrow 0$;
 3:    $\mathcal{FM}(e) \leftarrow$ fidgemattern($e$);
 4:    $\forall_{e^i}\ (e^i <: e)$ {
 5:       if ($e$ is last covering event of $e^i$)
 6:          delete $\mathcal{FM}(e^i)$;
 7:    }
 8:    $\forall_{e^i}\ (e^i \in \mathcal{FM}(e))$ {
 9:       if $(\exists_{e^j}\ e^i <: e^j\ \wedge\ \phi(e^i) = \phi(e^j))$
10:          if $(\forall_{e^k}\ e^k\ <: e^j\ \wedge\ \phi(e^k) \neq \phi(e^j)\ \wedge\ \mathrm{EventPosition}(e^k) \leq \mathcal{FM}(e)[\phi(e^k)])$
11:             $C_e \leftarrow C_e \cup (e^j, e);$
12:    }
13:    $\forall_{e^i}(\nexists_{e^j}\ e^i <: e^j)$
14:       if $(\forall_{e^k}\ e^k <: e\ \wedge\ \mathrm{EventPosition}(e^k) \leq \mathcal{FM}(e^i)[\phi(e^k)])$
15:          $C_e \leftarrow C_e \cup (e, e^i);$
16:    return($C_e$);
17: }
```

Figure 8.6: Dynamic Critical-Pair Computation

the value of this timestamp for an event e depends on the timestamps of the events that e covers. Therefore, we must maintain a copy of the Fidge/Mattern timestamp for any event for which we have not seen all the events that cover it. After that time we may delete it. This is the function of lines 4 to 7.

Under our general restriction of processing events in some linearization of the partial order, the number of events that are not completely covered at any given time can be arbitrarily large. While it is not possible to further restrict this linear ordering to guarantee a bounded number of events that are not completely covered, in practice a (small) bound does exist. There are three reasons for this. In the case of systems with only synchronous communication, there can never be more than N events that have not been covered. For systems with asynchronous communication, the number of events not yet covered is bounded by the resources of the distributed system. Since these resources are finite, the number of such events is bounded. Finally, in practice we do not observe systems that have significant numbers of uncovered events outstanding. This is probably because the correctness of such systems would be subject to system resources, which is not a robust basis for guaranteeing system behaviour.

Third, we must consider all events that are least-concurrent to the new event (lines 8 to 10). The algorithm uses a small variant on the least-concurrent method defined in Section 6.2.2. First, it does not need to check if the potentially least-concurrent event is a successor to e, since e is maximal. Second, it cannot use a simple precedence test, as there is no guarantee that the potentially least-concurrent event still has a Fidge/Mattern timestamp. We could alter the algorithm to ensure that such events did still have a Fidge/Mattern timestamp. We would do so by keeping track of the intersection of the predecessor sets of new events, and only removing the Fidge/Mattern timestamps of events covered by the slice corresponding to that intersection cut. Such an approach should only cost $O(N)$, and would still reasonably bound the number of Fidge/Mattern timestamps that were kept. However, there is a better way. The Fidge/Mattern timestamp of event e corresponds to the predecessor set of that event. To determine precedence with respect to e, it is then sufficient to determine membership in that set. This can be accomplished merely by looking at the EventID, and comparing with the relevant entry of e's Fidge/Mattern timestamp. This is the function of line 10. Since e is maximal, it must be greatest-concurrent to anything that is least-concurrent to it, and thus the critical pairs of the form (e^l, e) are identified.

Fourth, we must consider all events that are greatest-concurrent to the new event (lines 13 and 14). If the new event is least-concurrent to such events, they too would form a tentative critical pair. Per the optimization of the greatest-concurrent method (Section 6.2.2), since e is maximal, the greatest-concurrent set of e is precisely the set of maximal events, excluding e (line 13). Note that the maximal events are not simply the maximum events in each trace. For all such maximal events, the algorithm determines if there is an event causally prior to e that is also concurrent. The algorithm applies the same precedence-testing technique as was used for the least-concurrent precedence test. It can do so because all maximal events still have Fidge/Mattern timestamps, as they will be needed to compute successor Fidge/Mattern timestamps. If there is no causally prior concurrent event, then we have identified a critical pair of the form (e, e^g) (line 15).

For the single-partner case, this algorithm requires $O(N)$ time. The reason is as follows. The Fidge/Mattern computation (line 3) requires $O(N)$ steps. The first loop (lines 4–7) will have at most two iterations, since an event can only cover two other events in the single-partner case. The second loop (lines 8–12) will require N-iterations. However, each statement within it is $O(1)$. Specifically, the **if** statements of lines 9 and 10 will each examine at most two events, due to the single-partner restriction. The final loop (lines 13–15) will also require N-iterations, but again the statements within it are $O(1)$.

When multiple partners are permitted line 10 becomes $O(N)$ if e^j is a multi-receive event, turning lines 8 to 11 into $O(N^2)$ cost. Likewise, if e is a multi-receive, the line 14 becomes $O(N)$, turning lines 13 to 15 into $O(N^2)$. If multi-receive is permitted, without multicast, then the cost would be amortized $O(N)$, since although a multi-receive would be $O(N^2)$, there would have to be $O(N)$ corresponding unicasts. Since a multicast event only covers one event, and that in the same trace, they do not change the cost from the single-partner case. However, if both multicast and multi-receive were permitted, then the cost could be $O(N^2)$ amortized. This would amount to most or all communication events being multicast or multi-receive, which only occurs in highly fault-tolerant systems.

Synchronous events occurring in $O(N)$ traces would have an effect similar to multi-receive events in the presence of multicast. They would cover $O(N)$ events, and thus increase the cost to $O(N^2)$ for the particular event. Their amortized cost cannot be reduced to $O(N)$, since they also act in a like manner to multicast events, being covered by $O(N)$ events. While we do not have evidence to support the assertion that $O(N)$-home synchronous events would be rare (since no tool supports more than single partners), it is our belief that they would not represent a problem in practice.

The space consumption of the output of the algorithm is $O(N)$ since that is the number of critical pairs that will be produced. This number is supported empirically. However, there is no need to store the critical pairs after the event has been inserted into the pseudo-realizer. The only space that must be maintained between critical-pair calculations is that required to store the raw events and the Fidge/Mattern timestamps for events that are not yet fully covered. Thus, the amortized space-consumption cost is $O(1)$ per event.

8.2.2 Building Pseudo-Realizers

The requirements for the incremental building of a pseudo-realizer are as follows. First, it must reverse the critical pairs discovered for the new event. This is the sense in which it is a realizer for the partial order. It need not be a true realizer, however, in that we do not require the extensions we form to be linear. Second, it must be possible to determine efficiently the precedence relation between any two events stored in the pseudo-realizer. That is, we must be able to build vectors using the extensions. The Ore timestamp requires linear extensions to build vectors. We have discovered, and will describe in Section 8.2.3, that this requirement can be relaxed somewhat. Third, the insertion of an event into the pseudo-realizer must have an amortized cost of $O(1)$ with respect to the number of events, and we prefer that it be no more than $O(N)$ with respect to the number of processes. The first of these constraints is necessary for it to be considered a dynamic algorithm; the second requires that we are no less efficient in building the timestamps than is the Fidge/Mattern algorithm. The fourth requirement is that the space consumption per event must be no more than $O(d)$. Reduced space consumption is a primary benefit that we are seeking. Finally, d must be reasonably small, or at least a reasonably close approximation to the dimension of the partial order of computation. This final requirement is based on the fact that building the minimum number of extensions that reverses the critical pairs of a partial order is NP-hard for dimension greater than two [177]. Any dynamic algorithm we develop will be non-optimal. However, it must be sufficiently good in practice to produce pseudo-realizers with only a small number of extensions where the dimension is low. There would be no value in an algorithm that produced a number of extensions similar to the number of processes in the computation.

Given these requirements, we have defined a high-level event-insertion algorithm, shown in Figure 8.7. In brief, we must place the new event in every extension such that the set of extensions reverses all critical pairs associated with that event (recall Theorem 1, page 11). Note that while line 2 should not be construed as implying any particular placement order, or even an ordered placement approach at all, the `place` function of line 3 is required to remove any critical pairs reversed by that placement from the set of critical pairs, and return the resulting set for use in the remaining `place` operations. In this way it is small matter to determine if all critical pairs have been reversed (line 4), and it also means that later placements are not constrained by requirements that have already been satisfied. If all critical pairs have not been reversed in the existing extensions, we must create new extensions that will allow us to reverse the critical pairs

```
1: insert($e$, $C_e$, $R$) {
2:    $\forall_{l \in R}$
3:       $C_e \leftarrow$ place($e$, $l$, $C_e$);
4:    while ($C_e \neq \emptyset$) {
5:       $l \leftarrow$ createExtension();
6:       $R \leftarrow R \cup l$;
7:       $C_e \leftarrow$ place($e$, $l$, $C_e$);
8:    }
9: }
```

Figure 8.7: Event-Insertion Algorithm

that have not yet been reversed. This is the function of lines 4 to 8. Any reasonable placement algorithm should require only one new extension, and then only because the existing extension *cannot* reverse some critical pair, not because the choice of placement means that they *do not* reverse it.

The cost of this algorithm is $O(|R|\,k\,|C|)$, where k is the number of events a given event can cover. The $|R|$ factor comes from the number of iterations of the loop in line 2. The $k\,|C|$ factor is the cost of event placement in line 3, and will be justified in the following section. New-extension creation (lines 4–8) is expected to be extremely infrequent, and thus not a significant cost (if it is not, then this would not be a good timestamp for reasons other than the cost of pseudo-realizer creation).

This high-level algorithm identifies the requirement for two sub-algorithms, for event placement and new-extension creation, respectively. The algorithms used for the dimension-bound analysis work do not adequately address either of these problems for several reasons. First, they processed critical pairs, rather than events. Second, they store Fidge/Mattern timestamps with every event for later use in the algorithm, which violates the fourth requirement. Third, they built subextensions, rather than full extensions, which are insufficient for a pseudo-realizer.

8.2.2.1 Event Placement

We address the problem of event placement by first dividing the critical pairs into those of the form (e^l, e) and those of the form (e, e^g). The reversal of those in the latter category is trivial to satisfy since e is maximal. We can simply place e at the end of some extension. Note that this neither violates the partial order nor alters existing critical-pair reversals. In this regard, it is not strictly necessary to calculate critical pairs of this form. On the other hand, if we wish to minimally commit our extensions (that is, we do not wish to establish orderings that are implied neither by the partial order nor by the requirement to reverse critical pairs), then we may choose to be more selective.

However, there is an alternate solution that allows us to neither over-commit our extensions nor require us to compute the (e, e^g) critical pairs. Since placing the event at the end of some extension will satisfy the constraints imposed by any (e, e^g) critical pair, we maintain a virtual extension that simply records the order in which events are processed. That is, we maintain an index (referred to as the virtual-extension index), initially 0, that is incremented for each event stored in the partial order. The value of the index at the time of storage is used as the first element of our dynamic-Ore vector timestamp. This virtual extension thus costs constant space, as opposed to the actual extensions which will require space proportional to the number of events. However, it does require one additional integer storage per event for the timestamp.

Critical pairs of the form (e^l, e) are more difficult to reverse. If a new extension is required, the reversal of these pairs is what will force it, as they require that e be placed earlier in some extension than e^l. We can easily determine if an extension will allow the reversal of an (e^l, e) critical pair by determining whether the events that e covers occur after e^l in the extension. These events and e^l will have been processed, and thus we have vector timestamps for them. As such we can check the relevant timestamp entry. We can thus place a range on the location of e in the extension: after the events it covers and before the e^l events with which it forms (e^l, e) critical pairs. We have not yet investigated this notion of giving a range to a new event, rather than a specific location, though we have examined the idea of using arbitrary DAGs rather than (near-)linear extensions. We describe the results of that below.

A heuristic that satisfies the above requirement is to place e at the earliest location possible in each extension. This location is immediately after the last event in the extension that e covers. This is computable in k steps per extension, where k is the number of events that e covers. We then assign an index to the event, to be described in Section 8.2.3. We can then check, in one step per critical pair, whether or not the critical pair has been reversed. If the index assigned to the event is less than that of the e^l element of the critical pair, then the (e^l, e) critical pair has been reversed. Any critical pair that has been reversed is removed from the set of critical pairs. Any critical pair that has not been reversed, cannot be reversed in this extension, since we have placed event e at its first possible location that still satisfies the partial-order constraints.

The resulting extension is linear which may cause an over-commitment problem. However, we can think of no way that avoids this problem that is computationally reasonable. Consider the alternative of using an arbitrary DAG to represent an extension. If we built such extensions, how would we assign indices? This problem is solvable. Any valid linearization of the DAG order would suffice. Indeed, a complete linearization is not necessary. What is required is that any event that precedes another event in the DAG have a lower index. Suppose we had an extension containing events e^i, e^j and e^k, such that e^i preceded e^j, while both were concurrent to e^k in the extension. If e^k is assigned the same index as e^i, then it will have a lower index than e^j. How will we determine that $e^k \parallel_{\mathcal{E}} e^j$? The answer is, we do not have to. More precisely, no single extension is required to fully determine the partial-order relationships. Indeed, no single extension can, since their very purpose is to reverse critical pairs. The fact that $e^k \parallel_{\mathcal{E}} e^j$ means that there will be critical pairs that force an ordering between these two events in two different extensions. In those extensions, the indices would yield the required ordering to apply the Ore precedence test.

Now consider event insertion into the DAG. We will assume that the nodes of our extensions will contain pointers to successor events, and an integer index for the event. We refer to any such node as the location node of the event within the DAG. Rather than store a vector timestamp in the event structure, we will assume that we store a sequence of pointers to the event's location node within each DAG of the pseudo-realizer. To insert a new event, we would seek the events it covers and add pointers from those events' location nodes within the DAG to the new event.

Now the problem arises. We wish to reverse critical pairs in the extension. For this we must efficiently determine if the extension will accept the reversal of a critical pair. Recall that all such pairs will be of the form (e^l, e), where e is the new event being inserted into the extension. The extension can only reverse this critical pair if e^l does not precede in the DAG the events that e covers. Recursion has occurred. Determining precedence in DAGs is precisely the problem we are trying to solve with our timestamp. It may be the case that the properties of this DAG extension are simpler than those of the original partial order, and such a precedence determination is feasible. However, we do not have any method for this.

We have explored this concept of DAG extensions so as to demonstrate precisely where the problem lies in their usage. If this problem can be surmounted, then they can be used, and would probably be more effective than linear extensions. In the meantime, we propose using the heuristic with linear extensions, as described above.

8.2.2.2 Extension Creation

The main requirement of good extension creation is, as with event placement, to avoid over-commitment. If we were building a realizer rather than a pseudo-realizer, we would have to make any new extension linear. This would force premature commitment to a specific ordering that might well be poor. Prior experience shows this to be the case [163]. However, at the point where we need to create a new extension, precedence can already be determined between all prior events without this extension. In other words, the only precedence-testing ability that this new extension will give us is with the current and subsequent events. We can take advantage of this as follows. Rather than force an ordering on existing events, for each event we keep track of how many extensions were in use at the time it was inserted into the pseudo-realizer. This can be achieved in $O(1)$ space-consumption cost over all events because of the virtual extension. Since the first element of every event's timestamp will record the virtual-extension index at the time of storage, we merely need to record the value of the index at the time a new extension is created. We can then easily determine the number of extensions at the time an event was timestamped by the value of its first vector element.

This allows us to test for precedence using only a subset of the extensions. This in turn means we merely have to order existing events with respect to the new event, and subsequent new events, that required the new extension. Since the new event is maximal, and we still have its Fidge/Mattern timestamp at this point, we can readily determine which events precede the new event and which are concurrent. We then do the following. All events that are concurrent with the event being inserted are assigned the same index as the event being inserted. This must be done explicitly, by iterating forward on each trace from the greatest-predecessor events to the maximum event currently recorded for that trace. The upper bound on the number of events that must be given this additional timestamp vector entry is n. However, this implies that n events have been inserted, which means the amortized cost is $O(1)$ per event. All events causally prior to the event being inserted are simply given a 0 index value for this extension. This may be done implicitly by simply not placing the events in the extension.

How then do we deal with critical-pair reversal in such an extension, given the problem described in the previous section? The problem, as described in that section, is to determine if e^l does not precede in the DAG the events that e covers, where the critical pair is (e^l, e). The problem occurs if e^l or one of the events that e covers are in that collection of events that were stored prior to this extension's creation. Our solution is as follows. Per the above reasoning, all events prior to the event that caused the extension's creation, can be placed in arbitrary locations in the extension with respect to one another. This includes violating the partial order relationship. We therefore relocate e^l and any events that e covers as required to accept the critical pair. In relocating an event in the extension, we assign a new index to that event. Thereafter, it would have neither the 0 index nor the same index as the event that caused the extension's creation. We thus determine relocatability by whether or not an event has such an index. The relocation of an event is then in accord with the previous rules of placement, insofar as that is possible. Thus, we create a new extension that satisfies the precedence requirements with minimal over-commitment.

One final comment must be made about this creation mechanism. Given that it requires an alteration to existing timestamped events, it can be argued that it is not, therefore, dynamic. However, it should be noted that this augmentation does not invalidate prior timestamp relationships. The additions are for the use of the current and future events. There is nothing in this timestamp algorithm that precludes its use in an online-monitoring situation.

8.2.3 Vector Assignment

The vector-assignment problem is how to assign a value to an event to indicate its position within the extension. In the case of the Ore timestamp an integer was used. This is possible because the realizer is computed in entirety before positional integers are assigned. In our case, we must assign values as new events arrive, and those new events are not, in general, located at the end of the extension. We cannot simply adjust the position information for all events after the new event in the extension. An unbounded number of events would need to have their relevant vector-entry updated. Rather than do this, we first solve the problem in theory. Instead of using an integer to represent a position, we use a real number. Since the real numbers are infinitely divisible, we can always assign a new real number to a new event to correctly identify its place within an extension. Since real numbers do not exist in computers, we emulate their effect by leaving substantial gaps between the position integers. Whenever we can no longer subdivide the integer space to insert new events, we perform a vector update on all events in the extension after the new event. This is no more expensive than if we did it with each new event, and is amortized over a large number of events.

Given the need to rapidly locate events within an extension, and the occasional vector-update sweep, rather than store the indices in the EventRef::Data object, we store them in the extension, and have the EventRef::Data object timestamp vector elements point to the event's location within the various extensions. If an EventRef object is exported to another process, the current state of its vector indices can be extracted and exported in lieu of the pointers. Such a remote EventRef object would require a back-reference to the data structure in the case that a comparison operation required a more up-to-date copy of the timestamp.

8.2.4 Precedence Tests

Finally, we can specify the precedence test. This is essentially the Ore test, though only up to the greater number of extensions that existed at the time the two events were timestamped. Thus, the code is as follows.

```
bool operator<(EventRef& e1, EventRef& e2) {
   int range = max(e1.extensionCount(),e2.extensionCount());
   int index = 0;
   while((e1.timestamp(index) < e2.timestamp(index)) &&
         (++index < range));
   return (index == range);
}
```

The extensionCount() method returns the number of extensions at the time the event was timestamped. The timestamp() method returns the element of the timestamp at the given index. For a partial order with a pseudo-realizer containing d extensions, the cost of the test is $O(d)$. While this is superficially worse than the Fidge/Mattern test, as we have seen in Chapter 7, the effective cost of that test, due to the need for caching, is $O(N)$. This does, however, presume that this the size of d is sufficiently small that the entire structure can fit in main memory.

Note that this is a somewhat distributed precedence test, depending on how the vector indices are maintained. Specifically, while no other events need to be examined, the extensionCount() method would need to consult the partial-order data structure to determine the number of extensions present at the time the event was stored. If this proved to be a problem, that value can be stored directly in the EventRef::Data object itself, albeit at the cost of some space. Presuming the number of extensions to be small, it should easily fit within a byte. Likewise, as noted in the previous section, the vector indices can be copied for use in a remote process, but some precedence tests would require a centralizing update of the timestamp.

Second, since this timestamp is not trace-based, we no longer have a greatest-predecessor method. That method was provided by the Fidge/Mattern timestamp. However, we can adapt the method used to compute the least successor slice. That technique performed a search for the least successor on the trace over the range of the greatest predecessor up to the maximum event currently stored on the trace. To compute the greatest predecessor, we now have to search over the entire range of the trace. The execution time-bound for this is the same as that for computing the least successor, though the actual number of events that would have to be examined in practice would likely rise. The remaining precedence-related event sets can be computed per the description in Section 6.2.

8.3 Analysis

In this chapter we have presented a novel algorithm for dynamic, centralized dimension-bounded timestamps. We have completely solved the problem of incremental computation of critical pairs. We have presented one solution to the problem of building pseudo-realizers and it has, thus far, produced promising results. We have provided a solution to the problem of assigning vectors to events, though it centralizes the precedence test. We have presented the necessary precedence-test algorithms.

This approach is significant since, insofar as we are able to build efficient pseudo-realizers, we can create significantly smaller vector timestamps with little cost increment in the precedence test. The ability to build efficient pseudo-realizers is a function of the algorithm used and the distributed-computation event interactions.

However, there are a significant number of potential problems with this timestamp. First, it is unclear if efficient pseudo-realizers can be created and, even if they can, it is possible that elementary communication paradigms such as the repeated neighbour interactions of Life really do have a large dimension. In such a case, this approach would be inapplicable for this important class of applications. A clear characterization of the class of programs for which such a timestamp is applicable is required.

Second, and in some respects a more significant problem, there does not appear to be any good caching strategy for these timestamps. The algorithm we have defined in this chapter relies on the computation of Fidge/Mattern timestamps. It is thus not possible to directly adapt it for caching purposes. While it is possible to compute the Fidge/Mattern timestamp for any event given this timestamp (per the algorithm for greatest predecessors identified in the previous section), the cost of this operation is $O(N \log(n/N))$, which is less attractive than cached Fidge/Mattern timestamps. Thus the choice of this timestamp relies on the capability of fitting the timestamp structure into available memory. In this sense, they may be less space-consumptive, but it can also be argued that they are not scalable.

Third, it is not clear what the practical space-consumption requirements are of this algorithm. Clearly the extensions require $O(d)$ space per event, where the Fidge/Mattern timestamp required $O(N)$ space per event. However, the Fidge/Mattern timestamp is precisely N integers. In this algorithm, there are d pointers per event into the extension. Each location-node in an extension in turn requires one integer for the index and one pointer to indicate its successor. It may be that this can be improved upon [120]. However, as defined, this clearly requires about $3d$ in space per event. If we were able to solve the problem of creating extensions using DAGs, the space consumption is likely to increase, since the location nodes would then have multiple successors. Given the overall complexity of this approach, the value of d would have to be significantly smaller than N to make it attractive.

For these reasons, and one other, we have not yet implemented this algorithm, and thus cannot report experimental results other than the dimension-bounds already cited. The one other reason is that shortly after defining this algorithm, we developed the dynamic-cluster timestamp approach, and observed immediately that it appears to be a more promising approach.

9 DYNAMIC CLUSTER-TIMESTAMPS

We now turn to our alternate solution, the dynamic cluster-timestamp. This approach is based on the observation that many, possibly most, parallel and distributed computations employ a high degree of communication locality. That is, most traces do not communicate with many other traces. More precisely, there are no communication events directly connecting most traces with most other traces. Note that this does not imply that there is not a transitive connection. Indeed, our experience with Fidge/Mattern timestamps suggests that transitivity of communication rapidly connects all or most traces.

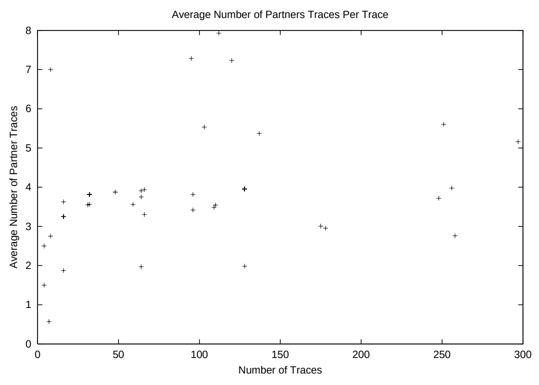
A standard example of this phenomenon is the Single-Program-Multiple-Data (SPMD) programming paradigm for parallel applications [30, 74]. In that technique there is typically a master process that will communicate with all other processes, while the remaining processes will only communicate with those which have required data, usually neighbouring processes in some N-dimensional grid. Thus, in the PVM implementation of Life, all but one process communicated solely with three other processes: left and right neighbours, and the master process.

It should not, however, be thought that this communication locality is limited to such master/slave parallel-programming approaches. We examined the communication patterns for all of the computations that we studied for our dimension-bound programs (see Section 8.1.3) to determine the number of partner traces each trace has. This was averaged for each computation, and a scatter plot of the results is presented in Figure 9.1(a). We then normalized this, by dividing by the number of traces in the computation, to produce the Figure 9.1(b). As can be seen in these figures, the average number of traces each trace communicates with is typically in the 3–4 range, and rarely more than six, regardless the number of traces. This in turn means that the average fraction of traces a given trace communicates with drops as the number of traces rises.

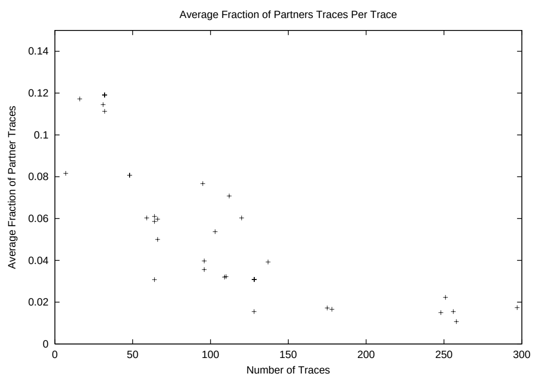
Our approach in this solution is then to attempt to exploit this locality. Our algorithms for this timestamp start with the Summers cluster-timestamp and make a number of very significant changes, enumerated as follows.

1. Encoding is changed to halve the space-consumption.
2. Timestamp-creation is dynamic.
3. Timestamps are hierarchical.
4. A complete precedence test is added.
5. It is independent of Fidge/Mattern timestamps.
6. Clusters are system-selected, not user-selected.
7. A caching strategy is developed.

We divide our algorithm description into three, roughly orthogonal, components. First, we develop the dynamic algorithm. Within this, we describe the precedence test, the mechanism for



(a) Average Number of Partner Traces Per Trace



(b) Average Fraction of Partners Traces Per Trace

Figure 9.1: Average Traffic Patterns

making the algorithm fully hierarchical, and the removal of the Fidge/Mattern computation. The algorithm that results, while being a cluster-timestamp algorithm, is essentially independent of the cluster strategy that is adopted. In Section 9.2 we create two cluster strategies, one static and one dynamic, and demonstrate their requisite cluster algorithms. Third, we needed a caching strategy to allow the integration of our algorithm into a scalable distributed-system observation tool such as POET. This is provided in Section 9.3. We have evaluated our algorithm both analytically and experimentally. We provide theoretical analysis with the algorithm description, and experimental results in Section 9.4. Preliminary versions of this work have been presented that first developed the hierarchical, dynamic algorithm [168] and then the self-organizing clusters [169]. This chapter is a substantial reworking of those papers, adding in the caching strategy which was not previously presented.

9.1 TIMESTAMP ALGORITHM

We describe our algorithm in three steps. We start with a simple dynamic variant of the Summers algorithm and provide an O(cluster size) precedence-test method that works for all events in the partial order. We then extend this to be an arbitrary-depth hierarchical cluster algorithm. This can provide additional aggregate vector-size reduction by allowing a tradeoff between the cluster size and the cluster-receive timestamp size. Finally, we remove the Fidge/Mattern timestamp computation.

9.1.1 TWO-LEVEL ALGORITHM

Our two-level variant on the Summers algorithm makes six changes to that method. First, we require all traces to be members of exactly one cluster. The methods by which this might be achieved are almost orthogonal to the timestamp algorithm, and will be discussed in Section 9.2.

The only point at which the clustering strategy affects the algorithm is in whether or not the strategy is static (that is, pre-determined). We will identify the relevant portions of the algorithm that are required only for dynamic clustering.

Second, we interleave the Fidge/Mattern-timestamp creation with the cluster-timestamp creation. This is based on the recognition that only a bounded number of Fidge/Mattern timestamps need to be maintained at any given time [165]. This interleaving is what enables us to make the algorithm dynamic.

Third, we use an alternative encoding to the Summers technique that can reduce the timestamp size by half. Specifically, the second-half of the timestamp can be replaced by a reference to the greatest cluster-receive that precedes (reflexively) the given event. This is based on the fact that a cluster-receive event e^i can only causally affect another event e^j within the cluster if it is causally prior to that event e^j. This causal precedence is encoded in the first half of the timestamp. Having identified the causal predecessors within the cluster, the cluster-receive predecessors would then be the set of greatest cluster-receives in these respective traces.

Fourth, we incorporate synchronous events into the algorithm. Summers' algorithm was created in terms of cluster-receive events, which were defined as receive events whose corresponding transmit partner is not on a trace within the cluster. We extend, and formalize, this as follows.

Definition 26 (Cluster-Receive) *An event e is a cluster-receive if-and-only-if it is a receive event with a partner event on a trace in a different cluster or a synchronous event whose home traces occur in different clusters.*

Note that this definition is not in terms of coverage. The reason is that coverage is insufficient. It is enough for receive events, as such events cover their transmit partners. However, a synchronous event may occur on traces that are in different clusters, but not cover events that are on traces in different clusters. The problem occurs when the first event in the trace is a synchronous event. In such a case, the event covers nothing.

Fifth, cluster-receive events simply retain their Fidge/Mattern timestamp. In the Summers algorithm cluster-receive events are assigned cluster-timestamps, but their Fidge/Mattern timestamps have to remain recorded. We remove this redundancy.

Finally, we provide a complete precedence-test algorithm that does not require the recreation of the Fidge/Mattern timestamp. It is constant-time for all events within a cluster and O(cluster size) when comparing events in different clusters.

The code for two-level cluster-timestamp creation in the single-partner environment is shown in Figure 9.2. Before describing the algorithm, we will first discuss the additional methods and data necessary to support this code. There are three aspects to this: augmentation of the existing classes, the Cluster class, and the GCR array.

The TraceID class is augmented with the cluster() method, which takes one argument, the PartialOrder object to which the trace belongs, and returns a reference to the Cluster object to which the trace invoking the method belongs. Note that the argument is required as the alternative would require every TraceID object to maintain this information. While this could be encoded reasonably efficiently if the number of partial orders in any given program was small (a not unreasonable assumption), it would result in redundant information in every EventRef::Data object.

```
void timestamp(EventRef& e) {
  e._event->FM = fidgemattern(e);
  if ((e.receive() || e.synchronous()) &&
      (!e.cluster().contains(e.partner().traceID())) &&
      !e.cluster().mergeable(e)) {
    e._event->CT.CT = e._event->FM;
    GCR[e.traceID()] = e.eventPosition();
  } else {
    if ((e.receive() || e.synchronous()) &&
        (!e.cluster().contains(e.partner().traceID())))
      e.cluster().merge(e);
    e._event->CT.CT = e.cluster().project(e._event->FM);
  }
  e._event->CT.CR = GCR[e.traceID()];
}
```

Figure 9.2: Two-Level Dynamic Cluster-Timestamp Creation

Specifically, many will have two or more TraceID objects as member data. In the single-partner case, every EventRef::Data object will have exactly two TraceID member objects.

Since the TraceID is accessible in both the EventID and EventRef classes, the cluster() method is provided directly in those classes. The only difference is that in the EventRef class the PartialOrder object does not need to be passed as an argument. The reason is that EventRef objects know the block of which they are a part (see Section 5.2), and those blocks will maintain the knowledge of the PartialOrder to which they in turn belong. Thus, although an event belongs to a trace, which in turn belongs to a cluster, we will tend to abuse notation and refer to that event as belonging to the cluster.

In addition to the cluster() method, the EventRef::Data class has the following new data members.

```
class EventRef {
...
  class Data {
  ...
  EventPosition* FM;    // Fidge/Mattern timestamp
  class CT {
    EventPosition  CR;  // Cluster Receive
    EventPosition* CT;  // Cluster timestamp
  } CT;
};
```

The FM EventPosition array holds the Fidge/Mattern timestamp until such time as it is disposed of. Since there should be a limited number of such timestamps stored at any given point (*per* the discussion in Section 8.2.1), this could be reworked so that this pointer was not required as member data, but was stored in some separate structure. We show it this way for convenience of explanation. The cluster timestamp is composed of two parts. The EventPosition array CT is of length equal to the cluster size and identifies the greatest predecessor slice to the event within the cluster. The EventPosition CR contains a reference to the greatest cluster-receive causally prior (reflexively) in this trace to the given event. We note briefly that the arrays could be built from SliceRef objects. This would be a bad idea as that class was designed for external usage, providing garbage collection thus requiring an extra dereference per operation. However, for convenience we will assume that TraceIDs map to integers from 0 to $N - 1$ where N is the number of traces, allowing us to index arrays by TraceID. Any reasonable implementation of the TraceID class would facilitate such an approach. Alternately, we could have used integers rather than EventPositions, as this is internal data. There are two reasons we choose not to. First, it makes the algorithm and code clearer. Second, a well-crafted implementation of the EventPosition class should require no greater cost that the integer class, but would allow greater flexibility. Specifically, it would enable the trivial change of encoding from short to integer to long to arbitrary length. Likewise, it facilitates code debugging.

The Cluster class itself is defined as follows.

```
class Cluster {
public:
  Cluster& merge(EventRef&);
  bool     mergeable(EventRef&);
  TraceID& map(int);
  int      inverseMap(TraceID&);
  bool     contains(TraceID&);
  int      size();
  EventPosition* project(EventPosition*);
};
```

The merge() method merges the various clusters in which the argument event and its partners occur, returning a reference to the resulting cluster. The mergeable() method determines, according to the rules of the clustering strategy, if the various clusters in which the argument and its partners occur can be merged. It returns true if they can, and false otherwise. It does not perform the merger, but merely determines its feasibility. The map() method maps CT array positions to TraceIDs, while the inverseMap() method performs the inverse function. The contains() method returns true if the cluster contains the given TraceID, and false otherwise. It is strictly redundant, since the inverseMap() method must return some indication of failure in the event that the TraceID requested is not present in the cluster. However, we prefer to maintain a separation of concerns at the interface-level of a class. The inverseMap() will therefore expect the TraceID to be present, and will throw an exception if it is not. The size() method returns a count of the current number of traces contained within the cluster. Finally, the project() method accepts an EventPosition array interpreted as a Fidge/Mattern timestamp, and returns the projection of that timestamp over the traces in the cluster.

The algorithms for these methods are the subject of Section 9.2. For now, it is sufficient to know what they do, and observe that the cost should not be appreciably worse than $O(1)$[1] for the mergeable(), contains(), (inverse) mapping, and size() methods and $O(c)$, where c is the cluster size, for the merge() and project() methods.

Third, we define the GCR array. This global array of EventPositions identifies the current greatest cluster-receive event in each trace. Initially the elements are set to EventPosition(0).

We now describe the actual timestamp-creation algorithm. It first computes the Fidge/Mattern timestamp (line 2) using the method described in Section 6.1.2. For events that are cluster-receives (lines 3 and 4) and whose clusters are not mergeable (line 5), we use the Fidge/Mattern timestamp (line 6), and adjust the vector of greatest cluster-receives to identify this new event (line 7). An event that does not satisfy the **if** condition is either not a cluster-receive or its cluster is mergeable with that of its partner event. If it is the latter (lines 9 and 10), then the two clusters are merged (line 11), rendering the event no longer a cluster-receive. Thus, on line 12 no event is a cluster-receive. The algorithm therefore takes the projection of the event's Fidge/Mattern timestamp over the cluster as the array portion of its cluster timestamp. To enable precedence determination beyond the cluster, the algorithm also records the greatest preceding cluster-receive in the event's trace (line 14). In the case of cluster-receive events, this will be a self-reference.

Those events with EventPosition(0) (that is, the fake event that precedes all other events on its trace) are deemed to be cluster-receive events with a Fidge/Mattern timestamp of all zeros. This is necessary to ensure the correct operation of the precedence test.

Lines 5 and 9 to 11 are only needed for dynamic clusters. Static clusters cannot be merged, and thus will never satisfy the mergeable() condition.

The computation cost of this algorithm is $O(N)$, where N is the number of traces. This cost arises because of the need to compute the Fidge/Mattern timestamp in line 2. To extend this algorithm to the multi-partner environment, the **if** conditions would need to check each partner to determine if the event was a cluster-receive. Merging would require that the clusters of all partners could be merged. The algorithm is otherwise unaltered, and would remain $O(N)$ cost as the Fidge/Mattern computation would still dominate.

Finally, we must delete Fidge/Mattern timestamps when they are no longer needed. They may be disposed of by using the technique used for dynamic critical-pair computation (Figure 8.6, lines 4 to 6), applied at any point in the code after line 12. Alternately, we might register a callback for events that are covered, as follows.

```
callbackCleanup c;
po.registerCallback(PartialOrder::cbType::covered, c);
```

We would then provide a callbackCleanup class as shown in Figure 9.3. The Fidge/Mattern timestamp is not deleted if the event is a cluster receive, since the timestamp will be pointed to by the CT element.

[1]By this phrase we mean that the computation cost should approximate a small constant in practice, even if it is not in theory. For example, event lookup, which is $O(\log n)$ in theory, takes two, or at most three steps in practice (see the discussion of this point in Section 5.2, page 75).

```
class callbackCleanup : public PartialOrder::Callback {
public:
  ˜callbackCleanup() {};
   bool callback(const TraceID&) { return false; };
   bool callback(...) { return false; };
   bool callback(const EventRef& e) {
     if (e._event->CT.CT != e._event->FM)
       delete [] e._event->FM;
     e._event->FM = 0;
   };
};
```

Figure 9.3: Fidge/Mattern Cleanup Callback

The precedence-test algorithm for the two-level cluster timestamp is shown in Figure 9.4. The basic principle behind the algorithm is as follows. If the **this**-event trace is in the same cluster as the argument e then the standard Fidge/Mattern test applies, with parameters adjusted to map the traces to timestamp entries appropriately. If it is not, then precedence is true if-and-only-if there is a cluster-receive that precedes the argument e and is a successor to the **this** event. Such a cluster-receive would have to be a predecessor to the (reflexive) greatest predecessors within the trace of the argument e.

We now walk through the code line by line. In line 2 we determine what the relevant entry for the **this** event's Fidge/Mattern timestamp would be if it maintained such a timestamp. This is identical with its EventPosition, *per* the definition of these timestamps (see Section 6.1.2). Now, if the argument e is a cluster-receive (line 3), then its cluster-timestamp array will be identical to its Fidge/Mattern timestamp, and so the algorithm can simply index the relevant entry and return the result *per* Equation 6.5 (line 4). Likewise, if event e is not a cluster-receive but the **this** event is in the same cluster as event e (line 5), then a simple inverseMap() operation determines what the relevant Fidge/Mattern entry would be for the test to apply (line 6). Note that because of cluster merging, the test is not the same requirement as the two events being in the same cluster. It is quite possible, indeed with dynamic clusters it invariably happens in practice early in the trace, that the cluster of the **this** event is not in the same as that of event e, even though its trace is in e's cluster. We will elaborate on this point in Section 9.2. Suffice it to say for the present, containment is the required test.

Should the code reach line 7, then the clusters of the two events are clearly distinct. Hitherto the algorithm has taken the approach of computing what the Fidge/Mattern entry for event e would be if it maintained such a timestamp. At this juncture it takes a different tack. Since the events are in different clusters, the objective is to determine if there is a cluster-receive prior to e that is a successor of the **this** event. To do this, the algorithm loops over the greatest-predecessor events of e within the cluster (line 7). Line 9 retrieves each such event g (note that the Fidge/Mattern timestamp definition records one position greater in each trace, other than in the trace of the event timestamped, than the reflexive greatest-predecessor), while line 10 then

```
 1: bool operator<(EventRef& e) {
 2:   EventPosition e1 = eventPosition();
 3:   if (e._event->CT.CR == e.eventPosition())
 4:     return (e1 < e._event->CT.CT[traceID()]);
 5:   if (e.cluster().contains(traceID()))
 6:     return
          e1 < e._event->CT.CT[e.cluster().inverseMap(traceID())];
 7:   for (int j = 0 ; j < e.cluster().size() ; ++j) {
 8:     TraceID jTrace = e.cluster().map(j);
 9:     EventRef g = _event->_block.po(jTrace, e._event->CT.CT[j]
                                     - (jTrace == TraceID ? 0 : 1));
10:     EventRef r = _event->_block.po(jTrace, g._event->CT.CR);
11:     if (e1 < r._event->CT.CT[traceID()]);
12:       return true;
13:   }
14:   return false;
15: }
```

Figure 9.4: Two-Level Dynamic Cluster-Timestamp Precedence Test

identifies the greatest cluster-receive r reflexively prior to such g events. If the **this** event precedes any such r event (line 11), then precedence holds, and the algorithm can return true (line 12). If no such event exists, then the **this** event does not precede e (line 14). Note that if we actually wanted to compute the Fidge/Mattern entry, then rather than have an **if**-test in line 11 and return in line 12, we would take the maximum of EventPositions over the whole loop. That maximum would represent the Fidge/Mattern value for event e at position `this->traceID()`.

Finally, note that the partial order must be known to execute this test. Specifically, lines 9 and 10 perform an event-retrieval operation. As noted in the discussion of the cluster() method, the EventRef::Data object knows the block of which it is a part, which in turn knows the partial order. There are two significant aspects to these retrievals. First, the ability to retrieve events with EventPosition(0) is necessary to ensure the correct operation of the precedence test. Specifically, the EventPosition stipulated in either line 9 or 10 could be EventPosition(0). In practice, the precedence test would skip any such retrieval and continue the loop. Since that unnecessarily obscures the essence of the algorithm we have chosen to show it this way. Second, the requirement for retrievals indicates that the precedence test is centralized, not distributed, for any between-cluster test.

The computation cost of this test depends on the relative locations and precedence relationships of the two events. If the events are in the same cluster or if the argument e is a cluster-receive, the cost is constant-time. If this is not the case, then the cost is O(e.cluster().size()), being on average less than half this if **this** $\prec_{\mathcal{E}}$ e. These costs presume that the various operations, in particular contains() and inverseMap(), can be performed in constant time. This point is discussed in Section 9.2.

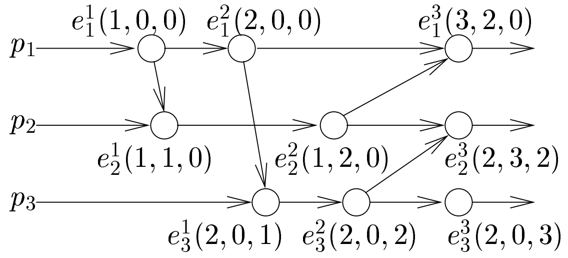


Figure 9.5: Variant Fidge/Mattern Timestamps

9.1.2 HIERARCHICAL ALGORITHM

We now extend the two-level algorithm to an arbitrary hierarchy of levels. The core idea 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. This allows a tradeoff between the number of cluster-receives, the size of the cluster, and the size of the cluster-receive timestamp. In the two-level algorithm, there is only a tradeoff between the first two of these.

First we must deal with a subtle detail of the Fidge/Mattern timestamp, as defined in Section 6.1.2. As defined in that section, the elements of the timestamp correspond to one greater than the greatest-preceding event position for every entry except the zero entries and the entry corresponding to the trace of the event being timestamped. While easy to compute, this unnecessarily complicates the following algorithms, which are sufficiently complex as is. Therefore we define the variant Fidge/Mattern timestamp as follows.

$$\mathcal{FM}'(e) = \max_{e^c <: e} (\mathcal{FM}(e^c)) \tag{9.1}$$

$$\mathcal{FM}(e)[i] = \begin{cases} \mathcal{FM}'(e)[i] + 1 & \text{if } i \in \phi(e) \\ \mathcal{FM}'(e)[i] & \text{otherwise} \end{cases} \tag{9.2}$$

(Recall that $\phi(e)$ maps event e to its traces.) This then means that the timestamp records precisely the reflexive greatest-predecessor set of event e. The precedence test is

$$e^i \preceq_{\mathcal{E}} e^j \iff \exists_{p\in\phi(e^i)}\ \mathcal{FM}(e^i)[p] \le \mathcal{FM}(e^j)[p] \tag{9.3}$$

An example of this variant Fidge/Mattern timestamp is shown in Figure 9.5, which the reader may wish to compare with our original example shown in Figure 6.2. For the remainder of this chapter this variant will be used as the Fidge/Mattern timestamp. Likewise, when we refer to the greatest predecessor, greatest-predecessor set, causally prior, precedes or any other such variant on the notion of precedence, we will mean the reflexive form, *per* Equation 9.3.

The code for hierarchical dynamic cluster-timestamp creation in the single-partner environment is shown in Figure 9.6. As with the two-level algorithm, we first describe the changes to existing classes and data necessary to support this code. These changes encompass the definition

```
void timestamp(EventRef& e) {
  e._event->FM = fidgemattern(e);
  int k = 0;
  while ((e.receive() || e.synchronous()) &&
         !e.cluster(k).contains(e.partner().traceID()) &&
         !e.cluster(k).mergeable(k,e)) {
    GCR[k][e.traceID()] = e.eventPosition();
    ++k;
  }
  if ((e.receive() || e.synchronous()) &&
       (e.cluster(k).uc() ||
       !e.cluster(k).contains(e.partner().traceID())))
    e.cluster(k).merge(k,e);
  e._event->CT.level = k;
  e._event->CT.CR = GCR[k][e.traceID()];
  e._event->CT.CT = e.cluster(k).project(e._event->FM);
}
```

Figure 9.6: Hierarchical Dynamic Cluster-Timestamp Creation

of cluster-receive, the cluster() method and Cluster class, the EventRef::Data::CT class, and the GCR data.

First, note that we can no longer identify an event as belonging to just one cluster. Rather, an event belongs to a series of clusters, extending out to a cluster that composes the entire computation. Thus clusters are assigned a level starting at 0, which is the innermost cluster. We redefine the cluster() method to take an integer parameter k, and return a reference to the level-k Cluster object to which invoking event belongs.

The outer-level cluster that encompasses the entire computation is referred to as the universal cluster. Any attempt to find the level-k cluster for an event which has no cluster at level-k will return this universal cluster. However, this cluster does not have the expected properties of a cluster that explicitly encompasses all traces. Rather, this is an implicit cluster. It may be thought of as the least upper-bound of all clusters. The relevant properties of this cluster are that its size is one and the containment test is true for any valid TraceID. The first property guarantees that events in different traces whose only common cluster is the universal cluster will be deemed to be mergeable, where a size constraint on clusters would otherwise prevent the merger. More precisely, a new cluster will be created at this new highest level. The second property guarantees that a common cluster is found for precedence-testing purposes. Our algorithms do not require (inverse) mapping or projection of this universal cluster, and so they are left undefined. This universal cluster is only relevant for dynamic clusters.

We likewise define the term “level-k cluster timestamp” to refer to a timestamp that encompasses exactly those traces of the level-k cluster. While such a timestamp will effectively have entries for the level-0 to level-$(k-1)$ clusters, the mapping and inverse-mapping functions for those clusters would not work correctly on the level-k cluster-timestamp. Thus, `c.inverseMap(t)` will yield the array index for the `t`-TraceID entry into the cluster-timestamp at level `c.level`. It will not yield the correct index into timestamps at other levels. It is therefore very important that mapping and inverse mapping be applied to clusters at the correct level for the cluster-timestamp array.

The Cluster class is augmented with the uc() method. In addition, a new project() method is added, to provide the projection of a level-k cluster-timestamp to a level-k' cluster-timestamp, where $k' \leq k$, and the merge() and mergeable() methods take different arguments. The formal interface is

```
class Cluster {
...
   bool uc();
   EventPosition* project(const EventRef&);
   void merge(int, const EventRef&);
   void mergeable(int, const EventRef&);
}
```

The uc() method returns true if the cluster is the universal cluster, and false otherwise. The project() method returns the projection of the cluster-timestamp of the event over the traces in the cluster. The merge() and mergeable() methods must now take the cluster level as well as the event as parameters, as level-k cluster merging is predicated on level-$(k + 1)$ merging. We discuss this point further in Section 9.2.2.
This change in cluster membership requires us to redefine the term “cluster-receive.”

Definition 27 (Level-k Cluster-Receive) *An event e is a level-k cluster-receive if-and-only-if it is a receive event with a partner event on a trace in a different level-k cluster or a synchronous event whose home traces occur in different level-k clusters.*

Note that a level-k cluster-receive is, by this definition, also a level-0 to level-$(k - 1)$ cluster-receive. This is a very significant aspect of this definition, since maintaining cluster-receive information correctly is crucial to ensure that transitive dependencies are captured properly.

A level-k cluster-receive will require a level-$(k+1)$ cluster-timestamp. Since a level-k cluster-receive is also a level-0 to level-$(k-1)$ cluster-receive, this implies that it needs level-1 to level-k cluster-timestamps as well as the level-$(k + 1)$ cluster-timestamp. However, such timestamps would be redundant, as the level-$(k+1)$ cluster-timestamp would contain all the necessary information. When a lower-level timestamp is required, the project() method will be applied.

For the various reasons above, we can no longer maintain just one GCR array, but must maintain such an array for every cluster level. All of these arrays are initialized to EventPosition(0). Likewise, a level-k cluster-receive does not update just the level-k GCR array, but all such arrays from level-0 to level-k (line 7), since it is also a cluster-receive at all those lower levels.

Finally, the EventRef::Data::CT class must be augmented with an integer member `level` that identifies the cluster-level at which the CT array is a cluster-timestamp, and thus the level of cluster-receive to which the CR member refers.

We now walk through the code, line by line. As with the two-level algorithm, it starts by computing the Fidge/Mattern timestamp of the event (line 2). It then ascertains the cluster-level at which the event and its partner are either in the same cluster or the different clusters are mergeable (lines 3 to 9). In the course of doing so, it updates the GCR arrays, since for every level k at which the event and its partners are in separate non-mergeable clusters, the event is a level-k cluster-receive. On exiting the loop (line 10), the value of k is equal to the cluster level in which the event and its partner are in the same cluster or mergeable clusters. Lines 10 to 12 determine if it is the latter condition, and, if so, line 13 will merge the two clusters. Thus, by line 14 the event and its partners are in the same level-k cluster. Line 14 records that cluster level for use in the precedence test. The greatest current level-k cluster-receive in the trace, recorded in the GCR structure, is recorded for the event. Finally, the projection of the Fidge/Mattern timestamp over the event's level-k cluster is recorded as its cluster-timestamp array.

As with the two-level algorithm, those events with EventPosition(0) are deemed to be cluster-receive events at the highest cluster level, with a timestamp of all zeros. This is necessary to ensure the correct operation of the precedence test.

Note that the algorithm as defined implicitly forces a merge on first communication. Specifically, the **if** test of lines 10 to 12 will always succeed if the event and its partners are in different clusters at all levels of the hierarchy. In effect, this forces a merge on first communication regardless of the outcome of the mergeable() test of line 6. The specific outcome depends on the definition of the merge() method. As we have defined it in Section 9.2.2, the result will either be a merging of the two clusters or the creation of a new parent cluster for the two clusters. This is not desirable, since the intent of the algorithm is to allow arbitrary merging criteria and not force merging on first communication. There is no conceptual difficulty in fixing this problem. We can simply identify this special case and assign the event its Fidge/Mattern timestamp as its cluster timestamp. However, we have left the algorithm as is for two reasons. First, the fix obscures the essence of both this algorithm and the algorithms that it is dependent upon. In particular, both the precedence test and the various Cluster-class methods require a substantial amount of special-case code, reducing their comprehensibility. Second, we are currently unconvinced that this is the correct solution to apply. An alternate, and we believe more attractive, solution is to resolve the problem of trace-movement between clusters. We can then allow merging on first communication and correct any such mergers that prove to be poor.

Lines 6 and 10 to 13 are only needed for dynamic clusters. Static clusters cannot be merged, and thus will never satisfy the mergeable() condition. Likewise, static clusters are expected to specify a complete cluster hierarchy, and thus every event will have some cluster, other than the universal cluster, that contains every trace. As such, lines 11 and 12 will always return false for static clusters, and thus the merge() method will never be invoked in line 13. Note also that the algorithm effectively reduces to the two-level algorithm if level-1 in the hierarchy is a cluster that contains all traces. Any cluster-receive event (a level-0 cluster-receive in this scheme), would reach line 10 with $k = 1$ while all other events would have $k = 0$ at that point. The projection of a level-1 cluster Fidge/Mattern timestamp would be identical to that timestamp.

The computation cost of this timestamp-creation algorithm remains $O(N)$ for the single-partner environment. First, it must be at least $O(N)$ because it still requires the calculation of the Fidge/Mattern timestamp for each event (line 2). Second, the loop that determines the cluster level at which the event and its partners are in the same cluster (lines 4–9) will be executed k times, where k is that common cluster level. Since all operations within that loop have a cost that is not appreciably worse than $O(1)$ and $k \ll N$, the cost of this loop is then negligible. Third, the merging of clusters (lines 10–13) will be no worse than linear in the size of the merged cluster, per our requirement on the merge() method. A merged cluster can contain no more than N traces and so this merge() cost is no worse than $O(N)$. Lines 14 and 15 are both $O(1)$ operations. Finally, the Fidge/Mattern timestamp projection of line 15 is likewise constrained to be no worse than linear in the size of the projected timestamp, which must be no bigger than size N. As nothing in the remainder of the algorithm is worse than $O(N)$, the algorithm cost is $O(N)$.

The deletion of Fidge/Mattern timestamps would be identical with the callback approach used in the two-level algorithm (Figure 9.3), though the **if** test is not necessary, as the `CT` pointer never points to the `FM` array.

The precedence-test algorithm for hierarchical cluster timestamps is shown in Figure 9.7. Note that the increased complexity of this precedence test arises in lines 6 to 20. Where the previous algorithm simply used the cluster timestamp of the argument event to determine the cluster-receive timestamps, this algorithm must compute the relevant cluster-receive events. It does so by repeatedly computing the maximum of greatest-predecessor cluster-receive timestamps in each cluster level as it moves up the cluster hierarchy. It stops at the cluster prior to that cluster which contains the necessary precedence information. It then effectively invokes the previous algorithm (lines 22 to 29). If the hierarchy only consists of two levels, the **while** loop of lines 7 to 20 will never execute, and the algorithm reduces to the two-level precedence test.

We now walk through the code line by line. As with the two-level algorithm, the first thing the precedence test does is to compute the value of the **this** event's Fidge/Mattern timestamp at the relevant location to apply the Equation 6.5 precedence test (line 2). It then determines if the argument event e also has an entry in its cluster timestamp for the relevant location (lines 3 and 4). If it has, then the algorithm applies the test and returns the result (line 5).

If, however, event e's cluster-timestamp does not contain the required trace, the algorithm must work up through the cluster hierarchy until it reaches a cluster that does contain the **this** event's trace. It starts at the cluster-level of event e's cluster-timestamp (line 3), and assigns the current timestamp (CTS) to e's cluster-timestamp (line 6). Using this timestamp, it retrieves the greatest predecessors of e within cluster level-k (line 11). It then determines the greatest level-k cluster-receive that precedes each of these events (line 12). The code to achieve this looks something like the following.

```
const EventRef& gcr(int k, EventRef& g) {
  int i = g._event->CT.level;
  while(i <= k) {
    g = g._event->_block.po(g.traceID(), g._event->CT.CR);
    i = g._event->CT.level;
  }
```

```
bool operator<=(EventRef& e) {
  EventPosition e1 = eventPosition();
  int k = e._event->CT.level;
  if (e.cluster(k).contains(traceID()))
    return
      e1 < e._event->CT.CT[e.cluster(k).inverseMap(traceID())];
  EventPosition* CTS = e._event->CT.CT;
  while (!e.cluster(k+1).contains(traceID()) {
    EventPosition* NTS = 0;
    for (int j = 0 ; j < e.cluster(k).size() ; ++j) {
      TraceID jTrace = e.cluster(k).map(j);
      EventRef g = _event->_block.po(jTrace, CTS[j]);
      EventRef r = gcr(k,g);
      NTS = max(e.cluster(k+1).size(),
                e.cluster(k+1).project(r),
                NTS);
    }
    ++k;
    if (CTS != e._event->CT.CT)
      delete [] CTS;
    CTS = NTS;
  }
  for (int j = 0 ; j < e.cluster(k).size() ; ++j) {
    TraceID jTrace = e.cluster(k).map(j);
    EventRef g = _event->_block.po(jTrace, CTS[j]);
    EventRef r = gcr(k,g);
    if (e1 < r._event->CT.CT[r.cluster(r._event->CT.level).
                                     inverseMap(traceID())])
      return true;
  }
  return false;
}
```

Figure 9.7: Hierarchical Dynamic Cluster-Timestamp Precedence Test

```
  return g;
}
```

Observe that because every level-k cluster-receive is a level-0 to level-$(k-1)$ cluster receive, this algorithm will terminate within k iterations of the **while** loop. The variable i will monotonically increase with each loop iteration. Note also that this is not the only way in which the greatest level-k cluster-receive might be identified, though it is as we have defined the timestamp-creation algorithm. The alternate solution would be to have every event identify its greatest cluster-receive at every level in the hierarchy. While this sounds space-consumptive, it is unlikely that there would be more than four or five levels in the hierarchy, for reasons that will become apparent when we analyse the computation cost of the precedence test. This choice represents a binary time/space tradeoff, with no clear basis for the choice. If the hierarchy is shallow, the space-consumption will be small, but so also will be the cost of computing the required cluster-receive. If the hierarchy is deep, the space-consumption will be large, as will be the computation cost.

The algorithm must then determine the TraceID-wise maximum of each of these cluster-receive events (line 13). Several points need to be made about this maximum operation. At a mundane level, the max() function performs a simple array comparison, thus.

```
EventPosition* max(int sz, EventPosition* a1, EventPosition* a2)
{
  if (a2 == 0) {
    a2 = new EventPosition[sz];
    for (int i = 0 ; i < sz ; ++i)
      a2[i] = a1[i];
  } else {
    for (int i = 0 ; i < sz ; ++i)
      a2[i] = a1[i] > a2[i] ? a1[i] : a2[i];
  }
  return a2;
}
```

The more significant aspects of the maximum operation are in the arguments. First, note that the size parameter is that of the level-$(k+1)$ cluster-timestamp. Event e will have no such timestamp, but it will be part of such a cluster, and that cluster will have a specified size and associated set of traces at the time that e was timestamped. However, not every level-k cluster-receive is guaranteed to have entries in its level-$(k+1)$ cluster-timestamp array for all of the positions specified by the event e level-$(k+1)$ cluster. This is because of cluster merging. The cluster-receive might well have been timestamped prior to a lower- or same-level cluster merge. On the other hand, the level-k cluster-receive might have many more entries in its cluster-timestamp array than does e's level-$(k+1)$ cluster. This is because it may be a level-$(k+1)$ cluster-receive as well as a level-k cluster-receive. The function, then, of the project() method that is used in argument 2 (line 14), is to ensure that any gaps are filled and no extraneous elements present. It thus provides a uniform-length timestamp, with consistent mapping of the array elements corresponding to those specified by event e's level-$(k+1)$ cluster, for all cluster-receive events, allowing the element-wise maximum operation to be correct. The algorithm for the project() method is described in Section 9.2. It has cost O(size of the output array).

Second, note that we have stated that event e will have no level-$(k+1)$ cluster-timestamp, but that it will be part of such a cluster, and by this we mean a cluster other than the universal cluster. At the relevant line of code (line 13), this statement will be true. If event e has no explicit level-$(k+1)$ cluster, then the `e.cluster(k`1)+ invocation would return the universal cluster (line 7). It in turn would return true to the TraceID containment query (line 7), thus exiting the **while** loop, continuing execution at line 22.

After the **for** loop has finished iterating, the NTS variable will contain the greatest predecessors reflexively-prior to event e in the level-k cluster (line 16). The algorithm then increments the cluster level (line 17), and sets the current timestamp to the value of this new level-k timestamp (line 20), cleaning up the old current timestamp first (lines 18 and 19).

The **while** loop will continue iterating if the next cluster-level beyond k does not contain the required TraceID (line 7). Thus, on exit of the **while** loop, the CTS array contains a level-k timestamp which identifies the greatest predecessors in e's level-k cluster to event e. This is the equivalent of the projection of e's Fidge/Mattern timestamp over e's level-k cluster.

This is effectively the identical situation to the two-level precedence test when it was at line 7. The remainder of the algorithm is therefore effectively the same. It iterates over the resultant CTS timestamp (line 22), retrieving the greatest predecessor in each trace (line 24), and using that to determine the greatest level-k cluster-receive (line 25). It exits, returning true, if it finds such a cluster-receive that succeeds the **this** event (line 26). If no such event is found, the algorithm returns false.

As with the two-level precedence test, the computation cost of this test depends on the relative locations and precedence relationships of the two events. If the cluster-timestamp of event e contains an entry for the **this** event's trace, the cost is constant-time. When this is not the case, the cost depends on how far up the cluster hierarchy the algorithm must go to find a cluster that encompasses both events. If this point is reached at level-2, then the cost is `O(e.cluster(1).size())`, being on average half this if **this** $\preceq_{\mathcal{E}}$ e. In the general case, going beyond level-2 clusters, we must compute additional timestamps, and this can be expensive. As such, this cost dominates. If the cluster that encompasses both events occurs in level-k, then line 13 of the algorithm is executed `O(e.cluster(k-2).size())` times and each time requires `O(e.cluster(k-1).size())` operations. The total cost of this algorithm is therefore `O(e.cluster(k-1).size()*e.cluster(k-2).size())`. If we wish to ensure that this cost is always less than `O(e.cluster(k).size())` we must increase the cluster-size at each level to the power of $(1+\sqrt{5})/2$ (about 1.62) or more.

It is worth noting that the algorithm effectively corresponds to a breadth-first search through the cluster-receive space. This begs the question as to whether an alternate strategy, notably depth-first search, might be better. The breadth-first algorithm is extremely predictable, as noted above, while a depth-first approach will depend heavily on the specific nature of the cluster-receive events. The best case would be much faster, completing in time proportional to the shortest cluster-receive path between the events; in other words, in time proportional to the height of the cluster hierarchy. On the other hand, the worst case would examine a number of cluster-receive events unnecessarily, that the max() function of line 13 eliminates in our algorithm.

9.1.3 Eliminating Fidge/Mattern Timestamp Generation

As we observed in our description of the precedence-test algorithm, if the events have a common cluster at level-k, then that algorithm effectively computes the projection of the Fidge/Mattern timestamp to level-$(k-1)$ and then applies the two-level algorithm. It is lines 6 to 21 that are computing this level-$(k-1)$ projection. Given this, we do not need to compute the full Fidge/Mattern timestamp for each event, as we are currently doing on line 2 of the timestamp-creation algorithm of Figure 9.6. We therefore alter that algorithm as follows. First, we remove line 2. Second after line 14 we know that we require a level-k cluster-timestamp. Note that line 16 is the first, and only, point at which we use the Fidge/Mattern timestamp. We therefore developed a projection() function, shown in Figure 9.8. This function takes an event e and a cluster-level i to which e's cluster-timestamp is to be projected. Lines 3 and 4 deal with the possibility that e's cluster-timestamp already exceeds the required level, and use the Cluster project() method (to be described in Section 9.2) to extract the required timestamp. Lines 5 to 20 are essentially a replica of lines 6 to 21 of the precedence test, with just the condition on the **while** loop changed to match the requirement of the function. On exiting the **while** loop the desired timestamp has been computed, and is returned (line 21). The cost of this algorithm is the product of the sizes of the level-i and level-$(i-1)$ clusters.

The remainder of the timestamp-creation algorithm is altered as follows. After line 14 each event covered by the event e being timestamped has its level-k cluster timestamp computed by the above projection() function. In line 15 `e._event->CT.CT` is assigned the element-wise maximum of these projections. It is then incremented by one in every element-location that corresponds to $\phi(e)$.

In our original timestamp-creation algorithm the computation of the Fidge/Mattern timestamp dominated the algorithm cost. Since this has been removed, we must re-analyse the cost. As with our original, it is clear that the cost of determining the cluster-receive level will be negligible. Cluster merging is required to be no more expensive than linear in the merged cluster size. Thus, the projection cost will determine the timestamp-creation cost. If the event being timestamped is not a cluster receive, then the cost will be proportional to the level-0 cluster size. If it is a level-k cluster receive, then the cost will be proportional to the cost of projecting the covered events' timestamps to level-k, and then taking the maximum over those timestamps. Assuming a single-partner environment, there will be two such events, either of which may need to have its timestamp projected, at a cost of `O(e.cluster(k).size()*e.cluster(k-1).size())`.

For level-k cluster-receive events where k is near the top of the cluster hierarchy, this approach can be more costly than the Fidge/Mattern computation. Specifically, in a two-level environment it is identical to the Summers mechanism for regenerating the Fidge/Mattern timestamp. That method is `O(N(e.cluster().size()))`. However, for non-cluster-receive events, the cost is `O(e.cluster().size())`, which is substantially better than the Fidge/Mattern

```
EventPosition* projection(EventRef& e, i) {
  int k = e._event->CT.level;
  if (k >= i)
    return e.cluster(i).project(e);
  EventPosition* CTS = e._event->CT.CT;
  while (k < i) {
    EventPosition* NTS = 0;
    for (int j = 0 ; j < e.cluster(k).size() ; ++j) {
      TraceID jTrace = e.cluster(k).map(j);
      EventRef g = _event->block.po(jTrace, CTS[j]);
      EventRef r = gcr(k,g);
      NTS = max(e.cluster(k+1).size(),
                e.cluster(k+1).project(r),
                NTS);
    }
    ++k;
    if (CTS != e._event->CT.CT)
      delete [] CTS;
    CTS = NTS;
  }
  return CTS;
}
```

Figure 9.8: Cluster-Timestamp Projection to Level-i

computation cost. Likewise, level-k cluster-receives where the cluster is not near the top of the hierarchy will be more efficiently computed using this method. It can be shown that for the two-level hierarchy, if the ratio of cluster-receive events to non-cluster-receive events is less than `1/e.cluster().size()` then this approach will be beneficial. In general, the entire approach of this suite of algorithms is premised on the observation that cluster-receives are notably less frequent than other events.

9.1.4 PRECEDENCE-RELATED EVENT SETS

We must now deal with the second half of the precedence problem, determining precedence-related event sets. The approach we take is essentially the same as was used with Fidge/Mattern timestamps (Section 6.2). That is, we must compute the greatest-predecessor set, and the remaining sets will be computed based on that, using the algorithms described in that section. The algorithm we use for greatest-predecessor determination is the same cluster-timestamp projection algorithm used to eliminate Fidge/Mattern timestamp generation (Figure 9.8). We simply require that the cluster level be that level which encompasses the entire computation. The cost is as described in the previous section.

Having described these algorithms, we observe that they are probably not a good choice of interface for our timestamp algorithm. Specifically, there is no partial-order-interface method that exists between event-precedence determination and greatest-predecessor set. Thus, if we wish to compute some number of greatest-predecessors, but not all, we have a poor choice. The greatest-predecessor set is somewhat expensive to compute using our timestamps, but the individual tests cannot take advantage of the aggregate nature of the query. Further, this is not a contrived case. Specifically, if we wish to indicate the predecessor events in a display, we would only wish to compute the predecessors for those traces that are currently visible. An extended interface to the data structure is clearly required, but is beyond the scope of this work.

9.2 Clustering Algorithms

We now turn to the problem of providing the Cluster-class and cluster() methods. This has been deferred until now as it is dependent on the cluster-strategy employed. Good cluster strategy is crucial to the quality of the timestamp algorithm. While the timestamp algorithm is able to exploit communication locality, it is incumbent on the cluster strategy to determine that locality.

The strategies can be broadly divided into two categories: static and dynamic. A static strategy is one in which, for the purpose of the timestamp algorithm, the clusters are pre-determined and will never change during the execution of the algorithm. Various methods satisfy this constraint. A user-defined-cluster approach would likely be static, specifying the mapping from traces to clusters. While such clusters in principle could be optimal, it is not reasonable to require user involvement in core data-structure issues. However, computation-specific information might be extracted through source-code analysis. This could then be provided to our timestamp algorithm to identify the cluster-hierarchy to which a trace belonged.

Alternately, we might employ a delayed timestamping scheme. In this technique we first capture some small number of events on each trace and cluster the traces according to the communication so captured. That is, we would use the current state of something equivalent to POET's communication matrix as our similarity metric. Any standard hierarchical clustering algorithm can be applied, though some care should be taken as a naive algorithm would be $O(N^2)$ or worse [4, 75]. The effect of a poor choice would increase the delay prior to timestamping. Not all traces are necessarily present at the start of a computation. However, it would be sufficient for this algorithm if the clusters are defined prior to the arrival (at the timestamp algorithm) of an event for a previously unknown trace. This would have some implications on the clustering algorithm. Specifically, it would have to be dynamic with respect to the arrival of new traces, though it could not alter existing cluster relationships. Insofar as the initial communication events between traces are reflective of subsequent clustering in the computation, this would be an effective approach.

A third possibility would be to simply select arbitrary contiguous ranges of traces. Such an approach would hardly be ideal in capturing communication locality, though it does have the virtue of extreme simplicity. In addition, if either of the previous two techniques were employed, it would be a simple matter to re-map the ordering of their traces such that the clusters they provided covered contiguous ranges. Thus, the Cluster-class methods that we develop for this technique are equally applicable to those static approaches. Further, by employing this technique for our static-algorithm experimental results we were provided with a nice lower bound on the performance we can expect from our timestamp algorithm.

No static technique can be perfect. Source-code analysis is target-specific, and thus less-attractive. Further, it is limited in the information it can extract by lack of knowledge of run-time data. Delayed timestamping has an unattractive transient startup time, and the quality of clusters produced will be dependent on the initial communication being reflective of the clustering. The only virtue of arbitrary contiguous clusters is simplicity. We therefore look at non-static alternatives.

Dynamic cluster strategies are ones in which the clusters are not known *a priori*, but are determined by the run-time behaviour of the computation. This makes the strategy dynamic in a very unusual sense. We wish to cluster traces, and thus they are the fundamental elements over which the clustering algorithm is applied. However, it is often the case that many of the traces will be known early in the computation. What is not known is the communication pattern that will occur between those traces. That is, the dynamic aspect is the similarity metric. It is changing with each communication event. Thus, what initially appears to require some form of hierarchical agglomerative clustering [31, 161] is apparently quite different. Further, we are constrained by the required performance of the Cluster-class methods, all of which must be O(size of the output) which is needed to maintain the dynamic properties of the timestamp algorithm, the requirement that every trace is a member of a cluster at any given instant, and the implicit limitations on space-consumption. We are not aware of any algorithm that satisfies all of these requirements. Indeed, the mere need to satisfy the performance requirement would substantially constrain the choices. We therefore developed our own approach, which we describe in Section 9.2.2.

We have developed two cluster strategies, fixed clusters and self-organizing clusters, which are static and dynamic, respectively. We now describe these strategies, and their associated Cluster-class methods.

9.2.1 FIXED CLUSTERS

We now describe the algorithms required to implement the Cluster-class and cluster() methods for the fixed-cluster strategy. As defined above, these clusters are contiguous over a range of traces in the Fidge/Mattern timestamp. We therefore assign each cluster an offset that indicates the starting element in the Fidge/Mattern array. The size is already required by the timestamp algorithm, and thus the termination point of the range is determinable. We now briefly recollect the methods that we must provide, together with the private data that we can now specify.

```
class Cluster {
public:
  bool     uc();
  Cluster& merge(EventRef&);
  Cluster& merge(int, EventRef&);
  bool     mergeable(EventRef&);
  TraceID& map(int);
  int      inverseMap(TraceID&);
  bool     contains(TraceID&);
  int      size();
  EventPosition* project(EventPosition*);
  EventPosition* project(const EventRef&);
private:
  int _offset;
  int _size;
};
```

The private data is self-explanatory.

The uc() method determines whether or not the cluster is the universal cluster, returning true if it is, and false otherwise. The universal cluster is identified by having an offset of -1.

The merge() and mergeable() methods are straightforward, as these clusters are fixed and so no merging can ever take place. Thus, mergeable() simply returns false, while merge() throws an exception. The size() method is likewise obvious, returning the `_size` member data.

Containment is determined by treating the TraceID argument as an integer, ranging from 0 to $N-1$, which is effectively required to have the clusters over a contiguous range, and checking if it falls in the interval `[_offset,_offset+_size)`. If it does, contains() returns true. It returns false otherwise.

The map() method adds the value of `_offset` to the argument and returns the result. Likewise the inverseMap() method subtracts the `_offset` from the argument and returns the result. Finally, the two project() methods are

```
EventPosition* Cluster::project(EventPosition* fm) {
  EventPosition* p = new EventPosition[_size];
  for (int i = 0 ; i < _size ; ++i)
    p[i] = fm[map(i)];
  return p;
}
```

and

```
EventPosition* Cluster::project(EventRef* e) {
  EventPosition* p = new EventPosition[_size];
  for (int i = 0 ; i < _size ; ++i)
    p[i] = e._event->CT.CT[e.cluster().inverseMap(map(i))];
  return p;
}
```

Observe that we have not added the requisite checking to any of these algorithms to ensure correctness of the arguments. This is because the manner in which our timestamp algorithm is written is such that we can be sure that they will never be called with a bad argument. If this class is to be widely available, however, such checking would be required. It should also be noted that these algorithms clearly satisfy the required performance and space-consumption bounds.

Finally we must provide the cluster() and cluster(int) methods. The cluster() method is an inline call to cluster(0). The cluster(int) method can be provided in a variety of ways. Each cluster could identify its parent, and the event would identify the lowest-level cluster. This would then cost $O(k)$, where k was the requested cluster level. This is not unreasonable, as k is likely fairly small. However, we can do better. Each event already identifies the block to which it belongs. Since cluster membership is fixed by trace, and each event block contains data for only a single trace, we record the cluster membership in the event trace, thus

```
class EventBlock {
...
  Cluster** _clusters;
};
```

Then the cluster(int) method becomes

```
class EventRef {
...
  Cluster& cluster(int k) const {
    return *_event->_block._clusters[k];
  };
};
```

Suitable bounds checking should be added if this method is made public.

9.2.2 Self-Organizing Clusters

We now turn to our dynamic algorithm. We start by outlining our design options given the constraints imposed. The membership requirement and the hierarchical nature of our clusters imply that we must use some variant on hierarchical agglomerative clustering. We therefore require that each trace initially belongs to a cluster of size one. We must then determine when to merge clusters and whether or not traces should move between clusters. Clearly we cannot pairwise compare all traces after the arrival of every new event to determine if the clustering has changed, nor is it desirable to do so, as no single event should have a significant impact on current clustering. Instead we deal with the computation cost of this approach by noting that we merely require good clusters, not optimal ones. It is thus sufficient to consider merging clusters based on some number of communications between the clusters, subject to a maximum cluster size.

The constraint on the number of communications examined is that we must make a merge decision at the earliest opportunity. The reason is that after an event is timestamped, either its timestamp will never change or we must pay a computation cost to change it. If timestamps are never changed, then some events will receive a longer timestamp than they should. Specifically, some events will be timestamped as cluster-receives even though in hindsight they are not, since the merger had not occurred at the time the event was timestamped (we refer to such events as false cluster-receives). In fairness, this will to some degree be balanced by the fact that those non-cluster-receive events that are timestamped prior to the merger will have somewhat shorter timestamps, reflecting the smaller pre-merger clusters. There is thus a space-penalty to pay, that grows as the merger decision is delayed. Conversely, if we alter the timestamps after a merger, the longer the merger is deferred, the larger the number of timestamp alterations that must be performed. Therefore, if clusters are to merge, the sooner the merger occurs the better.

The constraint on the maximum size of a cluster is required to prevent all traces from joining a single cluster. This upper bound must increase with each level of the cluster hierarchy or cluster-merging would never be able to take place beyond the lowest level. Further, it should increase by a least a factor of two, or no two lower-level clusters that had reached their maximum size would be able to merge. In general, it should be an integral multiple of the previous-level maximum cluster size for this same reason.

Second, we must deal with trace-movement between clusters. While our algorithms do not strictly presume that once a trace is part of a cluster it is forever part of that cluster, we have not yet evaluated alternatives to the contrary. In particular, it is unclear that we can satisfy both the time- and space-bounds should we move traces between clusters. Further, it substantially simplifies the merge() method when it is known that traces will never leave a cluster.

The third design option we must address is the issue of how to deal with timestamps given to events that were processed prior to a merger. Such timestamps are not valid in the merged cluster. There are two solutions that we have examined for this problem. We may adjust the timestamps for existing events, such that the interpretation given to them by the merged cluster is correct. This is largely a matter of iterating through the events on each trace of the merged cluster prior to the merge point and changing the entries as required. For most events it would presumably require allocating a larger array and filling in zeros in the appropriate elements, though for false cluster-receives it would presumably result in a space reduction. The advantage of this method is that it substantially simplifies the merge() and cluster() methods. However, it is not without disadvantages. Its cost is proportional to the size of the merged cluster times the number of events whose timestamp must be altered. If there are few prior events, this may be reasonable. However, it has a second drawback. Although we do not address the problem of trace-movement between clusters, we would like our system to remain open to that possibility. This scheme is clearly incompatible with that ideal.

The alternate solution is to keep track of the event number at which merging took place for each trace involved in the merger. We then maintain the information that interprets the timestamp both prior to and after the merger. This will be somewhat more space-consumptive, though that may be mitigated somewhat by the fact that smaller timestamps are used prior to the merger, as is the interpretation information for timestamps prior to the merger. This is the technique we have adopted for our algorithms. The implication of this is that if e^i is a transmit and e^j is the corresponding receive, and the result of this transmission is to cause their respective clusters to be merged, then e^j's cluster will contain e^i's trace but not *vice versa*. This is why the timestamp algorithm used TraceID containment testing rather than cluster equality.

Given these design options, we now provide our solutions. We start with a two-level solution, as it is notably simpler.

9.2.2.1 Two-Level Cluster-Class Algorithms

The clusters for our self-organizing algorithm must now keep track of the complete map and inverse-map data, rather than just an offset and size. Further, this data must be maintained in such a fashion as to enable the interpretation of timestamps both before and after merge operations. A naive approach would be to create a new cluster at each merge point to interpret the post-merge timestamps, leaving the old clusters to interpret the pre-merge timestamps. While easy to implement, this would not be ideal from a space-consumption perspective. We can do better. In particular, the post-merge cluster contains only two new pieces of information: the merge point and size data. None of the data required for interpretation changes in any way. We therefore divide the Cluster-class functionality over three entities: the event block, the Cluster-class front end, and a new Cluster-class back end. The event block maintains the merge-point information. The back end maintain the mapping and inverse-mapping data for a tree of merged clusters. The front end maintains the necessary indexing information into the back-end data to interpret it for the particular front-end cluster.

First, we will deal with the merge-point information. Unlike the fixed-cluster case, the cluster will not be the same for every event in a trace. When a cluster c_i merges with another cluster c_j to form a new cluster c_k, the cluster() method for the events prior to the merger must return references to c_i or c_j, as appropriate for their trace, while those after the merger must return a reference to c_k. We therefore augment the EventBlock with the following data

```
class EventBlock {
...
  class ClusterList {
    EventPosition _minEvent;
    Cluster*      _c;
  };
  ClusterList* _clusters;
  int          _currentCluster;
};
```

and define the cluster() method as follows.

```
Cluster& EventRef::cluster() const {
  int i = _event->_block._currentCluster;
  while (_event->_block._clusters[i]._minEvent > EventPosition())
    --i;
  return *_event->_block._clusters[i]._c;
}
```

```
class Cluster {
...
private:
  class BackEnd {
  public:
    int  _id;
    int* _map;
    int  _size;
    SparseArray<int> _inverseMap;
    SparseArray<int> _commCount;
    PartialOrder& _po;
  };

  int _size;
  int _offset;
  BackEnd* _be;
};
```

Figure 9.9: Self-Organizing Two-Level Cluster-Class Private Data

The number of clusters in the cluster list is, in the worst case, O(cluster size). On average it will be O(log cluster size) if all cluster mergers were between equal-size clusters. However, both of these values ignore that fact that no event block need maintain information for clusters that contain no events from the block. If we presume that most cluster mergers occur within the first event block, then the cluster list will likely be quite short in subsequent blocks.

Note also that we work backwards from the current cluster of which the trace is a member. This is because the cluster() method is used in two places, timestamping and precedence determination. In the precedence-determination case we cannot readily guess the location of the events whose clusters must be determined, and thus their likely clusters, though we would suspect that more events occur later, and are thus closer to the current cluster. In the timestamp case, however, we know that we are timestamping in a linearization of the partial order, and thus clusters that we look up are for events closer to or at the current cluster for the trace. This linear search is therefore probably quite fast, though in the worst case it could be O(cluster size). If this was found to happen frequently, and the cluster sizes were large, we would move to some form of binary search.

We now deal with the Cluster class, whose private data is as shown in Figure 9.9. For each merge operation, a new front end is created, while the back end merges in the new cluster information. The methods are provided at the front end, and augmented with front-end data, provide the correct interpretation of the back-end data. The `_map` and `_inverseMap` arrays of the BackEnd class provide the interpretation of the timestamp data. The `_map` array maps cluster-timestamp indices to TraceIDs. Conversely, the `_inverseMap` array maps TraceIDs to cluster-timestamp indices. To understand the operation of these arrays, we provide the merge() method, as shown in Figure 9.10. This method does four things. It creates a new front-end cluster (lines 4 to 7), it merges the back-end of the clusters (lines 8 to 14), it updates the information maintained in the event block that is required for the cluster() method (lines 15 to 19 and line 26), and it updates the offset data for clusters whose BackEnd `_map` variable has been relocated (lines 20 to 24). The creation of the new front end is straightforward, and so we do not comment further.

For the back-end merger we arbitrarily select the back end pointed to by **this** front end as the object into which we will merge the data. We could slightly more intelligently select the larger back end, where larger would be defined either by number of traces kept within the cluster or by the number of front-end clusters that pointed to that back end. Either approach would be somewhat better than arbitrary selection, but is of little significance to our algorithm description. The map data for the other back end is appended to the map data for **this** back end (lines 9 and 10). Note that this means that any front-end cluster that pointed to that other back end will now have incorrect offset information, as the offset has been shifted by the size of **this** cluster. It is the function of line 22 to correct this. The inverseMap data is likewise incorporated (line 11), the size of the new back end adjusted (line 13) and the other back end, no longer needed, is deleted (line 14). The deletion of this other back end means that any front-end cluster that pointed to that other back end will now have a dangling pointer. It is the function of line 23 to correct this.

Lines 15 to 19 and line 26 update the front-end cluster information maintained in the event block. This is primarily a question of iterating through the traces in the cluster and setting their current cluster value to indicate the new front end for future events (hence the use of one greater event position than the current maximum event on the trace). This is true for all but the event that triggered the merger, which must have its position identified as the beginning of the new front end, which is the purpose of line 26.

As already noted, both the offset and BackEnd pointer of front end clusters that referred to the deleted back end must be corrected. The traces for which this is an issue are those that were maintained by that back end, and are thus those at location `_size` and beyond in the merged entity. Note that this code is not strictly correct. This will only update clusters that are represented within this event event block. Clusters in prior event blocks will be missed. There are two solutions to this issue. Either we may create a new back end for the first merger in any new event block, or we must update back to the first block for each trace. The second solution is likely more expensive than we wish to incur, and so we presume the first solution.

Having described the merger process and the manner in which the back-end data are maintained, we can readily provide the remaining methods of the Cluster class. The map() method is simply

```
inline TraceID& map(int index) const {
  return _be->_map[index+_offset];
}
```

The inverse-mapping data, on the other hand, is somewhat more complex. It implements the mapping of TraceID to cluster-timestamp index, and is thus sparse over its domain. A space-consumption-naive implementation of this would be an array of size equal to the number of

```
Cluster& Cluster::merge(EventRef& e) {
  assert(_size == _be->_size);
  Cluster& c = e.partner().cluster();
  Cluster* nc = new Cluster();
  nc->_be = _be;
  nc->_offset = 0;
  nc->_size = _size + c._be->size;
  realloc(_be->_map, nc->_size);
  for (int i = 0 ; i < c._be->_size ; ++i ) {
    _be->_map[_size+i] = c._be->_map[i];
    _be->_inverseMap[c._be->_map[i]] = _size + i;
  }
  _be->_size = nc->_size;
  delete c._be;
  for (int i = 0 ; i < nc->_size ; ++i ) {
    EventRef f = e._block._po.maxEvent(_bc->_map[i]);
    ++f._block->_currentCluster;
    f._block->clusters[f._block->_currentCluster]._minEvent
      = f.eventPosition() + 1;
    f._block->clusters[f._block->_currentCluster]._c = this;
    if (i >= _size)
      for (int j = 0 ;  j < f._block->_currentCluster - 1; ++j) {
        f._block.clusters[j]._c._offset += _size;
        f._block.clusters[j]._c._be = _be;
      }
  }
  e._block->clusters[e._block->_currentCluster]._minEvent
      = e.eventPosition();
  return *nc;
}
```

Figure 9.10: Self-Organizing Two-Level Cluster Merging

traces, with most elements indicating no data present. This is not, however, as bad an idea as it may first appear. Specifically, it is very fast. We index the inverse map array by TraceID and find the relevant index in the cluster timestamp. Further, we will never have more back-end cluster objects than we have traces, and on each merge operation, the number of such back-end clusters is reduced by one. However, we can do better. At a minimum we can use any reasonable vector class that only allocates space over the domain of input indices. Thus, any cluster with only one trace will require an array of size one. Not only are such clusters present in our system, they are initially the only form of cluster. This thus eliminates at least half of the space problem. We can still do better. There is no reason why an inverseMap array representing two TraceIDs needs to use an array over the entire range of those TraceIDs, especially if that range is large. A two-step lookup process is still cheap. This eliminates a further quarter of the space requirement. A three-step process is likewise probably still cheap, though becoming less attractive. At some point we move from a size-of-cluster sparse array to a range-of-indices array, which is a tradeoff between the size of the range and the desire to have fast lookup. There are, of course, many other alternatives that an implementer can examine, including those in the various literature on sparse arrays and hashing. For this reason, we simply specify that the `_inverseMap` member-data is a sparse array of type int. The inverseMap() method is then

```
inline int inverseMap(TraceID& t) const {
  return _be->_inverseMap[t] - _offset;
};
```

Note that neither the map() nor inverseMap() methods checks their argument to determine if it is valid. A brief examination of the timestamp code should convince the reader that neither function will be invoked with out-of-domain arguments. That said, the containment test does effectively require the inverseMap() method to accept out-of-domain parameters. It is a test as to whether or not a given TraceID is represented in the cluster, and thus amounts to a lookup in the inverse map. We therefore require that the SparseArray class return an indicator when there is no data present. Since the data in the sparse array will be array indices, -1 is a suitable choice of indicator. This leads to the following containment test.

```
inline bool contains(TraceID& t) const {
  return (_be->_inverseMap[t] >= _offset &&
          _be->_inverseMap[t] < _offset + _size);
};
```

This leaves the size, uc, mergeable, and project methods. The size and uc methods are unchanged from the fixed-cluster case, as is Fidge/Mattern projection. The merge(k,e) and project(e) methods are only used in the hierarchical-timestamp algorithm, and are thus not required at this point. This leaves the mergeable() method.

The merger decision is taken by the mergeable() method. As we have indicated above, this must be made as quickly as possible, and as efficiently as possible. For the timestamp algorithm to be reasonable, this method must be roughly constant-time. The following code satisfies these requirements, and is consistent with the design options of merging based on some number of communication events.

```
bool Cluster::mergeable(EventRef& e) {
  Cluster& c = e.partner().cluster();
  if (_be->_size + c._be->_size > sizeThreshold)
    return false;
  if ((_be->_commCount[c._be->_id] + c._be->_commCount[_be->_id])
       /(_be->_size+c._size) < commThreshold)
    return false;
  return true;
}
```

The `_commCount` data is a sparse array whose elements are incremented on communication events between the clusters. This can be implemented by means of a callback on receive- and synchronous-event store() operations on the partial order. The purpose of dividing the `_commCount` sum by the size of the clusters is to filter out the effect of cluster size and capture instead the average number of communications events per trace. That said, we believe that a substantially greater study of clustering algorithms would be of value. For our experimental results, we have simply used initial communication as a sufficient requirement to merge clusters. This corresponds to a `commThreshold` value of zero, as mergeable is only ever invoked when communication occurs. It is also for this reason that we have not yet dealt with the issue of merging the communication-count information when clusters merge, as is clearly required.

9.2.2.2 Hierarchical Cluster-Class Algorithms

For the hierarchical Cluster-class algorithms we must deal with two additional problems that pertain to cluster merging. First, one or both of the the clusters to be merged might be the universal cluster. Second, when clusters merge, there are implications for the parent clusters (that is, the clusters at the next level in the hierarchy).

Before dealing with these issues we briefly observe that the remaining methods of the Cluster class are essentially unaltered from the two-level self-organizing algorithms described above. The cluster() method, as described in the timestamp algorithms, must now take an integer parameter indicating the level in the cluster hierarchy to which it refers. The algorithm for this is therefore formed by applying the hierarchical fixed-cluster approach to the two-level self-organizing cluster() method. The result is the following change to the EventBlock

```
class EventBlock {
...
  class ClusterList {
    EventPosition _minEvent;
    Cluster*      _c;
  };
  ClusterList**   _clusters;
  int*            _currentCluster;
  int             _maxLevel;
  static Cluster* _uc;
};
```

allowing us to define the cluster() method as follows

```
Cluster& EventRef::cluster(int k) const {
  if (k > _maxLevel)
    return *_uc;
  int i = _event->_block._currentCluster[k];
  while (_event->_block._clusters[k][i]._minEvent > EventPosition())
    --i;
  return *_event->_block._clusters[k][i]._c;
}
```

The universal cluster is kept as a static pointer in member data `_uc`.

We now address the problem of cluster merging. We deal with the second issue, that of parental involvement in merging, first. We first observe that the merging of two parent clusters is only of minimal relevance to their child clusters. Specifically, the event block for child clusters will need to be updated to reflect the new parent. Other than this, there is no change. However, the converse is not the case. When child clusters merge, their parent clusters must also merge, recursively.[2] This is required to ensure that no cluster has more than one parent. In the event that trace-movement between clusters is developed, and likewise cluster-movement between higher-level clusters, this requirement would presumably be altered.

This leads to the mergeable() test, which augments the two-level test as follows.

```
bool Cluster::mergeable(int k, EventRef& e) {
  if (!mergeable(k+1, e))
    return false;
  return twoLevelMergeable(e);
}
```

where `twoLevelMergeable()` is the previously defined algorithm for mergability testing in two-level clusters. Merging is likewise altered to become a recursive algorithm. However, merging must also deal with our first problem, namely that one or both of the clusters might be the universal cluster. If both clusters are the universal cluster, then a new cluster (that is, a new back end as well as a new front end) must be created, such that the child clusters refer to that new cluster as their parent, rather than the universal cluster. This is essentially a question of taking the two clusters and applying a modification of the merge algorithm from Figure 9.10.

[2]The requirement for parental approval has some precedent in the non-technical literature [133].

```
 1: Cluster& Cluster::newCluster(EventRef& e) {
 2:   Cluster& c = e.partner().cluster();
 3:   Cluster* nc = new Cluster();
 4:   nc->_be = new Cluster::BackEnd;
 5:   nc->_offset = 0;
 6:   nc->_size = _size + c._be->size;
 7:   nc->_be->_map = malloc(nc->_size);
 8:   for (int i = 0 ; i < _be->_size ; ++i ) {
 9:     nc->_be->_map[i] = _be->_map[i];
10:     nc->_be->_inverseMap[_be->_map[i]] = i;
11:   }
12:   for (int i = 0 ; i < c._be->_size ; ++i ) {
13:     nc->_be->_map[_size+i] = c._be->_map[i];
14:     nc->_be->_inverseMap[c._be->_map[i]] = _size + i;
15:   }
16:   nc->_be->_size = nc->_size;
17:   return *nc;
18:   // Adjust cluster pointers
19: }
```

This code shows only the creation of the new back and front end. The adjustment of cluster references in the event blocks (line 18) requires all event blocks for the traces represented in the cluster to add a new ClusterList element. This is a small matter of applying lines 15 to 19 and line 26 from Figure 9.10, adjusting to reflect the somewhat different structure of the ClusterList.

When only one of the two clusters to be merged is the universal cluster we do not need to create a new back end. Rather, for the event whose level-k cluster is the universal cluster, we use the level-$(k-1)$ cluster. We merge that level-$(k-1)$ back end into the other cluster, per the algorithm of Figure 9.10, though we do not delete the level-$(k-1)$ back end, and the cluster pointer update is per the universal-cluster merging method.

If neither cluster is the universal cluster, then we merge the two, per the two-level algorithm, and then we merge their parent clusters, up to the point of merging two universal clusters. We do not merge the two universal clusters, as the creation of a new cluster at that point is redundant. If the parent clusters are the same, then no such merger need occur.

We note at this juncture that this merging process is potentially quite expensive. However, it is also quite unlikely to occur over more than one or two levels in the hierarchy. Level-k clusters are only created in the timestamp algorithm because the level-$(k-1)$ clusters could not accommodate the merger. This is almost invariably because of the size limitation of the clusters. As such, we suspect it would be unusual for a cluster far from the top of the hierarchy to be involved in a merger. We do not yet have data on this point, as we have not yet implemented this approach. Likewise, we note that the space consumption has risen substantially over the two-level approach, as each level maintains its own copy of the cluster information. We would like to address this problem before we implement this algorithm.

9.3 Caching Strategy

When we first developed this algorithm, and the dynamic-Ore timestamp, we did so with the idea that a sufficient space-consumption reduction would enable an entirely in-core partial-order data structure. In such a case the precedence tests would assuredly be much faster than is currently the case. While our results will show that we have achieved up to an order-of-magnitude space reduction (Section 9.4), it would be hard to argue that this is enough. Although it enables an in-core data structure for partial orders that is noticeably larger than can currently be handled, we could not credibly claim that such an approach was scalable. Further, as soon as the data-structure size exceeded the core-memory size, performance would degrade dramatically. A caching strategy is required to claim that the structure is scalable.

We could simply adopt the POET caching strategy. That is, we would have multiple caches, each recording some timestamper state and some number of recently-computed timestamps. However, this is not the best approach possible. In particular, it fails to take advantage of the variation in cost required to compute timestamps at different levels in the cluster hierarchy when we have eliminated Fidge/Mattern timestamp generation (Section 9.1.3). More simply put, the lower the level of the cluster-timestamp, the cheaper it is to compute.

If we consider the two-level algorithm, any cluster-receive timestamp will cost $O(cN)$ while the remaining timestamps only cost $O(c)$, where N is the number of traces and c is the cluster size. Thus, the natural caching strategy is to keep only cluster-receive timestamps. If possible, we keep them all, and cache the remaining timestamps on an as-space-available basis. We can guarantee this possibility by increasing the size of the clusters until the number of cluster-receives is sufficiently small to fit in cache (this is based on the observation that the number of cluster-receive events decreases as cluster-size increases). We are, in effect, bounding the space-consumption of our timestamp. We note at this point that this approach does not lead to the minimum-possible space-consumption. Rather, it bounds the core-memory consumption. We call this mode of operation “caching mode” and distinguish it from “core mode,” where we would attempt to achieve the minimum space-consumption possible with the view that the entire data structure would reside in core memory.

The obvious extension of this approach to the hierarchical algorithm is to cache cluster-receive events from the highest level, working our way down, subject to available space. In this case it is doubtful that we could cache all cluster-receives at every level. However, the lower the level of cluster-receive, the cheaper it is to compute. A level-k cluster-receive costs $O(c_k c_{k-1})$ to compute, where c_i is the size of the cluster at level-i. At low levels, cluster size is small, and hence this cost is low.

Finally, we note the cost of precedence-tests in a caching situation. Recall (Section 7.1.2) that the cost of such tests for Fidge/Mattern timestamps is $O(N)$, as timestamps have to be computed dynamically. If we consider the two-level version of our described caching-mode, an in-cluster precedence test will require the computation of the appropriate cluster timestamps, and will thus be $O(c)$. A between-cluster test will require the calculation first of the relevant cluster-timestamp, and then examination of each greatest-preceding cluster-receive, and will thus also be $O(c)$, with the constant larger by one, since there would be one additional iteration over the cluster size.

9.4 Experimental Evaluation

We now turn to the evaluation of our timestamp. While we have looked at, and will present results for, the same environments and computations that we studied for our dimension-bound programs (see Section 8.1.3), we have focused this evaluation on PVM and Java computations. This focus was necessary to allow for more in-depth analysis and these environments were found to be reasonably representative. All of the following experiments will concern timestamp size, rather than precedence-test computation time. This choice is dictated by our study of the scalability problems of the Fidge/Mattern timestamp (Chapter 7), together with an inability to determine a representative sample of precedence-test operations (see Section 7.1.1 of that chapter).

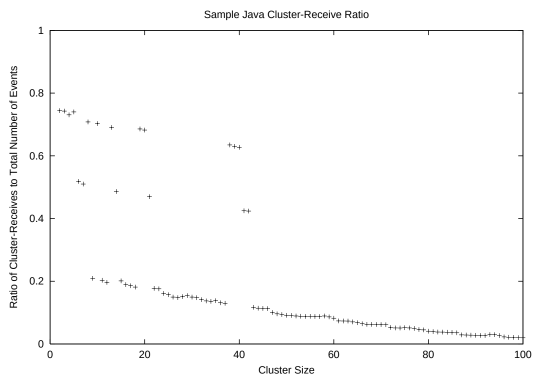
Our analysis is broken down according to the clustering algorithm used. We first evaluated fixed clusters, with no special pre-processing technique to improve the cluster selection. This represents a practical worst-case scenario for our timestamp. We then looked at self-organizing clusters. Here we chose the worst-case clustering algorithm, which in this instance is to merge clusters on first communication. Thus, the results we present should be viewed as the least time-stamp size-reduction one can expect with the application of our approach. For both clustering techniques we studied both caching- and core-mode operation.

9.4.1 Fixed-Cluster Algorithm

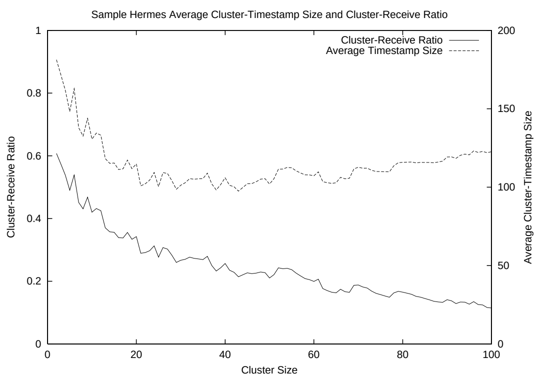
For all of our fixed-cluster experiments the timestamp-tool generated contiguous clusters of equal size at each level in the hierarchy. Thus, for a cluster-size c, the first c traces received by the tool were placed in the first cluster, the next c in the second cluster, and so forth. We start with an analysis of the two-level algorithm with the intent of core-mode use. That is, we wished to identify that cluster-size c which achieves the minimum-average cluster-timestamp size. The form of the experiments was to vary the cluster size from 2 to 50 and observe the changes in the ratio of cluster-receive events to total-event count, the average timestamp size, and the percentage of the Fidge/Mattern timestamp size.

Not surprisingly, the cluster-receive ratio was high when the cluster size was small. It tended to be around one-third for a cluster size of 1, though in the case of Hermes it was as high as two-thirds. It dropped off fairly rapidly, going below 20% by the time the cluster size was between 10 and 20. This is consistent with the average number of traces a given trace communicates with, as shown in Figure 9.1. Of greater interest was the degree of variability, especially in the Java computations. An example of this phenomenon is shown in Figure 9.11(a). What appears to be going on is that the arbitrary clusters that are selected by the timestamp tool are not always well-chosen. This is not surprising, and it strongly suggests that computation-specific clustering would be capable of generating significantly better, or at least consistent, results. In the PVM computations we found a much smoother decline in the cluster ratio, which is probably a function of regular communication patterns exhibited in the PVM programs we studied.

The average timestamp size tended to follow the cluster-ratio reduction for low cluster sizes, but grew as the cluster-size grew. This is unsurprising, as it reflects a substantial benefit in reducing the timestamp size for non-cluster-receive events when the cluster-size is very small. When



(a) Sample Java Cluster-Receive Ratio



(b) Sample Hermes Average Cluster Timestamp Size

Figure 9.11: Sample Cluster-Receive and Timestamp-Size Results

the cluster-size is larger, the benefit of cluster-timestamps is attenuated by the size of the cluster itself. Figure 9.11(b) shows an example of this phenomenon for a Hermes application. In this instance, the optimum benefit of the cluster timestamp is a one-third reduction in space-consumption over a Fidge/Mattern timestamp.

We then observed the percent reduction in timestamp size against Fidge/Mattern timestamps. To compare the two-level algorithm results with Fidge/Mattern timestamps we have to consider the manner in which the Fidge/Mattern timestamps were encoded. There are three different plausible ways in which Fidge/Mattern timestamps may be encoded in a distributed-system observation tool. First, we may use a fixed-size array. For this we used 300, since this is the value that POET uses. Second, it may use a timestamp of size equal to the number of traces currently known, plus one to indicate the current array size. This is optimal from a space-consumption perspective, though possibly poor in practice because of the “Swiss-cheese” effect on heap-memory allocation. Third, we presume that we know neither the number of traces nor an upper-bound on that number. We thus use a tabling technique and double the array size whenever it is insufficient to contain all traces. This technique depends on the starting size, which we arbitrarily selected as 2, and thus we rounded up to the nearest power of 2. In addition to these results, we aggregated the results by determining the minimum ratio for each method, and then determining its minimum, average, and maximum value over the different computations for each environment. We also determined the cluster-size range over which these minimums occur. These results are summarized in Table 9.1, while Figure 9.12 shows samples for the four different environments.

Note that in the best case PVM shows more than 90% size reduction. The worst result we have seen so far for the PVM environment is a two-thirds size reduction over Fidge/Mattern. On average the PVM computations achieved 80% size reduction, while the Java, DCE, and Hermes programs only achieved between one- and two-thirds reduction. The choice of Fidge/Mattern encoding technique had the most significant effect in the DCE computations, which is doubtless

Method	Minimum (%)	Average (%)	Maximum (%)	Range
Fixed-Size	7.6	17.0	33.0	10–64
Optimal	11.1	22.7	34.2	2–8
Tabling	11.1	22.6	34.2	2–8

(a) PVM

Method	Minimum (%)	Average (%)	Maximum (%)	Range
Fixed-Size	16.6	28.8	37.6	6–53
Optimal	16.6	39.1	51.8	6–24
Tabling	14.6	34.7	49.8	6–24

(b) Java

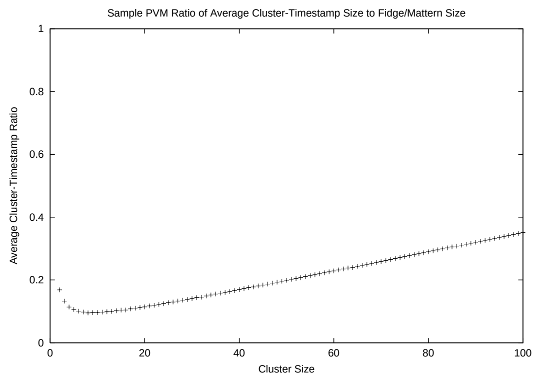
Method	Minimum (%)	Average (%)	Maximum (%)	Range
Fixed-Size	23.3	38.2	57.7	68–100
Optimal	62.6	64.8	67.2	11–38
Tabling	53.0	60.8	65.9	22–78

(c) DCE

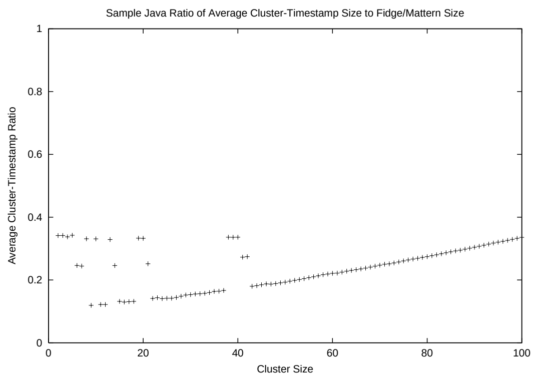
Method	Minimum (%)	Average (%)	Maximum (%)	Range
Fixed-Size	32.7	34.5	35.4	43–44
Optimal	32.8	39.7	42.9	22–44
Tabling	26.9	36.5	42.7	22–67

(d) Hermes

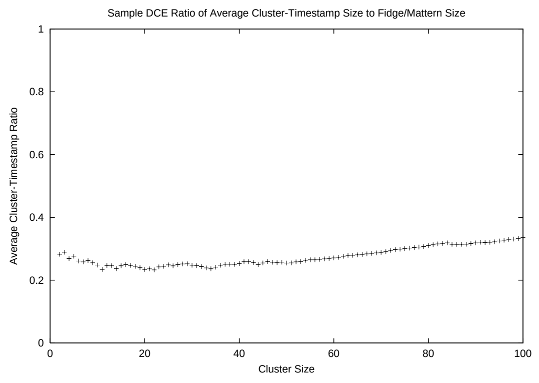
Table 9.1: Cluster- to Fidge/Mattern-Timestamp-Size Ratio Summary Results



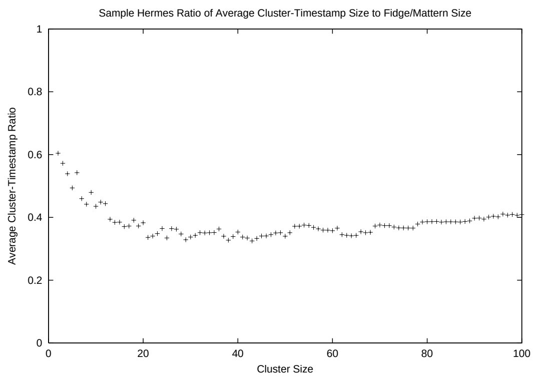
(a) PVM



(b) Java



(c) DCE



(d) Hermes

Figure 9.12: Sample Ratios of Cluster- to Fidge/Mattern- Timestamp Sizes

Environment	Range (20%)	Range (10%)	Range (5%)
PVM	4–9	6 [a]	-
DCE	4–41	10–12	20
		17–25	
		33–34	
Hermes	21–23	22	22
	25–27		
	29–33		
	36–45		

[a]Only 93% of the computations examined were within 10% at cluster-size 6.

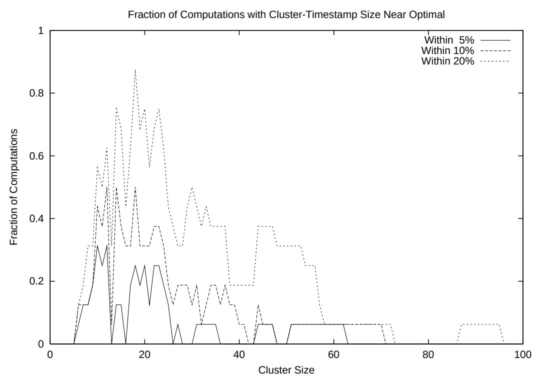
Table 9.2: Cluster-Size Required for Timestamps within 5%, 10%, and, 20% of Optimum

because all but one of those had less than 120 traces. This thus skewed the results in the fixed-size encoding artificially in favour of our cluster-timestamps. This also explains why the large cluster-size was optimal for this case. For the remainder of this work we presumed that the optimal Fidge/Mattern encoding was used, as this is least-favourable to the cluster-timestamp.

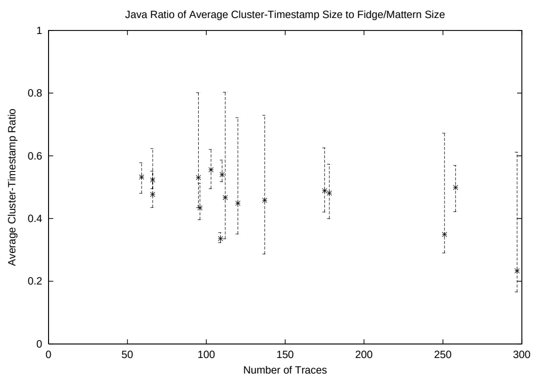
All of these size reductions presume that the optimal cluster-size is selected for the computation. Such a selection, in advance, is in general impossible. Indeed, over all of the computations we have studied the optimum cluster-size varied from 1 to 100. Even though it tended to be more constrained for the specific programming environments, the range was still large and, especially in the case of Java, the cluster-timestamp size would vary substantially over that range. We therefore analysed the range for each environment to determine what set of cluster-sizes resulted in a cluster-timestamp size within 5%, 10%, and 20% of the optimum. The intent was to determine if there was an intersection of these ranges over the set of computations which could be used to achieve “good” size-reduction. For three of the environments, PVM, Hermes, and DCE, we were able to determine cluster-size ranges that satisfied this at the 20%-level for all of the computations examined. For both Hermes and DCE we were able to identify a cluster-size that was within 5% of optimal for all computations. We show these ranges in Table 9.2.

Java was somewhat more problematic, which is not surprising given the variability in cluster-receive ratio that it exhibited. The best that could be achieved was slightly more than 85% of the computations within 20% of optimal. This problem can be easily seen in Figure 9.13(a), which shows the fraction of computations that have a timestamp-size near to the optimal for the given cluster size. Based on this data, a range of between 17 to 23 is the best that can be achieved. Using this range we examined the minimum, maximum, and average timestamp-size for the various Java computations. These results are presented in Figure 9.13(b).

Clearly the size-reduction desired has not been achieved for three of the four environments, and the PVM size-reduction is less than might be hoped for. In fairness to the PVM results, most of the computations have 128, or fewer, traces, and thus an order-of-magnitude reduction



(a) Fraction with Good Timestamp Size



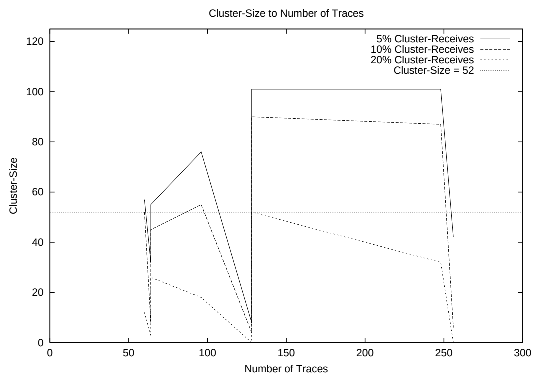
(b) Timestamp Size for Cluster-Size 17–23

Figure 9.13: Problems with Java Cluster-Size Selection

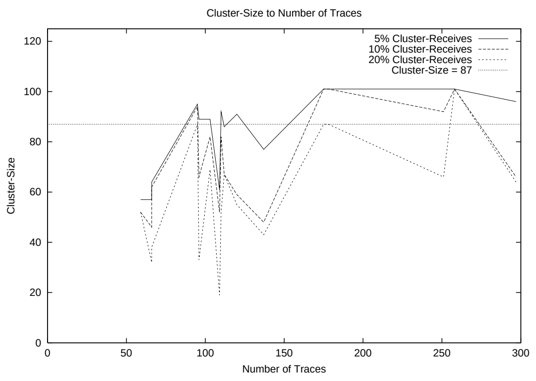
was plausibly more than could be expected. Further, the traffic-pattern analysis suggests that the ratio of cluster-timestamp to Fidge/Mattern-timestamp size will improve as the number of traces increases. In Java, while some reasonable size-reduction was exhibited, it was not consistent with a given range of cluster-size. In neither Hermes nor DCE was reasonable size-reduction exhibited. In all three of these environments we believe, based on traffic analysis, that the result is because of arbitrary, and thus poor, cluster selection.

We now evaluate the use of caching-mode. As we have already observed, no size-reduction amount will be sufficient for all computations. For caching-mode we presume that all cluster-receive events will be kept in core memory. We therefore wish to determine at what point the number of cluster-receives is less than some space-reduction threshold. We have arbitrarily selected 20%, 10%, and 5% as a reasonable fraction of the total number of events. This would correspond to 20%, 10%, and 5%, respectively, of the required Fidge/Mattern space-consumption. Figure 9.14 shows the results of this in terms of absolute cluster-size required, while Figure 9.15 provides the information as a ratio of the required cluster-size to the number of traces in the computation.

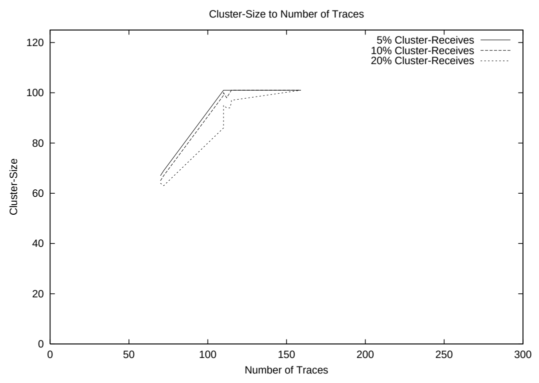
There are several points to be observed about these figures. First, note that the cutoff of 100 is false, as this is simply the maximum cluster-size that we examined. Thus, the apparent decline in the DCE ratio of Figure 9.15(c) can be seen to be false based on the cutoff of 100 shown in Figure 9.14(c). Further, it is apparent that the 5%-cluster-receive ratio is unrealistic for most of these computations. That said, there is a cluster-size bound that will achieve a 20%-cluster-receive ratio for PVM. It is shown as the horizontal line at cluster-size 52 in Figure 9.15(a). Likewise, for Java, at cluster-size 87 all but one computation achieve a 20%-cluster-receive ratio. It also appears that such a bound exists at the 10% and 5% levels, though this is near the edge of the data range, and thus we are hesitant to assert the bound. Given these bounds we can expect to compute the non-cluster-receive timestamps significantly faster than when using Fidge/Mattern timestamps.



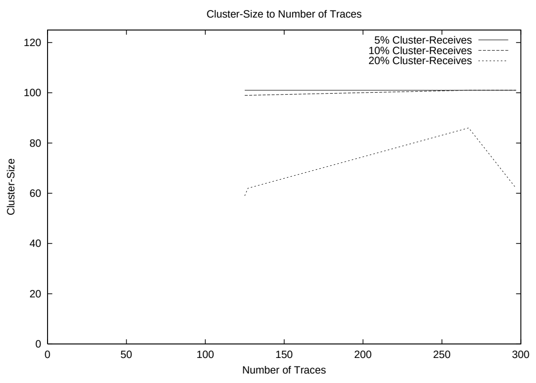
(a) PVM



(b) Java

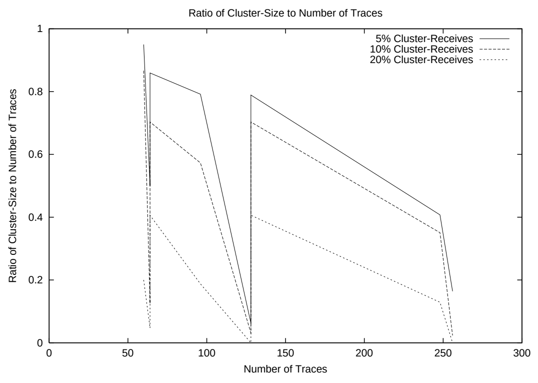


(c) DCE

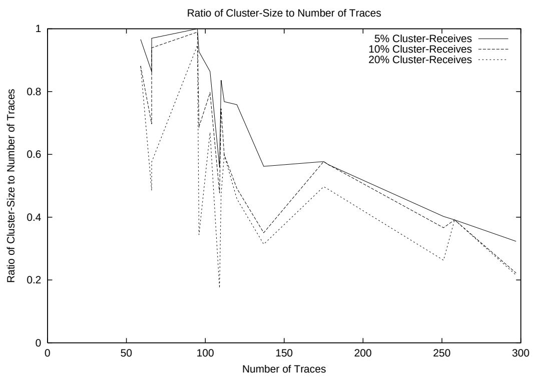


(d) Hermes

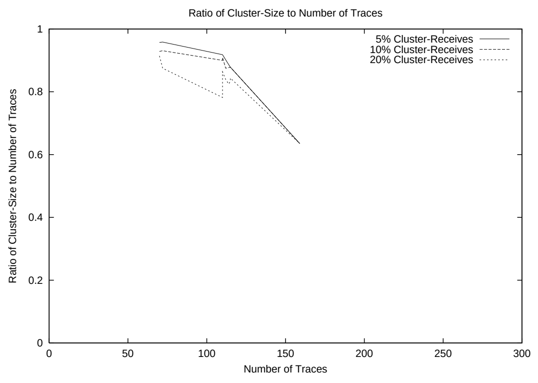
Figure 9.14: Cluster-Size v. Number of Traces



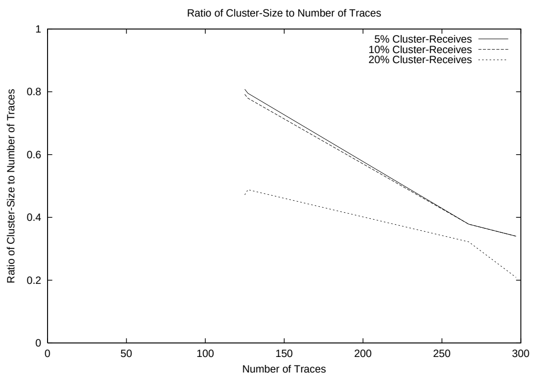
(a) PVM



(b) Java



(c) DCE



(d) Hermes

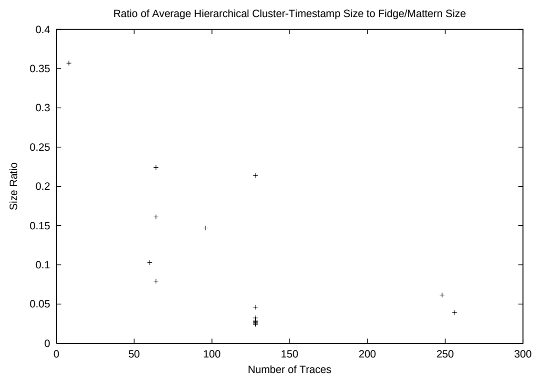
Figure 9.15: Ratios of Cluster-Size to Number of Traces

The final aspect of the fixed-cluster approach that we wish to study is the hierarchical algorithm. For this algorithm we must select some cluster-size for every level in the hierarchy. It is reasonable to presume that the system will select some factor by which to increase the cluster size at each level of the hierarchy. For this we considered two possibilities. For both of them we considered a system in which the level-1 cluster was size 2. For the first approach we presumed that the cluster-size doubled at each level in the hierarchy. This technique can be expected to yield the smallest average-size timestamps possible with fixed-clusters. However, it is not entirely realistic, as it makes event-precedence determination quite costly when the cluster that encompasses both events occurs at a high level in the hierarchy. For the second approach we presumed that the cluster-size was squared at each level of the hierarchy. We expected that this method would yield somewhat poorer average timestamp sizes than the first approach. However, it represents a realistic tradeoff between the cluster-timestamp size and time to compute event precedence.

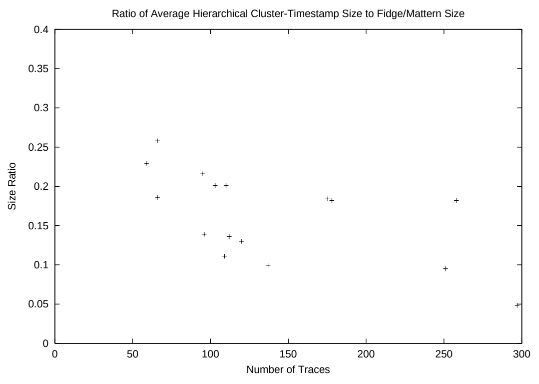
The results of these approaches are shown in Figure 9.16 for cluster-size doubling and Figure 9.17 for cluster-size squaring. These results encompass over 30 PVM computations with the number of traces varying from 50 to 250, and over 15 Java computations, with a similar range of traces represented. These figures illustrate the ratio of average cluster-timestamp size to optimal Fidge/Mattern timestamp size. As anticipated, the doubling technique achieved better results than did squaring, requiring as little as 2.5% of an optimal Fidge/Mattern encoding for PVM and 5% for Java. Most Java results were less than 20% of Fidge/Mattern size, while the PVM results tended to require less than 10%. That said, as noted above, these size-reductions come at the cost of more-expensive precedence tests.

For the hierarchical squaring, the PVM results were almost exclusively below 15%, while the Java results tended to be in the 20% to 35%-range. We believe the problem in the Java case is that the sensitivity to cluster-size is being exhibited. Specifically, the cluster-size 16 was not one of the preferred choices for Java in the two-level algorithm.

We now examine the number and ratio of cluster-timestamps at each level in the hierarchy. These are shown in Figure 9.18 for PVM and Java. We have used lines rather than points for these figures to make the data clear. That said, there are five discrete points per figure, rather than a continuous function. While the curves clearly differ, the essential shape is the same, with the largest quantity of cluster-timestamps being at level-0. In the case of PVM, about 75% of the events are in level-0, on average, while for Java the corresponding number was 30%. None of these results are surprising, though they do require some explanation. First, every transmit and every unary event will be timestamped in level-0. In the case of PVM, typically over one-third of the events are unary, and, as there are no multicast or synchronous events, the remainder are equally split between transmit and receive. This immediately explains about a 70% ratio, even if all receives at level-0 were cluster-receives. However, that is not the case. As is typical in parallel computation, there is substantial amount of nearest-neighbour communication. Since the level-0 cluster is size two, half of these nearest-neighbours will be within the level-0 cluster. The remaining half will be in level-1, which is why it contains the second largest number of events, on average. Java computations have far fewer events in level-0, as most of their communication is synchronous, and thus (potentially) cluster-receives, and there are, typically, far fewer unary events recorded in the

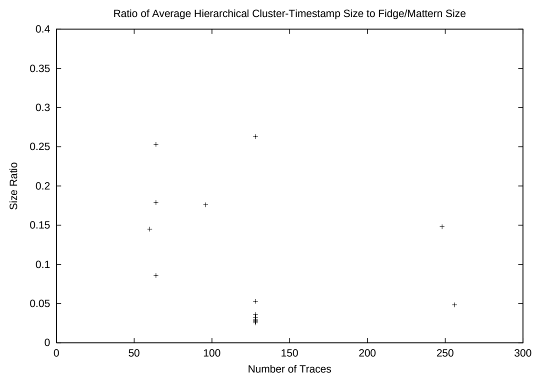


(a) PVM

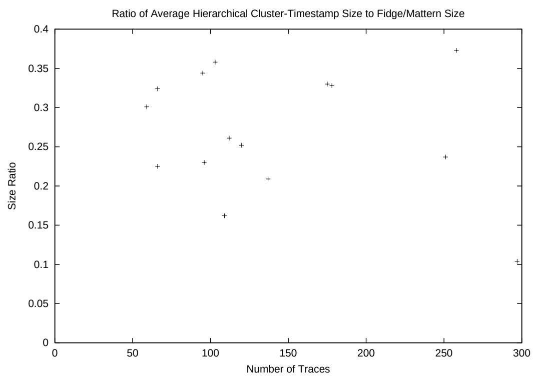


(b) Java

Figure 9.16: Results for Doubling Hierarchical Timestamp

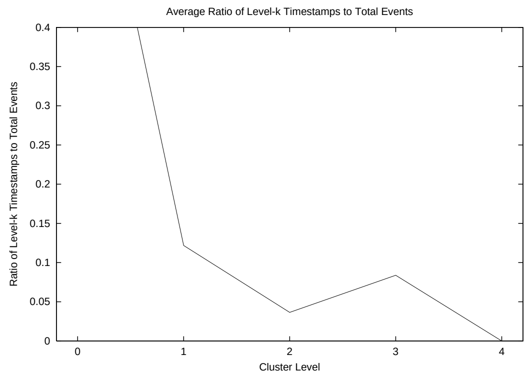


(a) PVM

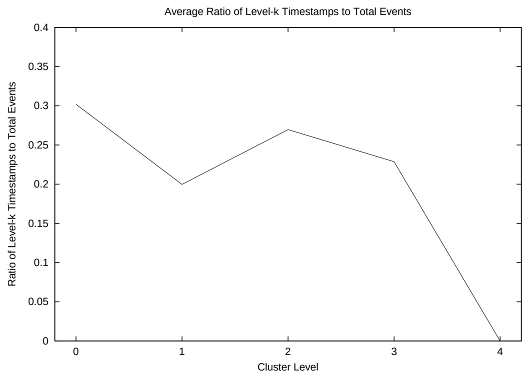


(b) Java

Figure 9.17: Results for Squaring Hierarchical Timestamp



(a) PVM



(b) Java

Figure 9.18: Average Ratio of Level-k Timestamps to Total Events

Java environment. The modest peak at cluster-level-2 (Java) or 3 (PVM) appears to be capturing much of the remaining communication. The number of level-4 timestamps is negligible, though that is in part because we are reaching the limits of our available data. These results suggest that a caching-mode is feasible for the hierarchical algorithm, though the fixed-cluster algorithm is a poor choice, and must be remedied.

9.4.2 Self-Organizing Algorithm

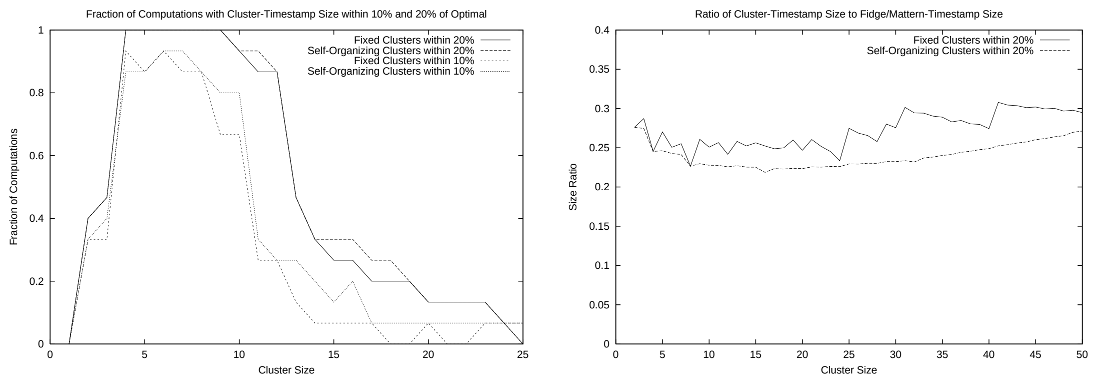
We now move to self-organizing clusters. As we have already indicated, we will only examine the two-level algorithm, with particular focus on the Java environment. The cluster heuristic we use for these results is to merge two clusters on first communication, subject to the upper-bound on cluster size. This is probably a worst-case dynamic-clustering algorithm, and so insofar as it provides good results, better clustering methods will only improve the results. We also chose this technique as it was developed first, since it substantially simplified the cluster-merging algorithms. As with the two-level fixed-cluster algorithm, we look first at core-mode and then caching-mode.

For core-mode analysis we are primarily interested in identifying that maximum cluster-size, or range of sizes, which minimizes space-consumption. As with the fixed-cluster examination, the form of our experiments was to vary the maximum cluster-size from 2 to 100 and observe the changes in the ratio of cluster-receive events to total-event count, the average timestamp size, and the percentage of the Fidge/Mattern timestamp size. In addition, we compared the space-requirements and ideal maximum-cluster-size ranges with those of the two-level fixed-cluster algorithm. For all of these experiments we presumed that the observation tool uses optimal encoding for the Fidge/Mattern timestamp.

The summary of our results can be seen in Table 9.3. As can be seen by comparison with

Environment	Minimum (%)	Average (%)	Maximum (%)	Range
PVM	11.0	22.0	34.3	2–16
Java	13.8	33.3	48.6	5–9
DCE	41.4	44.7	48.8	20–30
Hermes	26.6	34.6	37.9	21–22

Table 9.3: Self-Organizing Cluster- to Fidge/Mattern-Timestamp-Size Ratio Summary Results



(a) Fraction with Good Timestamp Size (b) Timestamp-Size Ratio Improvement

Figure 9.19: PVM Ideal Maximum-Cluster-Size Analysis (Note: Scales Differ)

the optimal-encoding fixed-cluster results (Table 9.1, page 172), the self-organizing algorithm is consistently better in every environment, with the exception of PVM. For PVM, the ratios are no worse, but the cluster-size range has increased from 2–8 to 2–16. The lack of improvement for PVM was not surprising, as fixed clusters were clearly a good fit. However, the range increase was surprising, and so we investigated it further. First, we analysed the near-optimal maximum-cluster-size to determine if this quality had degraded. The result of this investigation, shown in Figure 9.19(a), was that there was no significant change. Clearly the effect of this larger range was in no significant way detrimental. We then determined which computations exhibited a minimum timestamp-size ratio at a maximum-cluster-size of between 9 and 16. This revealed various computations, such as that whose timestamp-size ratio is shown in Figure 9.19(b). What has happened is that the self-organizing-cluster algorithm has smoothed out the variability that occurs, even occasionally in PVM, when using fixed clusters. This can be seen most explicitly when we compare the range of cluster-size that results in a timestamp within 5%, 10%, and 20% of the optimal, as displayed in Table 9.4. Thus the increase in cluster-size for optimum timestamp-size reflects an improvement, not a decline, in the PVM results.

The summary results could easily hide individual computations experiencing a significantly-worse average timestamp-size than was the case with the fixed-clusters. We therefore examined

Proximity	Fixed Clusters	Self-Organizing Clusters
5%	8, 24	8, 10–26
10%	4, 8, 12, 17, 20, 23–24	8–35
20%	4–24, 26–28	4–46

Table 9.4: Cluster-Size Ranges for PVM Sample

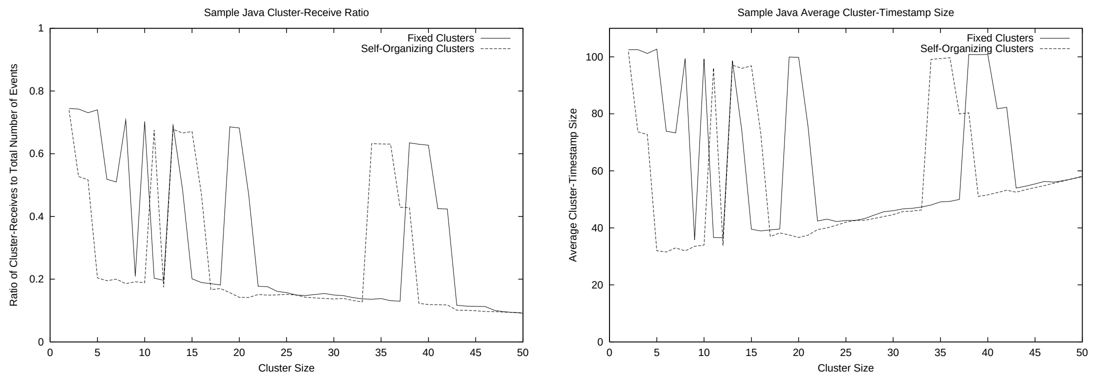
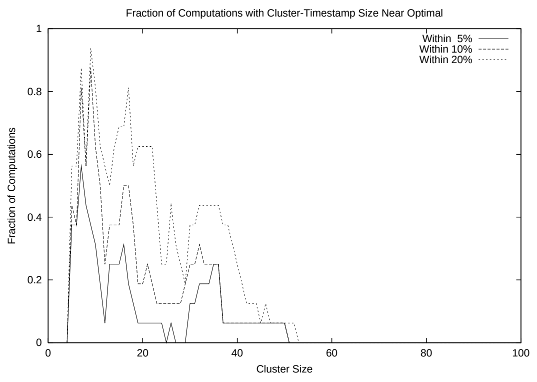


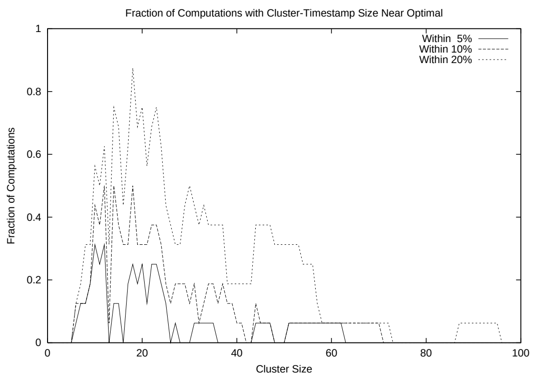
Figure 9.20: Sample Self-Organizing Cluster-Receive and Timestamp-Size Results

the individual computations and determined that this effect of more consistent and slightly better space-consumption was true over almost all experiments that we ran. Further, in no instance did our self-organizing cluster-timestamps achieve worse results than pre-determined clusters. In the cases where the results were no better, it was readily observed that the pre-determined clusters happened to be a good selection. This typically occurred in PVM programs where the communication was extremely regular and coincided with the pre-determined clusters.

We now turn to a somewhat more-detailed analysis of the Java results, as there was no ideal cluster-size range that satisfied all Java computations in the fixed-cluster algorithm. Sample results for cluster-receive ratio and timestamp size are shown in Figure 9.20. In those figures we display both the self-organizing and fixed data for a single Java computation. As can readily be seen, the self-organizing algorithm has not removed all deficiencies, but it has produced a sizable range in which the cluster-timestamp size is at or near minimum. In particular, this example demonstrates a size ratio of between 23 and 25 percent of Fidge/Mattern over a range of maximum cluster size from 5 to 10. By contrast, the pre-determined cluster algorithm experienced a size ratio of between 25 and 70 percent of Fidge/Mattern over that same range of cluster size. Simply put, the self-organizing clusters are far more stable in their space-consumption requirement than are the pre-determined clusters. While there are still spikes in the Java results, they are less frequent, and more easily avoided by our observation tool. These deficiencies are not surprising given the merging rule. We expect a more-sophisticated merging requirement would remove much of the remaining variability.

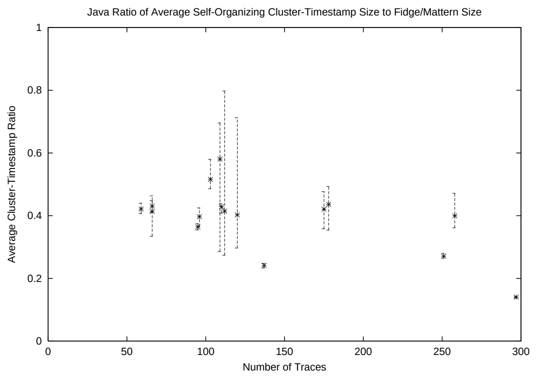


(a) Self-Organizing Clusters

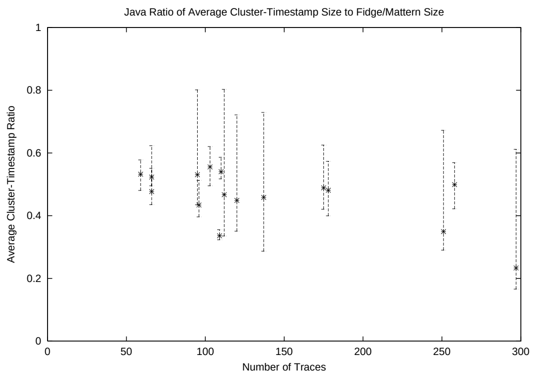


(b) Fixed Clusters

Figure 9.21: Optimum Cluster-Size Selection



(a) Self-Organizing Clusters (7–10)



(b) Fixed Cluster (17–23)

Figure 9.22: Timestamp Size for Optimum Cluster-Size Selection

The aggregate data for Java computations displays a similar level of improvement. Figure 9.21(a) shows the fraction of computations that have a timestamp-size near to the optimal for the given cluster size. While we have yet to achieve a single cluster-range that allows all computations to be within 20% of optimal, we have exceeded the fixed-cluster results, which we have reproduced in Figure 9.21(b) for comparison purposes. Where the fixed clusters achieved 85% commonality within 20% of optimal, the self-organizing clusters exceed 93%. In addition, the location of the ideal range has shifted substantially. Where the preferred fixed-cluster range was between 17 and 23, the self-organizing clusters ideal is from 7 to 10. This is advantageous as it reduces the average cluster-timestamp size, since any non-cluster-receive events will require a timestamp of about half the size as is required for the fixed-cluster case. Based on the range of 7 to 10 we examined the minimum, maximum, and average timestamp-size for the various Java computations. These results are presented in Figure 9.22(a). Again, for comparison purposes, we have reproduced the analogous results for the fixed-cluster case in Figure 9.22(b). As can be seen, there are now just three computations with significant variability. What the self-organizing clusters have not yet achieved, however, is a more-substantial reduction in timestamp size over the fixed-cluster case.

This then leads to caching-mode analysis. The caching-mode results for Java are virtually indistinguishable from those of the two-level fixed-cluster algorithm. This is as expected, since the minimum cluster-size required for caching-mode to achieve a reasonable space-reduction is sufficiently large that it is past most of the variability in the Java results. A brief examination of the sample Java cluster-receive ratio, shown in Figure 9.11(a), indicates that most of the variability occurs when the cluster-size is less than 50. This was typical for Java computations. By contrast, the cluster-size required to achieve a 20%, or less, cluster-receive ratio was 87. Given this lack of change, we do not repeat the results here. The final comment that must be noted is that we have been conservative when determining an appropriate bound for caching-mode. That is, we have used the least cluster-size for which no greater cluster-size exceeds the desired threshold. This is arguably more conservative than is required, particularly when more-stable results are achieved. On that point we observe that the cluster-receive ratio for the Java computations averages 27% when in the cluster-size range of 7–10, and is as low as 11%. It is probably therefore reasonable to revisit the caching-mode approach, with a view to tightening the bound.

9.5 Analysis

In this chapter we have presented a novel algorithm for dynamic cluster-timestamps. These timestamps attempt to exploit communication locality as determined by an appropriate trace-clustering algorithm. We have presented experimental results that demonstrate that this timestamp can produce a significant space-consumption reduction, provided the clustering algorithm is adequate to the task. In particular, in our PVM experiments we were able to achieve up to an order-of-magnitude size reduction. We have also demonstrated that the self-organizing cluster algorithm is more stable with cluster size and provides timestamps whose average size is consistently superior to a pre-determined cluster approach, even with a poor clustering algorithm.

This algorithm is significant since it appears, subject to the cluster algorithm, to have satisfied our requirement for a scalable timestamp. The qualifiers are necessary for various reasons. Although the results are reasonably good for PVM, they are clearly less-so for the remaining environments. The evidence we have presented leads us to believe that the lesser reduction of the remaining environments was in some part due to an inadequate cluster algorithm. However, we also note that in many instances we cannot realistically expect a significant space-reduction, even with good communication locality. By way of example, consider a 100-trace computation with only 10% cluster-receive events and an optimum cluster-size of 10. In such a case, the average timestamp-size will be 19, or a four-fifths reduction. These are the type of results we see for PVM. In the other environments, the poor-nature of merging on first communication, or the worse fixed-cluster algorithm, coupled with the ubiquity of synchronous events in those environments, cannot hope to achieve only 10% cluster-receive events, and indeed the results indicate that it is between 20% and 30%. Such a rate cannot reduce space-consumption in a 100-trace computation by more than two-thirds. However, as we move to larger computations, and a hierarchical, rather than two-level, algorithm, even a 30% cluster-receive rate can result in an order-of-magnitude space saving. We can see something of this effect in the fixed-cluster hierarchical-squaring results for Java, which achieved a significant additional reduction over the two-level fixed approach.

CONCLUSIONS

10 CONCLUSIONS

In this dissertation we have developed algorithms to enable a scalable partial-order data structure for distributed-system observation. In the process we have provided the following significant contributions to scientific and engineering knowledge.

1. The formalization of the operations on a partial-order data structure for distributed-system observation.
2. Proof that timestamp vector-size affects the performance of precedence-test execution time in distributed-system-observation tools.
3. Evidence that practical distributed computations do not in general have large dimension.
4. The creation of a centralized, dynamic timestamp algorithm that scales with partial-order dimension, not width.
5. The creation of a centralized, dynamic timestamp algorithm based on capturing communication locality.

While we will not repeat the discussion from Section 1.2, we will highlight some of the more important aspects of our work. Clearly, our most significant achievement is the development of the dynamic-cluster-timestamp algorithm. This timestamp can be integrated, with some small amount of programming, into POET in place of its current use of Fidge/Mattern timestamps. The benefit of doing so would be the achievement of scalability in the data structure that is lacking in Fidge/Mattern timestamps.

Such an integration would take a non-trivial effort, in large measure because POET, like every other distributed-system observation tool, was not built around the concept of a partial-order data structure. Rather, it was designed around the idea of events with logical timestamps, and even there, the formal properties of the timestamps were not enforced though an appropriate interface. It is for this reason that we spent a significant effort formalizing the requirements and interface of the partial order. Insofar as POET, or any other observation tool, is built on that interface, we can arbitrarily replace the implementation as is appropriate to the circumstances.

Also of substantial significance is the clear evidence we have presented of the scalability problems experienced by Fidge/Mattern timestamps. One of our more interesting findings was that the constant-time precedence test is not constant-time. An in-core data structure built using Fidge/Mattern timestamps will experience very poor processor-cache performance. When the data-structure exceeds core-memory size, it will either page, very poorly, or a caching scheme must be used, which results in $O(N)$ time for precedence tests. These observations enabled us to develop the caching-mode for the cluster-timestamp. While that mode needs further study, we believe it can substantially out-perform a large Fidge/Mattern-based structure.

Finally, we note that our dynamic-Ore timestamp may have longer-term significance, though this would depend on a substantial amount of further research, as we enumerate below. However, its value, from our perspective, is that its development enabled us to create the techniques needed to turn Summers' static cluster-timestamp into a dynamic timestamp. The dynamic-Ore algorithm itself depended on the quite curious online dimension-bound algorithm [165], which, at the time of its creation, served no clear purpose. We wonder if some of this work might be of value to those who work on partial-order theory.

10.1 Future Work

There are several areas of future work that we are actively exploring, which can be largely broken down into three categories. First, we wish to deal with the apparent functionality gap in the data-structure interface that exists between precedence testing and precedence-related event sets. There are two approaches that might be taken to this problem. The straightforward method would be to alter the interface to provide some intermediate operations. Probably the most appropriate route to take for this would be incorporation of trace-abstraction (Section 3.10.2) into the interface. If well-designed, this should allow the provision of the precedence-related event sets by abstract trace, rather than over the whole computation. Note that these abstract traces should not be confused with clusters in the dynamic-cluster timestamp. We most explicitly do not wish to expose the underlying timestamp algorithm through the interface. The alternate approach, that requires no change to the current interface, and is thus also worth investigating, is the application of lazy evaluation to the precedence-related event sets. This approach would require usage-pattern information to determine its value.

Second, there are several aspects in the dynamic-Ore timestamp that require further investigation. At a relatively easy level, a lower-bound dimension-analysis algorithm is clearly required. This is needed to determine if computations such as Life really do have a high dimension and to allow us to start classifying computations by dimension. A possible approach to this would be to seek cliques of mutually-incompatible critical pairs. The largest such clique would yield a lower-bound on the dimension. This technique effectively amounts to looking for an embedding of Crown $\mathbf{S}_N^0$ within the partial order, which is not strictly required for a dimension of size N. However, it would be a reasonable first-step, as a suitable algorithm would run efficiently, while computing the dimension is NP-hard.

Next, we would like to perform some experimental analysis of the dynamic-Ore algorithm. In particular, it is unclear at this juncture whether or not the dimension-bounds determined by our analysis are achievable in a dynamic-Ore timestamp. Even given the success of this, two other problems need to be addressed. The pseudo-realizer encoding would likely have to be much more efficient than our naive approach. If such an encoding cannot be developed (though there is reason to believe that it can [120]), then this timestamp would only be of value when the width exceeds the dimension by at least a couple of orders-of-magnitude. This would substantially limit its applicability. Second, a suitable caching needs to be developed. We currently have no idea how this might be achieved.

Third, we wish explore various aspects of our dynamic-cluster timestamp. We are currently looking at alternate, more sophisticated, dynamic-clustering algorithms. Specifically, our merging algorithm allows us to observe several communication events before deciding on the suitability of a cluster merger. We have not yet evaluated the effectiveness of this capability. The tradeoff in the use of the capacity is an increase in the number of false cluster-receive events. Second, and in a similar vein, we would like to extend this work to enable dynamically-reconfigurable clusters. That is, we would like the capacity to shift traces between clusters, and clusters between higher-level clusters. This is required when communication patterns are not stable in long-running computations or to correct mistakes made by the dynamic clustering algorithm. Third, we need to more thoroughly explore the tradeoffs in cluster size and arrangement for the hierarchical algorithm with three or more cluster levels. This is a complex tradeoff between average timestamp size, timestamp-creation time, precedence-test time, and greatest-predecessor-set computation time. Likewise, we wish to explore static cluster-analysis techniques to determine if they can be applied to improve the performance of the fixed-cluster approach. Fourth, the caching-mode of the timestamp needs to be further explored. In particular, we are currently far too conservative in assessing an appropriate bound for caching-mode use. We believe that we can be more aggressive in the bound-selection, particularly as we develop improved clustering capabilities.

Finally, we wish to integrate this timestamp technique into POET. There are several research motivations for this. First, the POET system is based on fixed-size Fidge/Mattern timestamps, with their distributed-precedence-testing capability. The implications of variable-size timestamps requiring centralized-timestamp testing are unclear. Second, it would enable the exploration of various caching strategies which cannot be evaluated by a POET client. Third, insofar as the timestamp is scalable, it will enable us to gather data from significantly larger computations. This will open up various research fronts, not least of which is how such large computations might reasonably be displayed to a user of the tool [167].

We also anticipate that our work might have application in other areas. In particular, partial orders are not uncommon in various fields, including scheduling, information structuring, and in any domain where directed-acyclic graphs are present. The limitations of our work to application beyond distributed-system observation are in the implicit requirement to process events in a linearization of the partial order (needed by the Fidge/Mattern computation) and in the nature of communication-locality and dimension-bound of distributed systems. Abdeddaim [1] developed a technique that uses the ideas of Fidge/Mattern timestamps to maintain dynamic directed graphs, allowing arbitrary edge insertion. If our timestamp can be melded with that technique then, subject to the nature of the data, it could develop wider applicability.

REFERENCES

[1] Saïd Abdeddaïm. On incremental computation of transitive closure and greedy alignment. In Alberto Apostolico and Jotun Hein, editors, *Proc. 8th Symp. Combinatorial Pattern Matching*, number 1264 in Lecture Notes in Computer Science, pages 167–179. Springer-Verlag, 1997.

[2] Frank Adelstein and Mukesh Singhal. Real-time causal message ordering in multimedia systems. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, pages 36–43, Vancouver, June 1995. IEEE Computer Society Press.

[3] Rosario Aiello, Elena Pagani, and Gian P. Rossi. Causal ordering in reliable group communication. *Computer Communication Review*, 23(4):106–115, October 1993.

[4] M. R. Anderberg. *Cluster Analysis for Applications*. Academic Press, 1973.

[5] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin-Cummings, Redwood City, California, 1991.

[6] Twan Basten, Thomas Kunz, James P. Black, Michael H. Coffin, and David J. Taylor. Vector time and causality among abstract events in distributed computations. *Distributed Computing*, 11(1):21–39, December 1997.

[7] Twan A. Basten. Hierarchical event-based behavioural abstraction in interactive distributed debugging: A theoretical approach. Master's thesis, Eindhoven University of Technology, Eindhoven, 1993.

[8] Peter C. Bates. Debugging heterogenous distributed systems using event-based models of behaviour. *ACM SIGPLAN Notices*, 24(1):11–22, January 1989.

[9] Adam Begulin. Xab: A tool for monitoring PVM programs. Technical Report CMU-CS-93-164, Carnegie Mellon University, School of Computer Science, Carnegie Mellon University, June 1993.

[10] Adam Begulin and Erik Seligman. Causality-preserving timestamps in distributed programs. Technical Report CMU-CS-93-167, Carnegie Mellon University, School of Computer Science, Carnegie Mellon University, June 1993.

[11] Eike Best and Brian Randell. A formal model of atomicity in asynchronous systems. *Acta Informatica*, 16:93–124, 1981.

[12] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Compueter Systems*, 5(1):47–76, February 1987.

[13] Kenneth P. Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[14] Peter A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: Concurrency in the Object-Oriented Language C++. *Software — Practice and Experience*, 22(2):137–172, February 1992.

[15] Thomas L. Casavant and Mukesh Singhal, editors. *Readings in Distributed Computing Systems*. IEEE Computer Society Press, Los Alamitos, California, 1994.

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

[17] Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, asynchronous and causally ordered communication. Submitted to Distributed Computing.

[18] Craig M. Chase and Vijay K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11:191–201, 1998.

[19] David R. Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 44–57, New York, 1993. ACM SIGOPS.

[20] Wing Hong Cheung. *Process and Event Abstraction for Debugging Distributed Programs*. PhD thesis, University of Waterloo, Waterloo, Ontario, 1989.

[21] Wing Hong Cheung, James P. Black, and Eric Manning. A framework for distributed debugging. *IEEE Software*, 7(1):106–115, January 1990.

[22] Mark Christiaens and Koen De Bosschere. Accordion clocks: Logical clocks for data race detection. In Sakellariou et al. [128], pages 494–503.

[23] Marshall Cline. C++ FAQ Lite. Available at www.parashift.com/c++-faq-lite/, 1991–2001.

[24] Marshall Cline, Greg Lomow, and Mike Girou. *C++ FAQs*. Addison-Wesley, 1999.

[25] Mariano Consens, Masum Hasan, and Alberto O. Mendelzon. Debugging distributed and parallel programs by visualizing and querying event traces. In Mendelzon [106]. CSRI-285.

[26] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 2nd edition, 2001.

[27] Janice Cuny, George Forman, Alfred Hough, Joydip Kundu, Calvin Lin, Lawrence Snyder, and David Stemple. The ariadne debugger: Scalable application of event-based abstraction. *ACM SIGPLAN Notices*, 28(12):85–95, May 1993.

[28] Suresh K. Damodaran-Kamal and Joan M. Francion. Testing races in parallel programs with an OtOt strategy. In T. Ostrand, editor, *Proceedings of the 1994 International Symposium On Software Testing and Analysis*, August 1994.

[29] Suresh K. Damodaran-Kamal and Joan M. Francioni. Nondeterminancy: Testing and debugging in message passing parallel programs. *ACM SIGPLAN Notices*, 28(12):118–128, May 1993.

[30] Frederica Darema, David A. George, V. Alan Norton, and Gregory F. Pfister. A single-program-multiple-data computation model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, April 1988.

[31] William H.E. Day and Herbert Edelsbrunner. Efficient algorithms for agglomererative hierarchical clustering methods. *Journal of Classification*, 1(7):7–24, 1984.

[32] Erik Demaine. Space saving by differential encoding of timestamps in POET. Personal Communication, October 2000.

[33] Giuseppe Di Battista, Peter Eadea, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: An annotated bibliography. *Computational Geometry: Theory and Applications*, 4(5):235–282, June 1994.

[34] Edsger Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[35] Paul S. Dodd and Chinya V. Ravishankar. Monitoring and debugging distributed real-time programs. *Software — Practice and Experience*, 22(10):863–877, October 1992.

[36] Albert Einstein. Zur elektrodynamik bewegter körper. *Annalen der Physik*, 17, June 1905. English translation, ”On the Electrodynamics of Moving Bodies” available at http://www.fourmilab.ch/etexts/einstein/specrel/www/.

[37] Greg Eisenhauer, Weiming Gu, Eileen Kraemer, Karsten Schwan, and John Stasko. Online displays of parallel programs: Problems and solutions. In *Proceedings of the International Conference on Parallel and Distributed Prcessing Techniques and Applications*, pages 11–20. PDPTA'97, 1997.

[38] Greg Eisenhauer and Karsten Schwan. An object-based infrastructure for program monitoring and steering. In *Proceedings 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 10–20, 1998.

[39] Darren Esau. Efficient detection of data races in SR programs. Master's thesis, University of Waterloo, Waterloo, Ontario, 1996.

[40] Etnus. TotalView users guide. Technical Report http://www.etnus.com/Support/docs/-rel5/html/user_guide/, Etnus, LLC, 2001.

[41] Paul D. Ezhilchelvan, Raimundo A. Macêdo, and Santosh K. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, pages 296–306, Vancouver, June 1995. IEEE Computer Society Press.

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

[43] Colin Fidge. Fundamentals of distributed systems observation. Technical Report 93-15, Software Verification Research Centre, Department of Computer Science, The University of Queensland, St. Lucia, QLD 4072, Australia, November 1993.

[44] Open Software Foundation. *Introduction to OSF/DCE*. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[45] 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.

[46] Mark Robert Fox. Event-predicate detection in the monitoring of distributed applications. Master's thesis, University of Waterloo, Waterloo, Ontario, December 1998.

[47] Daniel P. Friedman and David S. Wise. CONS shuld not evaluate its arguments. In S. Michaelson and Robin Milner, editors, *Proceedings of the Third EATCS International Colloquium on Automata, Languages and Programming*, pages 257–284, Edinburgh, Scotland, 1976. Edinburgh University Press.

[48] Michael Frumkin, Robert Hood, and Jerry Yan. On the information content of program traces. Technical Report NAS-98-008, NAS Parallel Tools Group, NASA Ames Research Center, Mail Stop 258-6 or T27A-1, Moffett Field, CA 94035-1000, March 1998.

[49] Jason Gait. A probe effect in concurrent programs. *Software — Practice and Experience*, 16(3):225–233, March 1986.

[50] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, New Jersey, 2002.

[51] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, 1979.

[52] Vijay K. Garg and Chakarat Skawratananond. String realizers of posets with applications to distributed computing. In *ACM Symposium on Principles of Distributed Computing*, August 2001.

[53] Al Geist, Adam Begulin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachusetts, 1994.

[54] Al Globus and Sam Uselton. Evaluation of visualization software. *Computer Graphics*, 29(2), May 1995. NAS Technical Report NAS-95-005.

[55] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996. Available at http://java.sun.com/docs/books/jls/.

[56] Siegfried Grabner, Dieter Kranzlmüller, and Jens Volkert. EMU — Event Monitoring Utility. Technical report, Institute for Computer Science, Johannes Kepler University Linz, July 1994.

[57] Boris Gruschke. A new approach for event correlation based on dependency graphs. In *Proceedings of the 5th Workshop of the OpenView University Association: OVUA'98*, April 1998.

[58] Weiming Gu, Greg Eisenhauer, Karsten Schwan, and Jeffrey Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 6(2):699–736, 1998.

[59] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Mullender [110], pages 97–145.

[60] Jessica Zhi Han. Automatic comparison of execution histories in the debugging of distributed applications. Master's thesis, University of Waterloo, Waterloo, Ontario, 1998.

[61] Paul K. Harter, Dennis M. Heimbigner, and Roger King. IDD: An interactive distributed debugger. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 498–506, May 1985.

[62] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, pages 29–39, September 1991.

[63] P. Henderson and J. Morris. A lazy evaluator. In *Third Symposium on Principles of Programming Languages*, pages 95–103. ACM, 1976.

[64] Alfred A. Hough and Janice E. Cuny. Belvedere: Prototype of a pattern-oriented debugger for highly parallel computation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 735–738, 1987.

[65] IBM Corporation. WebSphere application server, object level trace. Technical Report http://www-4.ibm.com/software/webservers/appserv/olt.html, IBM Corporation, 1998.

[66] IBM Corporation. *AIX 5L Version 5.1: Performance Management Guide*. IBM Corporation, 2nd edition, April 2001. Available at http://as400bks.rochester.ibm.com/doc_ link/-en_US/a_doc_lib/aixbman/prftungd/prftungd.htm.

[67] Intel Corporation. System performance visualization tool user's guide. Technical Report 312889-001, Intel Corporation, 1993.

[68] Christian E. Jaekl. Event-predicate detection in the debugging of distributed applications. Master's thesis, University of Waterloo, Waterloo, Ontario, 1997.

[69] Pankaj Jalote. *Fault Tolerance in Ditsributed Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[70] Claude Jard and Guy-Vincent Jourdan. Dependency tracking and filtering in distributed computations. Technical Report 851, IRISA, Campus de Beaulieu – 35042 Rennes Cedex – France, August 1994.

[71] Dean Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. In *International Conference on Software Engineering*, pages 360–370. IEEE, 1997.

[72] Dean F. Jerding and John T. Stasko. Using visualization to foster object-oriented program understanding. Technical Report GIT-GVU-94-33, Georgia Institute of Technology, Atlanta, GA, USA, July 1994.

[73] Dean F. Jerding and John T. Stasko. The information mural: A technique for displaying and navigating large information spaces. Technical Report GIT-GVU-97-24, Georgia Institute of Technology, Atlanta, GA, 1997.

[74] Harry F. Jordan. Structuring parallel algorithms in an MIMD shared memory environment. *Parallel Computing*, 3(2):93–110, May 1986.

[75] L. Kaufman and P.J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, New York, 1990.

[76] Richard B. Kilgore and Craig M. Chase. Testing distributed programs containing racing messages. *The Computer Journal*, 40(8):489, 1997.

[77] Valerie King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *40th Annual Symposium on Foundations of Computer Science*, pages 81–91. IEEE Computer Society, 1999.

[78] Valerie King and Garry Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing*, pages 492–498. ACM, 1999.

[79] Shmuel Kliger, Shaula Yemini, Yechiam Yemini, David Ohsie, and Salvatore Stolfo. A coding approach to event correlation. In *Integrated Network Management IV* [132], pages 266–277.

[80] James Arthur Kohl and Al Geist. XPVM 1.0 user's guide. Technical Report TM-12981, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, November 1996.

[81] Eileen Kraemer. Causality filters: A tool for the online visualization and steering of parallel and distributed programs. In *Proceedings of the 11th IPPS*, pages 113–120. IEEE, 1997.

[82] Eileen Kraemer and John T. Stasko. Creating an accurate portrayal of concurrent executions. *Concurrency*, 6(1):36–46, 1998.

[83] Dieter Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, GUP Linz, Johannes Kepler Universität Linz, Linz, Austria, September 2000.

[84] Dieter Kranzlmüller, Siegfried Grabner, R. Schall, and Jens Volkert. ATEMPT — A Tool for Event ManiPulaTion. Technical report, Institute for Computer Science, Johannes Kepler University Linz, May 1995.

[85] Dieter Kranzlmüller, Siegfried Grabner, and Jens Volkert. PARASIT — Parallel simulation tool. Technical report, Institute for Computer Science, Johannes Kepler University Linz, December 1994.

[86] Dieter Kranzlmüller, 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, September 1995.

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

[88] Dieter Kranzlmüller and Jens Volkert. Debugging point-to-point communication in MPI and PVM. In *Proceedings of EUROPVM/MPI '98*, volume 1497 of *Lecture Notes in Computer Science*, pages 265–272. Springer-Verlag, September 1998.

[89] Thomas Kunz. *Abstract Behaviour of Distributed Executions with Applications to Visualization*. PhD thesis, Technische Hochschule Darmstadt, Darmstadt, Germany, 1994.

[90] Thomas Kunz. Visualizing abstract events. In *Proceedings of the 1994 CAS Conference*, pages 334–343, October 1994.

[91] Thomas Kunz. Automatic support for understanding complex behaviour. In *Proceedings of the International Workshop on Network and Systems Management*, pages 125–132, August 1995.

[92] Thomas Kunz. High-level views of distributed executions. In *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging*, pages 505–512, May 1995.

[93] Thomas Kunz. Evaluating process clusters to support automatic program understanding. In *WPC '96: Proceedings of the IEEE Fourth Workshop on Program Comprehension*, (Berlin, Germany; March 29-31, 1996), pages 198–207. IEEE Computer Society Press, March 1996.

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

[95] 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.

[96] Leslie Lamport. Time, clocks and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, 1978.

[97] Leslie Lamport and Nancy Lynch. Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science*, volume 2, pages 1157–1199. Elsevier Science Publishers B. V., 1990.

[98] Thomas J. LeBlanc, John M. Mellor-Crummey, and Robert J. Fowler. Analyzing parallel program executions using multiple views. *Journal of Parallel and Distributed Computing*, 9:203–217, 1990.

[99] Masoud Mansouri-Samani and Morris Sloman. Monitoring distributed systems (a survey). Technical Report DOC92/23, Imperial College of Science, Technology and Medicine, 1992.

[100] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau de Bonas, France, December 1988. Elsevier Science Publishers B. V. (North Holland).

[101] Friedemann Mattern and S. Funfrocken. A non-blocking lightweight implementation of causal order message delivery. Technical Report TR-VS-95-01, Technical University of Darmstadt, Department of Computer Science, Technical University of Darmstadt, Germany, March 1995.

[102] Paul Mazzucco. The fundamentals of cache. Available at http://www.systemlogic.net/-articles/00/10/cache/, October 2000.

[103] Paul Mazzucco. Intel pentium 4: In-depth techincal overview. Available at http://www.-systemlogic.net/articles/01/8/p4/, August 2001.

[104] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4), December 1989.

[105] Sigurd Meldal, Sriram Sankar, and James Vera. Exploiting locality in maintaining potential causality. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 231–239, May 1991.

[106] Alberto O. Mendelzon, editor. *Declarative Database Visualization: Recent Papers from the Hy+/GraphLog*. CSRI, University of Toronto, June 1993. CSRI-285.

[107] Barton P. Miller and Jong-Deok Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th IEEE International Conference on Distributed Computing Systems*, pages 316–323. IEEE Computer Society Press, 1988.

[108] Gordan Moore. Moore's law. In *The New Hacker's Dictionary* [121]. Also available online at http://www.intel.com/intel/museum/25anniv/hof/moore.htm.

[109] MPI Forum. The message passing interface (MPI) standard. Available at: http://www-unix.mcs.anl.gov/mpi/indexold.html.

[110] Sape Mullender, editor. *Distributed Systems*. Addison-Wesley, New York, 2nd edition, 1993.

[111] Robert H.B. Netzer and Yikand Xu. Replaying distributed programs without message logging. In *Proceedings of the Sixth International Symposium on High Performance Distributed Computing*, August 1997.

[112] Oleg Y. Nickolayev, Philip C. Roth, and Daniel A. Reed. Real-time statistical clustering for event trace reduction. *Journal of Supercomputing Applications and High-Performance Computing*, 11(2):144–159, 1997.

[113] Ernst-Rüdiger Olderog. Operational petri net semantics for CCSP. *IEEE Network*, pages 34–43, September/October 1997.

[114] Oystein Ore. *Theory of Graphs*, volume 38. Amer. Math. Soc. Colloq. Publ., Providence, R.I., 1962.

[115] Cherri M. Pancake. Applying human factors to the design of performance tools. In P. Amestoy, P. Berger, M. Daydé, I. Duff, V. Frayssé, L. Giraud, and D. Ruiz, editors, *EuroPar'99 Parallel Processing*, Lecture Notes in Computer Science, No. 1685, pages 44–60. Springer-Verlag, 1999.

[116] Guru Parulkar, Douglas Schmidt, Eileen Kraemer, Jonathan Turner, and Anshul Kantawala. An architecture for monitoring, visualization, and control of gigabit networks. *IEEE Network*, pages 34–43, September/October 1997.

[117] Alexander Peleg and Uri Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. United States Patent 5,381,533, January 1995. Available at: http://www.uspto.gov/.

[118] Beth Plale, Greg Eisenhauer, Karsten Schwan, Jeremy Heiner, Vernard Martin, and Jeffrey Vetter. From interactive applications to distributed laboratories. *IEEE Concurrency*, 6(2):78–90, 1997.

[119] Beth Plale and Karsten Schwan. Run-time detection in parallel and distributed systems: Application to safety-critical systems. In *International Conference on Distributed Computing Systems*, pages 163–170, 1999.

[120] Darrell Raymond. *Partial Order Databases*. PhD thesis, University of Waterloo, Waterloo, Ontario, 1996.

[121] Eric S. Raymond. *The New Hacker's Dictionary*. MIT Press, 3rd edition, 1996. Also available online as *The Jargon File* at http://www.jargon.org.

[122] Michel Raynal, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, 1991.

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

[124] Golden G. Richard III. Efficient vector time with dynamic process creation and termination. *Journal of Parallel and Distributed Computing*, 55(1):109–120, 1998.

[125] Luis E. T. Rodrigues and Paulo Verissimo. Causal separators for large-scale multicast communication. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, pages 83–91, Vancouver, June 1995. IEEE Computer Society Press.

[126] David A. Rusling. *The Linux Kernel*. Linux Documentation Project, 1996–1999. Version 0.8-3. Available at http://www.linuxdoc.org/LDP/tlk/tlk.html.

[127] Mark Russinovich. Inside memory management. *Windows NT Magazine*, August 1998. Available at http://www.winntmag.com/Articles/Index.cfm?IssueID=56.

[128] Rizos Sakellariou, John Keane, John Gurd, and Len Freeman, editors. *EuroPar'01 Parallel Processing*, volume LNCS 2150 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2001.

[129] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[130] André Schiper, Jorge Eggli, and Alain Sandoz. A new algorithm to implement causal ordering. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, pages 219–232, Berlin, 1989. Springer.

[131] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[132] Adarshpal S. Sethi, Yves Raynaud, and Fabienne Faure-Vincent. *Integrated Network Management IV*. Chapman and Hall, 1995.

[133] William Shakespeare. *Romeo and Juliet*. 1597. Available in “The Complete Works of William Shakespeare” Chatham River Press, New York, 1975.

[134] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3), September 1989.

[135] Joseph L. Sharnowski and Betty H. C. Cheng. A visualization-based environment for top-down debugging of parallel programs. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 640–645. IEEE Computer Society Press, 1995.

[136] Mukesh Singhal and Ajay Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43:47–52, August 1992.

[137] Richard M. Stallman. Debugging with GDB: The GNU source-level debugger. Technical Report http://www.gnu.org/manual/gdb-4.17/gdb.html, Free Software Foundation, 1998.

[138] Janice M. Stone. A graphical representation of concurrent processes. *ACM SIGPLAN Notices*, 24(1):226–235, January 1989.

[139] Robert E. Strom et al. *Hermes: A Language for Distributed Computing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

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

[141] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, New Jersey, 3rd edition, 1996.

[142] Ashis Tarafdar and Vijay K. Garg. Addressing false causality while detecting predicates in distributed programs. In *International Conference on Distributed Computing Systems*, pages 94–101, 1998.

[143] David J. Taylor. Scrolling displays of partially ordered execution histories. In preparation.

[144] David J. Taylor. A prototype debugger for hermes. In *Proceedings of the 1992 CAS Conference*, volume 1, pages 29–42, November 1992.

[145] David J. Taylor. The use of process clustering in distributed-system event displays. In *Proceedings of the 1993 CAS Conference*, pages 505–512, November 1993.

[146] David J. Taylor. Integrating real-time and partial-order information in event-data displays. In *Proceedings of the 1994 CAS Conference*, pages 505–512, November 1994.

[147] David J. Taylor. Event displays for debugging and managing distributed systems. In *Proceedings of the International Workshop on Network and Systems Management*, pages 112–124, August 1995.

[148] David J. Taylor, Thomas Kunz, and James P. Black. Achieving target-system independence in event visualisation. In *Proceedings of the 1995 CAS Conference*, pages 296–307, November 1995.

[149] Brad Topol, John T. Stasko, and Vaidy Sunderam. Integrating visualization support into distributed computing systems. Technical Report GIT-GVU-94-38, Georgia Institute of Technology, Atlanta, GA, October 1994.

[150] Brad Topol, John T. Stasko, and Vaidy S. Sunderam. Dual timestamping methodology for visualizing distributed application behaviour. *International Journal of Parallel and Distributed Systems and Networks*, 1(2):43–50, 1998.

[151] Brad Topol, John T. Stasko, and Vaidy S. Sunderam. PVaniM: A tool for visualization in network computing environments. *Concurrency: Practice and Experience*, 10(14):1197–1222, 1998.

[152] Francisco J. Torres-Rojas. Performance evaluation of plausible clocks. In Sakellariou et al. [128], pages 476–481.

[153] Francisco J. Torres-Rojas and Mustaque Ahamad. Plausible clocks: Constant size logical clocks for distibuted systems. *Distributed Computing*, 12:179–195, 1999.

[154] William T. Trotter. Graphs and partially-ordered sets. In R. Wilson and L. Beineke, editors, *Selected Topics in Graph Theory II*, pages 237–268. Academic Press, 1983.

[155] William T. Trotter. *Combinatorics and Partially Ordered Sets: Dimension Theory*. Johns Hopkins University Press, Baltimore, MD, 1992.

[156] William T. Trotter. Partially ordered sets. In R. Graham, M. Grötschel, and L. Lovász, editors, *Handbokk of Combinatorics*, pages 433–480. Elsevier Science, 1995.

[157] G. J. W. van Dijk and A. J. van der Wal. Partial ordering of synchronization events for distributed debugging in tightly-coupled multiprocessor systems. In *Proceedings of the 2nd European Distributed Memory Computing Conference*, number 487 in Lecture Notes in Computer Science, pages 100–109. Springer-Verlag, 1991.

[158] Jeffrey Vetter and Karsten Schwan. Models for computational steering. Technical Report GIT-CC-95-39, Georgia Institute of Technology, Atlanta, GA, 1995.

[159] Jeffrey Vetter and Karsten Schwan. Progress: A toolkit for interactive program steering. In *Proceedings of the 24th International Conference on Parallel Processing*, pages II:139–142, Oconomowoc, WI, 1995.

[160] Jeffrey S. Vetter. Computational steering annotated bibliography. *ACM SIGPLAN Notices*, 32(6):40–44, June 1997.

[161] Ellen M. Voorhees. Agglomererative hierarchical clustering algorithms for use in document retrieval. *Information Processing and Management*, 22:465–476, 1986.

[162] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc., November 1994.

[163] Paul A.S. Ward. On the scalability of distributed debugging: Vector clock size. Technical Report CS98-29, Shoshin Distributed Systems Group, Department of Computer Science, The University of Waterloo, Waterloo, Ontario, Canada N2L 3G1, December 1998. Available at ftp://cs-archive.uwaterloo.ca/cs-archive/CS-98-29/CS-98-29.ps.Z.

[164] Paul A.S. Ward. An offline algorithm for dimension-bound analysis. In Dhabaleswar Panda and Norio Shiratori, editors, *Proceedings of the 1999 International Conference on Parallel Processing*, pages 128–136. IEEE Computer Society, 1999.

[165] Paul A.S. Ward. An online algorithm for dimension-bound analysis. In P. Amestoy, P. Berger, M. Daydé, I. Duff, V. Frayssé, L. Giraud, and D. Ruiz, editors, *EuroPar'99 Parallel Processing*, Lecture Notes in Computer Science, No. 1685, pages 144–153. Springer-Verlag, 1999.

[166] Paul A.S. Ward. A framework algorithm for dynamic, centralized dimension-bounded timestamps. In *Proceedings of the 2000 CAS Conference*, pages 78–87, November 2000.

[167] Paul A.S. Ward. Issues in scalable distributed-system management. Technical Report CS-2001-01, Shoshin Distributed Systems Group, Department of Computer Science, The University of Waterloo, Waterloo, Ontario, Canada N2L 3G1, January 2001. Available at http://www.shoshin.uwaterloo.ca/˜pasward/Tech-Reports/CS-2001-01.ps.gz.

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

[169] Paul A.S. Ward and David J. Taylor. Self-organizing hierarchical cluster timestamps. In Sakellariou et al. [128], pages 46–56.

[170] Colin Ware and Glenn Franck. Evaluating stereo and motion cues for visualizing information nets in three dimensions. *ACM Transactions on Graphics*, 15(2), 1996.

[171] Gregory V. Wilson. The ansi c implementation of the cowichan problems. Technical report, University of Toronto, 1995.

[172] Gregory V. Wilson and R. Bruce Irvin. Assessing and comparing the usability of parallel programming systems. Technical report, University of Toronto, 1995.

[173] Roland Wismüller, Jöse Trinitis, and Thomas Ludwig. OCM — A monitoring system for interoperable tools. In *SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 1–9, New York, 1998. ACM.

[174] Jerry C. Yan, Haoqiang H. Jin, and Melisa A. Schmidt. Performance data gathering and representation from fixed-size statistical data. Technical Report NAS-98-003, Nasa Ames Research Center, Mail Stop T27A-1, Moffett Field, CA 94035-1000, February 1998.

[175] Jerry C. Yan, Sekhar R. Sarukkai, and Pankaj Mehra. Performance measurement, visualization and modelling of parallel and distributed programs using the aims toolkit. *Software — Practice and Experience*, 25(4):429–461, April 1995.

[176] Cheer-Sun D. Yang and Lori L. Pollock. The challenges in automated testing of multithreaded programs. In *Proceedings of the 14th International Conference on Testing Computer Software*, pages 157–166, June 1997.

[177] Mihalis Yannakakis. The complexity of the partial order dimension problem. *SIAM Journal on Algebraic and Discrete Methods*, 3(3):351–358, September 1982.

[178] Daniel M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30(4):369–384, 1993.

[179] Yuh Ming Yong. Replay and distributed breakpoints in an OSF DCE environment. Master's thesis, University of Waterloo, Waterloo, Ontario, 1995.

[180] Yuh Ming Yong and David J. Taylor. Performing replay in an OSF DCE environment. In *Proceedings of the 1995 CAS Conference*, pages 52–62, November 1995.