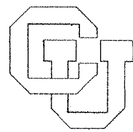
An Infrastructure to Generate Experimental Workloads for Persistent Object System Performance Evaluation

Thorna O. Humphries

CU-CS-906-00



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

An Infrastructure to Generate Experimental Workloads for Persistent Object System Performance Evaluation

Thorna O. Humphries

Department of Computer Science
University of Colorado
Boulder, CO 80309–0430 USA

University of Colorado
Department of Computer Science
Technical Report CU-CS-906-00 August 2000

© 2000 Thorna O. Humphries

Abstract

Performance evaluation of persistent object system implementations requires the use and evaluation of experimental workloads. Such workloads include a schema describing how the data are related, and application behaviors that capture how the data are manipulated over time. Currently, these experimental workloads generate data in a manner that does not support sharing of applications or data among researchers either because it is specific to a particular hardware platform or it is specific to a particular persistent object system. Using trace-driven simulation as a technique for analyzing the performance of persistent object systems, an infrastructure for generating experimental workloads and capturing their behavior is designed and implemented in this dissertation. This infrastructure consists of a common trace format that allows data to be shared among researchers and a modeling toolkit that reduces the effort to model, implement, and instrument applications. This infrastructure also consists of a new technique to generate multi-user workloads.

PTF (POSSE Trace Format) is a general-purpose trace format that is the specification of a set of events characterizing application operations on persistent object stores. PTF is novel in that the semantics of the higher-level application is maintained through the trace events (e.g., the notion of an object is captured in the trace events). It also captures the information about an application that is not specific to a particular persistent object system implementation.

AMPS (Application Modeling for Persistent Systems) is a toolkit that consists of a set of C++ classes and a TCL interface to ease the creation of self-tracing applications. The set of classes provides mechanisms for specifying a schema, coding application operations on the schema, and transparently instrumenting an application to record trace events. The TCL interface provides an interactive mechanism for specifying the workload of an application.

Several benefits can be derived from the use of the infrastructure of this dissertation. These benefits are as follows: the process of building new experiments for analysis is made easier; experiments to evaluate the performance of implementations can be conducted and reproduced with less effort; and pertinent information can be gathered in a cost-effective manner.

Keywords: experimental workloads, performance evaluation, trace formats, persistent object systems, benchmarking, multi-user workloads

Contents

1 Introduction **1**

1.1 The Problem 2
1.2 Approach 4
1.3 Evaluation 6
1.4 Contributions and Benefits 6
1.4.1 The Common Trace Format 6
1.4.2 The AMPS Toolkit 7
1.4.3 A New Technique to Generate Multi-user Workloads 7
1.5 Organization of Dissertation 7

2 Background **9**

2.1 Basic Terminology 9
2.2 Overview of Trace-Driven Simulation in Evaluation of Persistent Object Systems . . 10

3 Trace Format **13**

3.1 Design Goals 14
3.2 Overview of PTF Design 14
3.3 Trace Events 16
3.3.1 Trace Format 16
3.3.2 Events that Manipulate Objects of the Persistent Store 18
3.3.3 Garbage Collection Directives 26
3.4 Representation Formats 26
3.4.1 Binary Format 26
3.4.2 ASCII Format 26
3.5 Summary 27

4 Modeling Toolkit **28**

4.1 Motivation and Overview 28
4.2 Architecture 29
4.3 Modeling an Application Workload 30
4.3.1 Implementing the Persistent Store Schema 30
4.3.2 Implementing Application Behaviors 33
4.3.3 Implementing the Workload 35
4.4 Summary 36

5 **Experience Modeling OO7** 38
5.1 OO7 Overview 38
5.2 Using AMPS to Model OO7 39
5.2.1 Modeling the OO7 Benchmark Schema 40
5.2.2 Modeling Extents 40
5.2.3 Modeling a Traversal 41
5.3 Comparison of PTF Traces 46
5.4 Summary 48

6 **Infrastructure to Support Generation of Multi-user Workloads** 49
6.1 Setting the Stage 50
6.1.1 Design Goals 50
6.1.2 Approach To Support Generation of Multi-user Workload 51
6.1.3 Assumptions 51
6.2 Design Issues 53
6.2.1 Enforcing Correctness of Interleaved Traces 53
6.2.2 Data Sharing 53
6.2.3 Concurrency 54
6.3 PTF Extensions 55
6.3.1 Trace Formats for Transaction Events 55
6.4 Generation of the Workload 56
6.4.1 Enhancement to AMPS Toolkit 57
6.4.2 Implementing Several Single-user Workloads 57
6.5 Scenarios 58
6.5.1 Scenario 1: Treatment of Creation of Objects 58
6.5.2 Scenario 2: Example of an Interfering Transaction 61
6.5.3 Scenario 3: An Example of a Non-Conflicting Transaction 64
6.5.4 Scenario 4: Example of Structural Change to Shared Region 64
6.6 An Example Trace Merger 64
6.6.1 The Notion of Time 64
6.6.2 Synchronization Algorithm 64
6.6.3 Garbage Collector Interaction 67
6.6.4 Overview of Implementation 68
6.7 Summary 73

7 **Experience with OO7 Multi-user Workload Specification** 74
7.1 Overview of the Multi-user OO7 Benchmark 74
7.1.1 Description of the OO7 Persistent Store 74
7.1.2 Description of the OO7 Multi-User Workloads 75
7.2 Modifications to OO7 Multi-user Specification 78
7.3 Experiments 78
7.3.1 Experiment 1: Three Client Producer/Consumer Multi-user Workload . . . 78
7.3.2 Experiment 2: Two Client Multi-user Workload with Reorganizations 81
7.4 Observations 82

8 Related Work **85**
8.1 Trace-Driven Simulation . . . 85
8.2 Trace Formats for Performance Evaluation . . . 85
8.3 Performance Evaluation Based on Workload Models . . . 86

9 Conclusion **88**
9.1 Future Work . . . 89

A Parameters of PTF Trace events **93**
A.1 Trace.h File . . . 93

B Interface Specifications for Components of the Example Trace Merger **99**
B.1 Functions of the Trace Merger . . . 99
B.1.1 Main Function . . . 99
B.1.2 Scheduler Function . . . 100
B.1.3 Swapper Function . . . 100
B.2 The C++ Classes of the Trace Merger . . . 101
B.2.1 The LockManager Class . . . 101
B.2.2 ClientCoordinator Class . . . 102
B.2.3 Lock Class . . . 109
B.2.4 LRQueue Class . . . 109
B.2.5 LRQueueItr Class . . . 110
B.2.6 LockRequest Class . . . 111

Chapter 1

Introduction

In the 1980s, the complexity of data-intensive applications surpassed the support level of traditional database management systems. These applications were typically in such areas as computer-aided design (CAD), document preparation, and software engineering. These applications required the manipulation and management of complex objects that traditional database management systems were not designed to manage. Therefore, researchers in the database community as well as researchers in the programming language community developed solutions to solve the problem of managing complex object storage. The database researchers developed object-oriented database management systems while the programming language researchers extended languages to allow data to persist after a process terminated and called their systems, *Persistent Object Systems* (POS). Persistent object systems were developed to effectively move data between main memory and secondary storage. Thus, research in the area of persistent object systems is directed toward the design and implementation of efficient persistent stores that are type safe.

One of the earliest persistent object systems was POMS (Persistent Object Management System) [12], which was developed for the language PS-Algol. A major concern in the implementation of POMS was how to actually organize the data structures on disk. As with POMS, the central subsystem of persistent object systems is the storage manager. The storage manager has several responsibilities, such as placement of objects on secondary storage, transferring data between secondary storage and main memory, creating and updating objects, and concurrency control. Because of its crucial role, the performance of the storage manager is an important factor in the evaluation of a persistent object system. Therefore, much emphasis is placed on the design and implementation of algorithms that make up the storage management subsystem. These algorithms fall into such categories as clustering, the use of indexing, and storage reclamation.

To evaluate the performance of the storage management subsystem, several techniques have been employed. They are as follows:

- **Implementation of a Prototype**. This technique requires that a prototype of a persistent object system is implemented. Examples of prototypes that have been used to investigate storage management algorithms are Exodus [22] and more recently PJAMA [4], a system that adds orthogonal persistence to JAVA.

- **Benchmarking.** Benchmarking is a technique that has been widely used to evaluate the performance of various relational as well as object database commercial products for over a decade [37]. It is therefore no surprise that it has been adopted as a technique for evaluating the performance of persistent object systems. Conceptually, a benchmark consists of two elements: the structure of the persistent data and the workload, in other words, the behavior

of an application accessing and manipulating the data. The process of using a benchmark to assess a particular persistent object system involves executing or simulating the behavior of the application while collecting data reflecting its performance.

- **Simulation.** Discrete event simulations have been developed to evaluate the performance of persistent object systems. These simulations have been used in the evaluation of clustering algorithms and storage reclamation. At the core of the simulation technique is a model of a persistent object system. Examples are the client-server model of a persistent object store developed by Yong et al. [46] and more recently the generic discrete-event random simulation model, VOODB (Virtual Object-Oriented Database) [18]. VOODB has been used to simulate the behavior of the Texas persistent object store [42] as well as an object-oriented database management system.

- **Trace-driven Simulation.** A proven assessment technique for evaluation of proposed systems is trace-driven simulation [44] and at the heart of this technique is the sequence of application events called a *trace*. A trace-driven simulation consists of three phases: collection, reduction, and processing [27]. Performance evaluation of storage management algorithms is concerned with dynamic behavior; therefore, the application must be instrumented in such a way that the relevant events can be collected in a trace during execution. The trace is then used as input to the reduction and processing phases of the simulation. These phases use the trace events to evaluate the performance of algorithms related to persistent object systems. Trace-driven simulation has been used in the study of storage reclamation [13, 15, 16].

- **Analytical Models.** Analytical models are also used to evaluate storage management algorithms. These models cost less to implement then both prototypes and simulations. The models are usually not as detailed as those of simulations. An example of the use of analytical modeling is the work by Butler [7] on storage reclamation.

As shown in the literature [8, 12, 22, 33, 42], prototyping has been widely used to evaluate persistent object systems implementations. However, the cost of implementation and the lack of generality of prototypes have increased interest in simulation techniques that allow researchers to evaluate algorithms without implementing prototypes. Through simulations, components of a persistent object system can be implemented so that system-dependent information can be included as needed. Furthermore, alternative algorithms can be incorporated in the implementation of a model of a persistent object system. Because of its track record in the area of performance analysis of caches and main memory designs, this dissertation is based on the premise that trace-driven simulation is a viable technique for evaluating the performance of persistent object systems. Further, evidence of its use in the area of storage management was shown by the work of Cook et al. [13, 15, 16]. However, since trace-driven simulation has not been used extensively in the evaluation of persistent object systems, there is a need for a better infrastructure to alleviate some of the problems that researchers encounter in the application of this technique for evaluating storage management algorithms of persistent object systems.

1.1 The Problem

A key component of trace-driven simulation is the accurate representation of the behavior of an application through trace events. Unlike TPC [23], the benchmark for evaluating transaction processing, there are no benchmarks for storage reclamation. There are several reasons why such

a benchmark has not been developed. First, vendors of persistent object systems place controls on the publishing of performance numbers of their products as noted by Carey et al. [10]. Thus, it is difficult to obtain timings to perform direct comparisons of implementations. Secondly, accessibility to real applications is very difficult. These applications are not available because of the sensitive nature of the data that is stored in the persistent object store. This sensitivity may be due to the privacy rights of a person represented by the data, as in an accounting or a health care application, or to the proprietary nature of the data with respect to a specific company or organization (e.g., in a CAD application, a design that has not been patented). In any case, vendors of persistent object systems are not readily giving their test applications and data to academicians.

Without a benchmark, researchers in the POS community perform their analysis of storage algorithms by developing a variety of applications using several different platforms. Since there is no common data format, the implementations of the applications as well as the data generated are not typically shared among researchers.

In addition to the above, a lot of analysis has been performed using single-user workloads to generate data. However, very little analysis has been done with data generated from a multi-user workload, thus limiting the exploration of storage management algorithms. Evidence to substantiate the above claim was shown in the experimentation by Banerjee and Gardner [5], whose experimentation with the MMST/WORKS program lead them to the observation that multi-user testing was necessary in order to measure the amount of process interaction. They found that the interaction between processes provided insight into how to tune database activities such as clustering. They state that the support of a multi-user environment and workload would not only bring out the ability of an object database system to handle the workload but also throw light on the concurrency semantics of the object database system [5].

To date only the developers of the OO7 benchmark have investigated the design of a multi-user benchmark [9]. They found the development of a multi-user benchmark was more difficult due to the increased number of dimensions in which the workload can vary and due to the complexity of interactions among many concurrent activities. Although the developers performed some preliminary experimentation with their specification, they did not pursue it further because of insufficient information about what would constitute a multi-user workload for a typical object database application.

Finally, another problem with trace-driven simulation is the process of instrumentation to generate the trace. The instrumentation process is very time consuming, error prone, and costly. It is time consuming because it may take several attempts to correctly instrument an application. Also, the process of instrumentation, especially by hand, is error prone in that it is easily to miss some operations inadvertently, thus requiring careful inspection of the instrumented application. Cost of instrumentation is thus measured in terms of the time to complete instrumentation and whether the instrumented application is reusable on a variety of platforms.

Since all of the above problems cannot be addressed in one dissertation, the work in this dissertation is geared to answering the following two questions that arise with trace-driven simulation.

1. What is a good representation for a trace that allows one both to capture a wide range of application data and behaviors, and to share the traces among analysts?

2. How do we minimize the effort needed to instrument an application to create traces?

Thus, the goal of this dissertation is to identify and design mechanisms and tools that allow efficient gathering of more accurate information for effective performance evaluation of persistent object systems.

1.2 Approach

Although trace-driven simulation has three phases, there are several ways to approach each of these phases. In the collection phase, a component of a system is usually instrumented to generate trace events as the component is executing. From experimentation with garbage collection algorithms [13, 15, 16], it was observed that information that is needed to analyze these algorithms is not dependent on a particular persistent object system. This information consists of creations of new objects and modification of inter-object references. Thus, trace events can be collected by executing an instrumented persistent object application independently of a persistent object system. Thus, the approach shown in Figure 1.1 was developed. This approach to trace-driven simulation is based on AMPS (Application Modeling for Persistent Systems) and PTF (POSSE Trace Format), the two central components of the infrastructure investigated, designed, and developed in this dissertation.

AMPS consists of a set of C++ classes and a TCL interface to ease the creation of self-tracing applications. The set of classes provides mechanisms for specifying a schema, coding application operations on the schema, and transparently instrumenting an application to record trace events. The TCL interface provides an interactive mechanism for specifying the workload of an application in terms of persistent store behaviors such as generation, traversals, and updates.

Using AMPS, as shown in Figure 1.1, an application is modeled using a combination of a schema specification, an application specification, and the AMPS library; the schema and application specifications are defined using C++ classes derived from classes in our library. The application model is instrumented to produce PTF trace events using a library of the AMPS toolkit. The application model then runs without the need for an actual persistent object system, producing a PTF trace file that can then be fed into an analyzer.

PTF is a portable general-purpose trace format; in other words, it is a specification of a set of events that characterize application operations on persistent object stores. It is novel in the following respect: The semantics of the higher-level application is maintained through the trace events (e.g, the notion of an object is captured in the trace events). By maintaining the notion of an object, PTF traces can be used for a variety of purposes, including simulation studies, application visualizations, debugging, and statistical summaries of application behavior.

Typically, in the area of persistent object systems, instrumentation is accomplished at the system level, thus the trace events capture references and updates to *data* as in the MaStA trace format [39] as opposed to *objects.* At this level of instrumentation, it is more likely that platform dependencies are introduced into the trace. To avoid such dependencies, we have designed PTF so that it is devoid of system dependent information such as physical offsets and data sizes. Thus, PTF traces can be used across several platforms and by a variety of persistent object systems.

While PTF can be used independently of AMPS, using them together gives substantial leverage to a researcher interested in assessing persistent object systems, reducing both the time and effort required to create experiments. In general, the advantages of our approach include the following.

- Applications can be modeled independent of any particular persistent object system.
- The instrumentation necessary to perform experimentation and analysis is abstracted from the application layer.
- The effort of developing benchmark applications can be reduced through the reuse of classes provided as libraries.
- Trace event files can be generated once and then used in many different experiments by different experimentors.

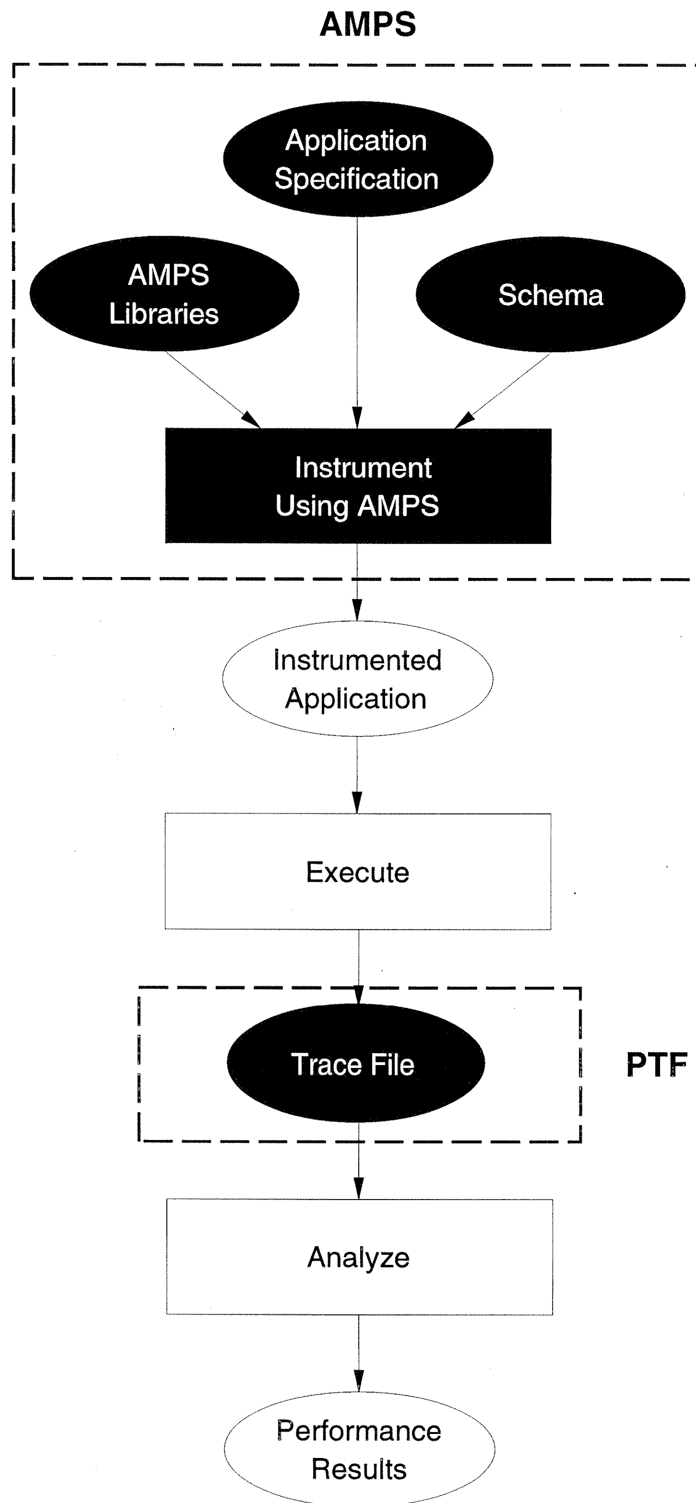


Figure 1.1: An Approach to Trace-driven Simulation for Persistent Object Systems Performance Analysis.

1.3 Evaluation

Prior work has shown that trace-driven simulation is a feasible technique for evaluating the performance of POS storage management algorithms [13, 15, 16]. In order to validate the work of this dissertation, a detailed case study of the OO7 benchmark specification as well as its multi-user extension was completed. The C++ implementation of the OO7 benchmark that is provided as free software by the Shore group at the University of Wisconsin was modified and instrumented using AMPS to generate single-user workloads. Traces were then generated independently of a persistent object system. These traces were compared with a hand-instrumented version of the OO7 benchmark implemented under Exodus.

The comparison showed that trace files generated from the C++ implementation instrumented with AMPS and that those generated from the hand-instrumented verion of the OO7 benchmark under Exodus contained trace events that were identical in order and content. Since the traces that were generated from a hand-instrumented version of the OO7 benchmark were verified through a simulator of a persistent object system to check their correctness, the comparison also showed that the AMPS instrumented version also correctly captured the structure and behavior of the OO7 benchmark.

A case study of the OO7 multi-user benchmark was also conducted to validate the feasibility of the instrumentation infrastructure that was designed and implemented to generate multi-user workloads. The OO7 multi-user application was instrumented using AMPS and then used to generate single-user workloads. These single-user workloads were then merged in a variety of ways to generate interesting multi-user workloads. Two experiments were performed. The first experiment showed that the infrastructure could be used to test aspects of a multi-user environment with respect to a given application such as contention between clients. The second experiment showed how the persistent store could be reorganized under the instrumentation infrastructure of this dissertation.

Through the OO7 multi-user case study, we were able to show that our instrumentation infrastructure was well suited for the OO7 multi-user benchmark. Furthermore, we were able to generate a variety of multi-user workloads without rerunning the OO7 multi-user application, thus providing a flexible framework for generating multi-user workloads.

1.4 Contributions and Benefits

This dissertation makes three major contributions: a POS-independent common trace format, the AMPS toolkit, and a new technique to generate multi-user workloads. Each of these contributions are discuss below.

1.4.1 The Common Trace Format

PTF captures the structure and manipulation of a persistent store by abstracting the logical behavior from the details of the physical implementation. Since physical attributes about the data of an object are not recorded in the trace events, the trace files generated using PTF are portable across several platforms. The portability of PTF makes it possible for researchers to interchange application workloads captured in trace files. This potential for sharing data could greatly reduce the effort in performing new experiments to evaluate the performance of components of persistent object systems. Researchers are also able to perform direct comparisons of POS implementations with PTF. Furthermore, PTF could be used to gain information about the behavior of a persistent

object application over some period of time in a manner similar to the way logs are used in database management systems.

The final benefit of the research on PTF is that the trace format provides a starting point toward the development of a standard trace format for the POS community of researchers. The need for a standard trace format has been discussed at the International Workshop on Persistent Object Systems (POS), but no unified action has been taken by the POS community of researchers to date.

In addition to the above benefits, PTF affords its users two advantages by maintaining the semantics of objects. These advantages are

- The objects recorded in the trace can be mapped to the physical layout of any persistent object system.
- Traces can be used to study the behavior of a persistent store at an application level (e.g., a visualization tool).

1.4.2 The AMPS Toolkit

In general, the implementation of an instrumented application is a time consuming process. With the AMPS toolkit, classes are provided as libraries that can be reused thus reducing both the time and effort to implement and instrument models of applications. By restricting all instrumentation functionality to access methods for data members of objects within an application, instrumentation is localized to specific methods and the process of instrumentation becomes less error prone. Lastly, by using the language `C++`, a language that is supported on many hardware platforms, and instrumenting at the application-level, the reusability of an instrumented application is greatly increased.

1.4.3 A New Technique to Generate Multi-user Workloads

The third contribution of this dissertation is an investigation of how to generate multi-user workloads in a flexible manner. Currently, the effects of concurrency are studied by either actually running more than one client on a prototype or artificially adding concurrency in the workload. In this dissertation, an approach for artificially introducing concurrency between single-user workloads to form a multi-user workload is investigated, designed, and implemented. This approach is novel in that single-user workloads are captured in traces that are generated by executing an application instrumented using AMPS and later merged under a concurrency control model and a transaction model to generate a multi-user workload. This approach provides several advantages. First, the single-user workloads can be combined a variety of ways, thus allowing for a broader range of experimentation with respect to multi-user workloads. Second, the process of combining single-user workloads does not require an re-running of the application to generate a multi-user workload. Third, the approach provides a framework for using alternative scheduling algorithms, transaction models, and concurrency control models to generate multi-user workloads.

1.5 Organization of Dissertation

A detailed discussion of PTF and the AMPS toolkit is provided in the remainder of the dissertation. The dissertation is organized as follows:

- Chapter 2 provides a more detailed discussion on the state of trace-driven simulation as a technique for performance evaluation in the area of storage management of persistent object systems. It also introduces the terminology that will be used in the remainder of the dissertation.

- Chapter 3 introduces PTF (POSSE Trace Format). It presents the goals and motivation behind the design of PTF. An overview of PTF is presented that includes an example illustrating the use of PTF in capturing a workload generated from a two-behavior application that builds a binary tree. The remainder of the chapter describes the semantics and structure of each of the trace events of PTF.

- Chapter 4 provides a detailed discussion of the AMPS toolkit. A brief overview of the architecture of AMPS is presented. Using a simple binary tree persistent object application, the process of modeling an application using the AMPS toolkit is then described. Once the application is implemented, a discussion of how to generate a single-user workload using the TCL front end is provided.

- Chapter 5 presents the details of the case study that was performed on the OO7 benchmark. It provides a brief overview of the OO7 benchmark. The remainder of the chapter presents experiences using the AMPS toolkit to model the persistent store and behavior of the OO7 benchmark. A comparison is then presented between trace files generated using the AMPS toolkit and trace files generated by executing a hand-instrumented version of the OO7 benchmark under Exodus.

- Chapter 6 describes the instrumentation infrastructure for generating multi-user workloads that was designed and implemented as part of this dissertation. The chapter begins by presenting the design goals of the infrastructure and the design issues that were encountered while designing this infrastructure. The infrastructure is then described. Enhancements to PTF and AMPS to support the implementation of the infrastructure are then presented. The chapter concludes with a description of an example implementation of the trace merger, the component of the infrastructure that merges single-user workloads to form a multi-user workload.

- Chapter 7 presents a discussion of the case study that was performed on the OO7 multi-user benchmark to show the feasibility of our approach to generating multi-user workloads. The chapter presents an overview of the OO7 multi-user benchmark specification. It provides a discussion of modifications that were made to the OO7 multi-user benchmark for the case study. It then presents an overview of two experiments: one to evaluate the contention among clients and the other which evaluated a reorganization function. The chapter concludes with observations about the instrumentation infrastructure that were made as a result of studying the OO7 multi-user benchmark.

- Chapter 8 reviews related work in the areas of trace-driven simulation and trace formats with respect to performance evaluation of persistent object systems. It then provides a discussion on how prior work relates to the work of this dissertation with respect to trace formats and application benchmarking and modeling.

- Chapter 9 summarizes the work completed in this dissertation and concludes with a discussion of future research directions.

Chapter 2

Background

For years, trace-driven simulation has been a popular approach for evaluating the performance of proposed cache and paging designs and has proven to be a cost-effective method for estimating the performance of primary-memory system designs. As a result of the effectiveness of the technique, many trace-driven simulation tools have been developed. In a recent survey, Uhlig and Mudge [44] compared over 50 trace-driven simulation tools as part of an effort to formulate criteria for evaluating trace-driven methods.

Trace-driven simulation has been used for a wide variety of purposes, including the evaluation of dynamic storage management implementations. For example, in the early 90s, Zorn [47] and Wilson [45] used trace-driven simulations to study the performance impact of garbage collection on caches. The success of this approach in the domain of primary memory led Cook et al. [13, 15, 16], the POSSE (Persistent Object SyStems Evaluation) group at the University of Colorado, to apply the approach to the related study of performance of the storage manager of persistent object system implementations.

Through trace-driven simulations, Cook et al. investigated methods to improve the performance of algorithms for automatic storage reclamation, focusing on policies to effectively select partitions to collect and the rate at which to perform the collection. Others have also used this approach. Specifically, Scheuerl et al. [40] used event traces to analyze the I/O performance of various recovery mechanisms.

In this chapter, we present an overview of the use of trace-driven simulation as a technique to evaluate the performance of the storage manager of a persistent object system. Before providing this overview, some basic terminology that will be used throughout the remainder of the dissertation is presented.

2.1 Basic Terminology

There are few terms that are used throughout the dissertation. Below, definitions are provided for these terms.

- **Application.** An application is a specification/program that uses a persistent object system for a specific purpose. An application consists of a schema and a behavior.
- **Schema.** A schema is a description of the data manipulated by an application. This description includes the legal operations that can be performed on the data (i.e., the data are abstract data types).

- **Task.** A task is a legal operation or a specific combination of legal operations that can be performed on the data defined by the schema. Examples of tasks are: to create an initial object store, to delete every other object, and to search for all objects satisfying a given predicate.

- **Behavior.** A behavior is a set of tasks that can be performed by an application.

- **Object Store.** An object store is an instantiation of a schema. In other words, it is a container of data. In some instances the object store is a simulated container of data and therefore it is just a representation of data rather than the actual data themselves.

- **Workload.** A workload is a specific combination of tasks that is performed by an application on a specific object store. The tasks of the workload can be instrumented to produce trace events that capture the manipulations of a persistent store during the execution of the application.

- **Multi-user Workload.** A multi-user workload is the combined workload of multiple clients or users concurrently accessing a persistent store during the execution of an application.

- **Trace.** A trace is a record of the effect of simulating or executing a workload.

- **Logical Workload Trace.** A logical workload trace is a trace containing trace events that capture information about the manipulation of a persistent store that is independent of the physical characteristics of either the in-memory representation of the objects of the persistent store or the on-disk representation of these objects by the persistent object system.

- **Physical Workload Trace.** A physical workload trace is a trace that consists of information specific to a particular persistent object system implementation. Within the physical workload trace, the trace events are augmented with information such as the size of an object, numeric offsets of data within the object, and the physical location of the object on the disk.

- **Synthetic Application.** A synthetic application is an application that is developed for the purpose of exercising and/or studying a persistent object system. A synthetic application can, of course, be derived from a real-world application through some sort of modeling activity.

- **Benchmark.** According to the Encylopedia of Computer Science [35], a benchmark is a standardized computer program for which there is a history of measurement data (typically timings) for execution of the programs with specifically defined input and reproducible output. In the context of this dissertation, a benchmark is a *standard* in the sense that it has been, or is proposed to be, adopted by the community of persistent object systems researchers. Evidence of acceptance is its use by several people. Furthermore, for our purposes, the benchmark consists of an application, schema, and a set of one or more workloads.

2.2 Overview of Trace-Driven Simulation in Evaluation of Persistent Object Systems

As stated above, Cook et al. effectively used trace-driven simulation to perform analysis of garbage collection policies. In the first generation of experimentation, Cook et al. built a synthetic application whose persistent store consisted of a forest of binary trees in which some trees had additional edges. The application simulated a single process that probabilistically created, accessed, and

deleted objects of the persistent store. The application made direct procedure calls to the simulation system, ODBsim [14], that was built to simulate a persistent object storage manager. Using this application, Cook et al. [16] investigated different partition selection policies for garbage collection, resulting in the design of a policy that performed better than any existing implementable policy with respect to collecting more garbage with less I/O. As a result of their experiences, they decided to investigate a way to make the experimental input (not just output) available to other researchers. To accomplish this goal, a general-purpose trace format, PTFF (POSSE Trace File Format), was developed. The set of trace events of the PTFF specification capture the effect of tasks of an application on a persistent store (e.g., object creations, accesses, and modifications).

PTFF was then used in instrumenting applications, thus forming the second generation of experimentation with a trace-driven approach to performance evaluation of persistent object systems. In the second generation, Cook et al. hand instrumented an implementation of an application to collect application traces as illustrated in Figure 2.1. In this approach, the application is implemented in a language supported by a given persistent object system. To perform hand instrumentation correctly, the source code of the application must be completely understood to identify the appropriate places to insert instrumentation functions. During the process of hand instrumentation, if a single event is overlooked, the state recorded in the trace does not reflect the state of the persistent store created by the application. Once the application is hand instrumented, it is then executed using an actual persistent object system. The execution results in the creation of a persistent store and traces that capture the persistent store as well as the behavior of the application. The traces are then used as input to an analyzer. In this case, the analyzer is a simulator of a persistent object system.

Using this approach, Cook et al. [13] investigated the impact of garbage collection rate on application behavior. They developed two semi-automatic, self-adaptive policies to control the garbage collection rates. For this set of experiments, the researchers hand instrumented an E [21] implementation of the OO7 benchmark [10] to generate several trace files as the application executed under Exodus. In order to assess correctness of the hand instrumention of the application, Cook et al. created very small stores by altering the parameters of the benchmark. The traces were then submitted to the simulator ODBsim as a means of verification. For example, the simulator was used to check the connectivity of the store that was generated from the trace. In order to get the instrumentation correct, several tests were run. Cook et al. found that the process required a significant amount of manual labor and the hand instrumentation was error prone. Furthermore, the instrumented application was not reusable with respect to other persistent object systems.

Although hand instrumentation proved to be problematic, Cook et al. found that the generation of the traces proved invaluable to their experimentation process. They were easily able to organize and document their experiments. They had more control over the running (and rerunning) of the same experiments multiple times (e.g., in the face of resource limitations or simulator errors). They were also able to gather statistical information about the various workloads.

The work of Cook et al. formed the foundation for the work completed in this dissertation. Specifically, PTF was based on the specification of PTFF. Also, their experience with hand instrumentation lead to the conceptualization of AMPS, which allows applications to be instrumented more easily and flexibly.

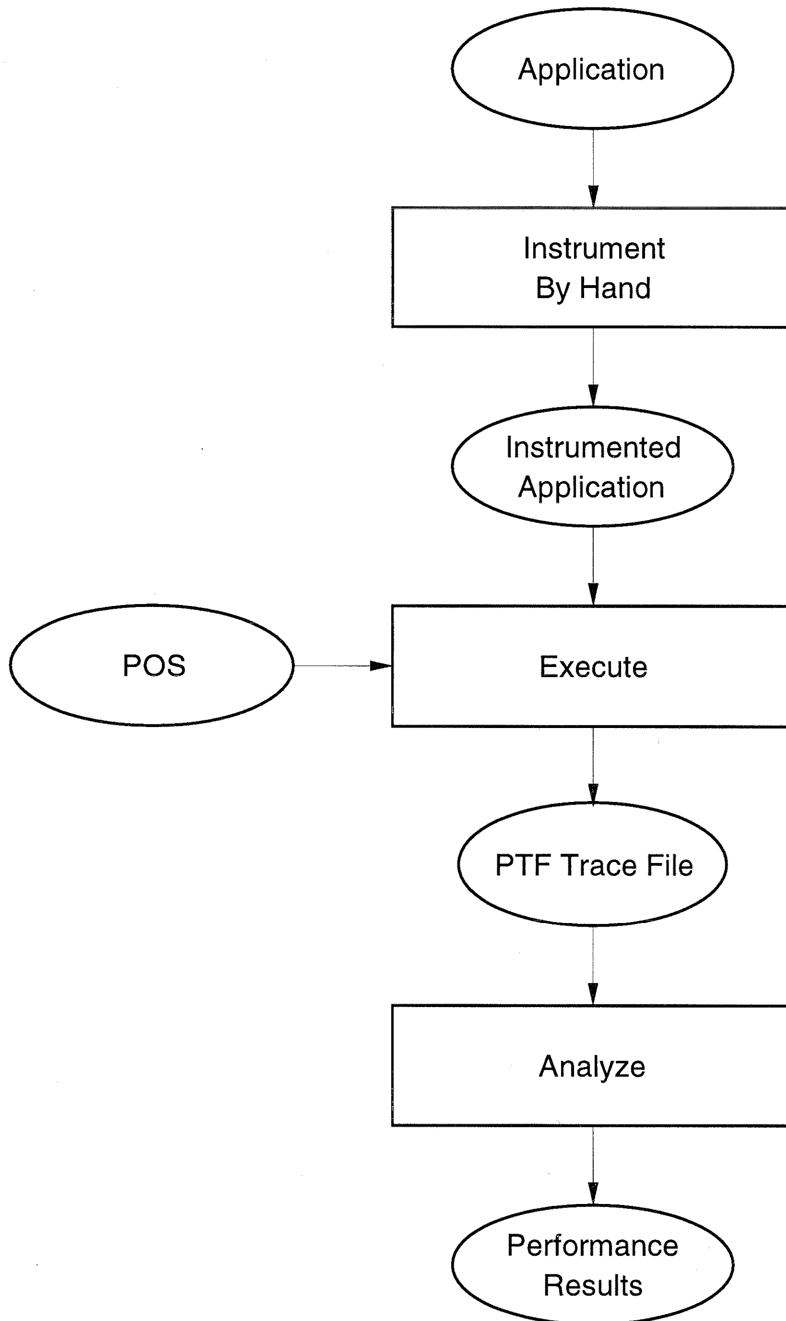


Figure 2.1: An Approach to Application Instrumentation.

Chapter 3

Trace Format

The POSSE Trace Format (PTF) is a first effort toward the development of a common trace format. It is novel in that the trace format characterizes the structure of an object store and the time-varying behavior of an application as manipulations of persistent objects during the execution of the application. For simplicity, we always assume that the object store is initially empty, and that the application begins by populating the store.

During the execution of an application, objects manipulated by the application fall within one of the following categories:

- Transient objects.
- Evolving objects which are newly created objects that have not been attached to a persistent root or have not been explicitly made persistent.
- Objects that are already persistent.

Of these objects, PTF captures the creation and manipulation of *only* the persistent objects that are generated by the application layer.

With PTF, we are able to conduct a *direct performance comparison*, by which we mean that the performance of two implementations can be compared based on a single trace. As much as we can, our intent is to capture the logical workload in PTF events abstracting away details of a particular persistent object system implementation that would appear in a physical workload. For example, while events in the logical workload carry information such as object type and symbolic offsets to object fields, the physical workload augments this information with information about object size, numeric offsets, and the physical location of the object on the disk. This goal is similar in spirit to the design of the Java Virtual Machine [29], which also abstracts away physical information in its representation.

PTF has been used to capture the structure of simple persistent stores. It was also used to capture the structure of the persistent store as described in the schema specification of the OO7 benchmark [10] as well as the behavior of the tasks that make up single-user workloads of the OO7 benchmark. PTF traces can be used for a variety of purposes, such as simulation studies of storage management in persistent object systems, application visualizations, debugging, and statistical summaries of application behavior.

This chapter describes PTF. We present the design goals in the first section. We then provide an overview of the PTF design followed by a description of the syntax and semantics of each trace event. A description of the binary and ASCII representations is then provided. The chapter then concludes with a summary of PTF.

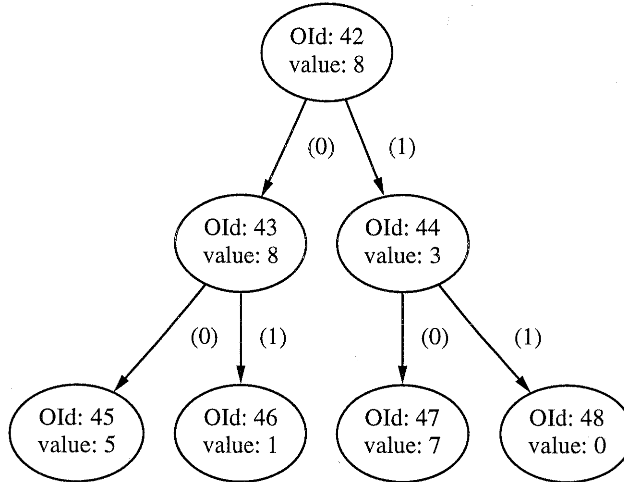


Figure 3.1: A Simple Persistent Store Organized as a Binary Tree.

3.1 Design Goals

The initial design goals of PTF are as follows:

- To develop a trace format that can be used to study and evaluate performance issues of persistent object systems, such as storage management, as well as other analysis activities including comparative experimentation of implementations of persistent object systems.
- To develop a system-independent representation of a workload.
- To promote the creation of a collection of representative application behaviors in a common trace format.

3.2 Overview of PTF Design

PTF captures the persistent objects of a persistent object application and their relationships to other persistent objects within the application as a directed graph. The persistent objects form the nodes of the graph and the relationships between objects are edges of the graph. The lifetime of a persistent object is from the point that the object becomes persistent to the point at which the object is either implicitly deleted by a garbage collector after disconnection from the graph or explicitly deleted. Within PTF, each object is represented by a logical object identifier (OId). To illustrate this view of a persistent store, we present a simple example of a persistent store organized as a binary tree. Figure 3.1 depicts a state of the binary tree persistent store, in which each node contains a data value and pointers to left (offset 0) and right (offset 1) subtrees. The logical OIds, used in the PTF trace to identify each object, are also shown.

PTF uses the logical object identifier to maintain independence from physical address implementations. At the creation of an instance of a class, an event is generated to represent the creation of an object and the assignment of an OId to the object. From that point on, any reference to the object is made through the assigned OId. Within an application, an OId is never reused.

Table 3.1: PTF Trace Events.

Category	Event Name	Abbreviation	Arguments
Format Object	`format object`	`fo`	super class format OId, number of pointers, length of object name, number of data attributes, number of array data attributes, list of format OIds, list of array format OIds and number of elements for each array, name of format
Object	`create object`	`co`	format OId, OId
	`create array object`	`cao`	format OId, OId, container OId, number of elements
	`delete object`	`do`	format OId, OId
	`set root`	`sr`	format OId, OId
	`get root`	`gr`	
Atomic Data	`data read`	`dr`	format OId, OId, offset
	`data write`	`dw`	format OId, OId, offset
Array Data	`array data read`	`adr`	format OId, OId, offset, number of indexes, index
	`array data write`	`adw`	format OId, OId, offset, number of indexes, index
Connections	`edge read`	`er`	format OId, OId, offset
	`edge write`	`ew`	format OId, from OId, offset, to OId
Directives	`begin no collection`	`ngs`	
	`end no collection`	`nge`	

PTF contains events that reflect operations to create, delete, access, and modify persistent objects. Table 3.1 outlines the events in PTF, placing them into six categories. We model the data in an object (but not the values of those data) and the pointer connections between objects. The manipulation of the data of an object is represented using the events `data read`, `data write`, `array data read`, and `array data write`, where each event indicates that a single value or a range of values has been read or written. Although we do not describe nor illustrate this here, actual data values optionally can be recorded in the trace as annotations on the events. We model manipulations of the pointer connections between objects with the `edge read` and `edge write` events. Each edge is referred to by its unique offset within the object, and edges are numbered starting from zero.

The events `create object`, `delete object`, and `set root` determine the lifetime of objects that can be accessed by an application. The event `create object` additionally records information about the format of the object created, specifically the OId of a "format" object. The format object describes the fields of an object in terms of their formats and their relative positions in the representation of the object. The event `format object` records this information.

Dynamic data are treated as a separate object and thus the allocation of dynamic data are captured through the event `create array object`. From the perspective of the containing object, the dynamic data is considered in the same manner as a pointer. The data contained in the dynamic allocation is manipulated using the events `array data read` and `array data write`. The array object cannot contain pointer values. More details on how to capture dynamic data structures containing pointers are provided in the detailed discussion of the event `create array object`.

Our persistence model uses the mechanism of persistence by reachability [3]. The event `set root` indicates that the object contains the root set to be used in the reachability analysis. This object is referred to as the super root object and is attached to the root that is maintained

by the persistent object system. At the application level, any number of objects can be designated as roots and are contained in the root set. These roots are captured through manipulations of the super root object. The event `get root` captures a reference by the application to the super root object.

It is important to understand that PTF does not enforce any notion of access consistency. Nor does it require any particular storage reclamation scheme, namely manual versus automatic storage reclamation. Clearly, the operation `delete object` leaves an application vulnerable to such inconsistencies. But we assume that applications will be written to behave "properly", respecting access consistency and, therefore, also respecting persistence by reachability.

Explicit deletion of objects is only one approach to persistent storage reclamation. Automatic garbage collection is an alternative that does not require the use of the event `delete object`. On the other hand, automatic garbage collection requires careful control over when the garbage collector can operate. The events `begin no collection` and `end no collection` are necessary to identify atomic sequences of operations with respect to the creation of new objects. In particular, the garbage collector must be prevented from running between the time a new object is created (signified by the event `create object`) and the time that the new object becomes reachable from the persistent root (signified by the event `write edge` or the event `set root`). We note that the `begin no collection` and `end no collection` events provide a very weak form of a transaction.

Before proceeding with a detailed description of each trace event, we now present an example to illustrate the use of PTF in capturing a workload. Figure 3.2 contains the PTF trace for a simple two-behavior application that first builds the binary tree of Figure 3.1 and then sums the values contained in the nodes. (The text to the right of each event is not part of the trace, but only an annotation added by hand to aid the reader's understanding of the figure.)

The first behavior, bracketed by the protective events `ngs` and `nge`, creates the objects in the store and then links them together using a combination of events `co` and `ew`. The writing of data is represented by the event `dw`. After the persistent store is created, the second behavior of the application traverses the tree in a breadth-first manner, accessing the data value at each node. To reduce the complexity of the example, we assume that the application knows the depth of the tree and, hence, does not need to read the edges at the leaves.

3.3 Trace Events

This section describes the overall format of the trace event stream, as well as each trace event in detail. The semantics of every event is described. The description of an event is independent of the representation format of the event (i.e., either a binary format or an ASCII format). Representation formats are described separately in Section 3.4.

3.3.1 Trace Format

The structure of PTF traces is described by the following grammar:

```
trace_file     := <begin> <events> <end>
begin          := "Trace begin\n"
end            := "Trace end\n"
events         := <event> "\n" <events> | <null>
event          := <event_id> <core_event>
event_id       := <integer>
```

```
Trace begin
fo 41 0 2 1 0 11 BinTreeNode Format object for the BinTreeNode object
ngs                          Disallow garbage collection until after Te event
co 41 42                     Create object with OID 42 whose format OID is 41
dw 41 42 1                   Write data value to position 1 in object 42 of format 41
sr 41 42                     Set object 42 of format 41 to be a persistent root
co 41 43                     Create object with OID 43 whose format OID is 41
dw 41 43 1                   Write data value to position 1 in object 43 of format 41
ew 41 42 0 43                Write edge 0 from object 42 of format 41 to object 43
co 41 44                     Create object with OID 44 whose format OID is 41
dw 41 44 1                   Write data value to position 1 in object 44 of format 41
ew 41 42 1 44                Write edge 1 from object 10 of format 41 to object 44
co 41 45                     Create object with OID 45 whose format OID is 41
dw 41 45 1                   Write data value to position 1 in object 45 of format 41
ew 41 43 0 45                Write edge 0 from object 43 of format 41 to object 45
co 41 46                     Create object with OID 46 whose format OID is 41
dw 41 46 1                   Write data value to position 1 in object 46 of format 41
ew 41 43 1 46                Write edge 1 from object 43 of format 41 to object 14
co 41 47                     Create object with OID 47 whose format OID is 41
dw 41 47 1                   Write data value to position 1 in object 47 of format 41
ew 41 44 0 47                Write edge 0 from object 44 of format 41 to 47
co 41 48                     Create object with OID 48 whose format OID is 41
dw 41 48 1                   Write data value to position 1 in object 48 of format 41
ew 41 44 1 48                Write edge 1 from object 44 of format 41 to object 48
nge                          Allow garbage collection to occur
dr 41 42 1                   Read data value from position 1 in object 42 of format 41
er 41 42 0                   Read value of edge 0 from object 42 of format 41
er 41 42 1                   Read value of edge 1 from object 42 of format 41
dr 41 43 1                   Read data value from position 1 in object 43 of format 41
er 41 43 0                   Read value of edge 0 from object 43 of format 41
er 41 43 1                   Read value of edge 1 from object 43 of format 41
dr 41 44 1                   Read data value from position 1 in object 44 of format 41
er 41 44 0                   Read value of edge 0 from object 44 of format 41
er 41 44 1                   Read value of edge 1 from object 44 of format 41
dr 41 45 1                   Read data value from position 1 in object 45 of format 41
dr 41 46 1                   Read data value from position 1 in object 46 of format 41
dr 41 47 1                   Read data value from position 1 in object 47 of format 41
dr 41 48 1                   Read data value from position 1 in object 48 of format 41
Trace end
```

Figure 3.2: Annotated PTF Trace Generated from a Binary Tree Application.

where <integer> and <null> have the usual meaning.
The general specification of an event is as follows:

```
core_event      := <ev_type> <ev_parameters>
ev_type         := <identifier>
identifier      := <lower_case_char> | <lower_case_char> <identifier>
string          := <non_digit> | <string> <non_digit> | <string> <digit>
ev_parameters   := <integer> <ev_parameters> | <string> <ev_parameters> | <null>
non_digit       := _ | <lower_case_char> | <upper_case_char>
lower_case_char := a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
                   p | q | r | s | u | v | w | x | y | z
upper_case_char := A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
                   P | Q | R | S | U | V | W | X | Y | Z
digit           := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

An event always starts with an identifier that indicates its type, followed by an arbitrary number of parameters, which are not specified by the above grammar. Each event type has a specific number of parameters. In addition to the number of parameters, the order, size, and semantics of each parameter are also described with the individual event. Below, the semantics of each event are described in a uniform manner.

3.3.2 Events that Manipulate Objects of the Persistent Store

Format Object

Event:	<format_object>		
Parameters:	(int)	FormatId	format OId
	(int)	SuperFormatId	format Id for super object
	(int)	NumberOfPointers	number of pointer values
	(int)	NumberOfDataMembers	number of data members
	(int)	NumberOfArrayMembers	number of array members
	(int)	LengthOfClassName	length in bytes of name
	(int)	DataMemberFormatId	format Id of data members
	(int)	ArrayDataMemberFormatId	format Id of array members
	(int)	NumberOfElements	number of elements in arrays
	(string)	NameOfClass	name associated with class

Associated with each object is a format object. The format object specifies a layout for each of the class specifications in a schema. The layout does not capture the physical representation in the same manner as a class specification nor does it capture the object format of a particular object store. Its primary use is to capture the relevant information about an object such as the number of pointer values and the format and the position of the data values of an object. This information can then be used to adapt the PTF trace files to a specific object format of a particular POS. There must be an object format for each persistent class in the persistent object application. An object must be created for each of these object formats prior to the execution of behaviors of an application.

Primitive Format Object	Object Identifier
char	10
int	11
short	12
long	13
unsigned	14
unsigned char	15
unsigned long	16
float	17
double	18
long double	19
array of char	30
array of int	31
array of short	32
array of long	33
array of unsigned	34
array of unsigned char	35
array of unsigned long	36
array of float	37
array of double	38
array of long double	39

Table 3.2: Object Identifiers of Primitive Format Objects

The format object trace event has ten parameters. The parameter `FormatId` contains the object identifier for the format object represented by the format object trace event. PTF supports single inheritance so the format contains a parameter to indicate the format of the super object `SuperFormatId`. This field contains a zero value if the object does not inherit its format from another object. The number of pointer values of an object is provided by the parameter `NumberOfPointers`. The parameter `NumberOfDataMembers` indicates the number of all the data attributes that are primitive object formats (e.g. integers, floating points) excluding fixed sized array object formats. Parameter `DataMemberFormatId` is a list of format object identifiers for the data attributes. There are zero or more format object identifiers in the category of the parameter `DataMemberFormatId` depending on the value of the parameter `NumberOfDataMembers`.

In the case of fixed sized arrays, the parameter `NumberOfArrayMembers` indicates the number of data members that are arrays. Fixed sized arrays are specified using two parameters, `ArrayDataMemberFormatId` and `NumberOfElements`. The format object identifier for a fixed sized array is stored in the parameter `ArrayDataMemberFormatId`. The number of elements of the array is stored in the parameter `NumberOfElements`. All arrays, fixed sized or dynamic, are treated as one dimensional arrays and therefore, multi-dimensional arrays must be converted to their one dimensional equivalent.

The format object identifiers for the primitive format objects are shown in Table 3.2. Object identifier values from 10 to 40 have been reserved for the primitive format objects.

The object format augments the other trace event types during the processing of the trace files. The size of the data portion of objects can be calculated using the parameters NumberOfDataMembers, DataMemberFormatId, ArrayDataMemberFormatId, and NumberOfElements.

To illustrate how to translate a class type specification to an object format event, let us look

at a portion of the `CompositePart` C++ class specification of the OO7 benchmark shown in Figure 3.3. Figure 3.3 shows the application view of the composite part object using C++, the PTF layout format, and a possible object format of a persistent object system. We begin by translating the the super class `DesignObject` into a format object event. A logical object identifier is assigned to the format object and recorded in the second field of the trace event format. Since the DesignObject does not have a super class, the second parameter `SuperFormatId` takes on the value of a 0 to indicate that it is NULL. The DesignObject class contains no pointers; thus, the parameter `NumberOfPointers` contains the value 0. The parameter `NumberOfDataMembers` contains the value 2 and the parameter `DataMemberFormatId` will be repeated twice containing two integer format object identifiers. The parameter `NumberOfArrayMembers` contains the value 1. `ArrayDataMemberFormatId` contains the logical object identifier for an array of characters, which follows the data member format object identifiers in the format object layout, and the parameter `NumberOfElements` contains the value 10. The parameter `NameOfClass` will contain the string "DesignObject" and the parameter `LengthOfClassName` will contain the length of this string. A logical object identifier is then assigned to the format object representing the `CompositePart` class. The parameter `SuperFormatId` contains the logical object identifier of the DesignObject format object. Since the CompositePart class contains five pointers, the parameter `NumberOfPointers`, which is the fourth field in the trace event format, contains a 5. The parameter `NumberOfDataMembers` contains the value 0 and the parameter `NameOfClass` contains the value "CompositePart" with its length recorded in the seventh field of the format.

Create Object

Event:	`<create_object>`		
Parameters:	(int)	FormatId	format OId
	(int)	OId	logical object Id to be created

Creates a new object with logical object identifier `OId`. Using the `FormatId`, the format of the created object can be used to calculate the size of the object and the number of pointers can be ascertained. This information may be required by the POS back end.

The total object size depends on the storage requirements for the POS backend and consists of the data size of the object and the size for the object's out-edges or pointers. The number of out-edges can be taken as a hint by the POS. A POS that is capable of dynamically adjusting the number of out-edges of an object does not need to reserve enough space to hold all edges but can add them dynamically as they are written.

`OId` 0 is reserved to represent the `NULL` object. All out-edges without a specific initial value point to the `NULL` object.

Create Array Object

Event:	`<create_array_object>`		
Parameters:	(int)	FormatId	format OId
	(int)	OId	logical object Id to be created
	(int)	ContainerOId	logical object Id of containing object
	(int)	NoOfElements	number of elements

C++ Class

```
class DesignObject {
  public:
      int id;
      char type[TypeSize];
      int buildDate;
  ...
};

class CompositePart: public DesignObject {
  public:
      class Document *documentation;
      class Assoc *parts;
      class AtomicPart *rootPart;
       // list of assemblies in which part is used
       // as a private component
      Assoc *usedInPriv;
       // list of assemblies in which part is used
       // as a shared component
       Assoc *usedInShar;
    ...
 };
```

fo	OId (DO)	0	0	2	1	12

OId (int)	OId (int)	OId (array of char)	10	DesignObject

PTF Object Format

fo	OId (CP)	OId (DO)	5	0	0	13	CompositePart

Object Format of the Persistent Store

5 (number of pointers)
TotalSize
OId (documentation)
OId (parts)
OId (rootPart)
OId (usedInPriv)
OId (usedInShar)
int (id)
10 chars (type)
int (buildDate)

Figure 3.3: Three Levels of Object Layout Descriptions.

Since some languages support the creation of arrays both statically and dynamically, PTF handles both fixed size and dynamically allocated arrays. This event is used to capture the allocation of dynamic data. The dynamic data structure is treated as a separate object and is given a logical object identifier. The number of elements of the array is specified through the `NoOfElements` parameter. The parameter `FormatId` indicates the format object identifier of the elements of the array object. The `ContainerOId` links the array object to its containing object and maintains the object's identity as a data member of the containing object. There can be only one containing object per array object.

The `create array object` event must be preceded by a `create object` event for the containing object. The forward link from the containing object to the array object is captured through an `edge write` event.

Deletion of an array object either occurs when the array object is explicitly deleted through a delete operation or implicitly deleted using garbage collection. If garbage collection is in effect, the array object or its containing object may become unreachable from a persistent root, thus making the array object eligible for garbage collection. In cases where a reference to either the containing object or the array object is overwritten, an `edge write` event should appear in the trace event stream to capture the overwriting of the reference.

With explicit deletion, a request to delete the containing object may have occurred and thus the array object is deleted prior to deleting the containing object. If a request is made to delete the array object only, an `edge write` event that captures the overwriting of the reference to the array object must occur prior to the `delete object` event that captures the deletion of the array object.

This event should not be used to capture the creation of a dynamic structure that contains pointers. It is important to be able to capture the overwriting of pointer values and thus a dynamic array of pointers should be captured by first using the event `create object` and then using the events `edge write` and `edge read` to capture the writing and reading of pointer elements.

PTF does not contain an event type to represent the resize operation. The effect of such an operation can be obtained by using the create array object event and adjusting the value of the `NoOfElements` parameter.

Delete Object

Event:	`<delete_object>`		
Parameters:	(int)	FormatId	format OId
	(int)	OId	logical object Id to be deleted

Explicitly deletes the existing object associated with the logical object identifier specified by parameter `OId`. Deleting a non existent object is an error. Depending on the semantics of the delete operation of the persistent object system, the object might actually be deleted, marked as invalid, or marked as garbage to be collected. After deleting an object no data or meta-data of the object should be read or written.

Edge Write

Event:	`<edge_write>`		
Parameters:	(int)	FormatId	format OId
	(int)	FromOId	logical object Id of from-object
	(int)	ToOId	logical object Id of to-object
	(int)	Edge	number of edge to be written

Changes edge `Edge` of object `OId` to reference object `ToOId`. Any previously existing reference of this edge is automatically overwritten. `ToOId` must be an existing object or the null object.

Edge Read

Event:	`<edge_read>`		
Parameters:	(int)	FormatId	format OId
	(int)	FromOId	logical object Id of from-object
	(int)	Edge	number of edge to be read

Queries the reference of edge `Edge` of object `OId`. If this edge has not been written prior to the read, then the resulting value is the null object; otherwise, it is the `ToOId` most recently written into this edge.

Data Write

Event:	`<data_write>`		
Parameters:	(int)	FormatId	format OId
	(int)	OId	logical object Id to write to
	(int)	Offset	position of the attribute within format object

Writes a number of bytes of data starting from the physical offset of the given attribute, determined by the `FormatId` and `Offset` parameters of the event. The `Offset` parameter contains the position of the data member with respect to all of the data members associated with the object. The number of bytes to be written is determined using the format object identifier of the attribute located at the position indicated by the `Offset` parameter. The format object identifier is associated to a format object which has been assigned a specific number of bytes corresponding to the requirements of the hardware platform on which the persistent object system executes.

The calculated physical offset to start writing must be within a legal range. The actual data written is not contained in the event since it is not part of the structural information needed to reproduce the behavior of the persistent store. However, it is assumed that the persistent object system writes to (or simulates a write operation on) the specified region of the object.

Data Read

Event:	<data_read>		
Parameters:	(int)	FormatId	format OId
	(int)	OId	logical object Id to read from
	(int)	Offset	position of the attribute within format object

The parameters `FormatId` and `Offset` determine the starting offset of the object addressed by `OId` to begin reading and the number of bytes to read. The data block read must be contained completely within the data block defined at object creation time. Similar to data writes, the actual data that is read is not included.

Array Data Write

Event:	<array_data_write>		
Parameters:	(int)	FormatId	format OId
	(int)	OId	logical object Id to write to
	(int)	Offset	position in containing object
	(int)	Index	the index into the array
	(int)	Length	number of elements

The `array data write` event captures writing of data for both fixed arrays and dynamically allocated arrays. The parameter `Index` indicates which index within the array to start writing data. The parameter `Length` specifies the number of elements to be written. Using the `Index` and the `Length` parameters, a range can be specified, beginning at the offset calculated using `Index` and ending at the offset calculated using `Index` + `Length`.

For dynamic arrays, the parameter `Offset` contains a negative one and the parameter `OId` is the logical object identifier of the array object. The offset to begin writing is calculated using the parameter `Index` and the number of bytes per element, which can be obtained from the parameter `FormatId`. Using the number of bytes per element along with the parameter `Length`, the total number of bytes to be written can be calculated.

In the case of a fixed array, the parameter `OId` refers to the logical object identifier of the containing object and the parameter `FormatId` contains the format object identifier for the fixed array object format. The starting offset is calculated using the `Offset` parameter to obtain the position within the containing object and the `Index` parameter. The number of bytes per element to be written is obtained from the format object. As with the dynamic array, the number of bytes per element and the parameter `Length` are used to calculate the total number of bytes to be written. The actual data written is not included.

Array Data Read

Event:	<array_data_read>		
Parameters:	(int)	FormatId	format OId

(int)	OId	logical object Id to read from
(int)	Offset	position in containing object
(int)	Index	the index into the array
(int)	Length	number of elements

As with the array data write event, the **array data read** event reads data of both fixed arrays and dynamically allocated arrays. The parameter `Index` indicates the first index to begin reading data. The parameter `Length` specifies the number of elements to be read. Using the `Index` and the `Length` parameters, a range can be specified, beginning at the offset calculated using `Index` and ending at the offset calculated using `Index` + `Length`.

For dynamic arrays, the parameter `Offset` contains a negative one and the parameter `OId` is the object identifier of the array object. The offset to begin reading is calculated using the parameter `Index` and the number of bytes per element, which can be obtained from the parameter `FormatId`. Using the number of bytes per element along with the parameter `Length`, the total number of bytes to be read can be calculated.

In the case of a fixed array, the parameter `OId` refers to the logical object identifier of the containing object and the parameter `FormatId` contains the format object identifier for the fixed array object format. The starting offset is calculated using the `Offset` parameter to obtain the position within the containing object and the `Index` parameter. The number of bytes per element to be read is ascertained from the format object. As with the dynamic array, the number of bytes per element and the parameter `Length` are used to calculate the total number of bytes to be read. Similar to array data writes, the actual data that is read is not of interest, but it is assumed that the persistent object system performs a read of the specified region or simulates such a read.

Set Root

Event:	`<set_root>`		
Parameters:	(int)	FormatId	format OId
Parameters:	(int)	OId	logical object Id to move to root set

This event has one parameter, the OId of the super root object, which contains the root set of the persistent store. It can be thought of as the starting point of the store. The objects of the root set indicate which objects are persistent through reachability.

Get Root

Event: `<get_root>`

This event captures a reference to the super root object.

3.3.3 Garbage Collection Directives

No Garbage Collection Start

Event: `<noGC_start>`
Parameters: none

The `no garbage collection start` event serves as an indicator to the POS to prevent the garbage collector from executing until a `no garbage collection end` event has been reached. These events bracket trace events that are representing the creating of objects or the updating of references within objects.

No Garbage Collection End

Event: `<noGC_end>`
Parameters: none

The `no garbage collection end` event serves as an indicator to the POS to allow the garbage collector to run as necessary.

3.4 Representation Formats

The PTF implementation allows the creation of trace files in either binary format or ASCII format. The binary format was developed to reduce the size of trace files and increase the efficiency of processing the trace files. The ASCII format is provided to allow manual inspection of the trace files by a developer. Below, we briefly describe each representation format.

3.4.1 Binary Format

The binary format of PTF requires that the trace begin with a header followed by the trace events. The header is in ASCII format and consists of a version number on the first line of the header, followed by several lines of user notes. The separator `$$binary$$` is used to end the header and must appear on a separate line. Each binary event consists of n + *1* bytes where n is determined by the event type. The first byte of each event represents its type. The format and number of bytes for each event type are provided in the Trace.h file, which can be found in the appendix.

3.4.2 ASCII Format

The ASCII format consists of one event per line with the line ending in a carriage return. Each event begins with an event identifier followed by the parameters of the event. A sample trace in ASCII format can be seen in Section 3.2.

3.5 Summary

A large variety of trace formats have been developed to capture information about the behavior of applications in various areas of computer systems design and evaluation. Trace format designers are primarily concerned with the following issues:

- *Trace compactness.* Often traces represent literally billions of operations, and as such, their physical size can be of great concern if one needs to store and distribute them. Studies have shown that data-specific compression techniques (e.g., for compressing program address traces [38]) have significant advantages over standard text compression algorithms. We do not anticipate generating traces as large as address traces get, and so expect traditional compression to be sufficient for our purposes.

- *Trace usability.* Usability is directly related to how much information the trace contains, and how easy that information is to manipulate. Including extra information in a trace can make it more usable, but at the same time also increases its size. Our initial goal for the design of PTF has been to ensure that it supplies all the information necessary for our storage management performance studies. Additionally, information has been added to enhance trace processing performance. For example, we include the logical object identifier of the object format in each of the trace events that capture the manipulation of an object. The resulting redundancy only slightly increases the size of the trace files. In fact, we observed only a 10% increase in compressed file size over a trace without the format object information. The advantage of including the object layout identifier with each event is that tools processing the trace do not have to always look up the format of each object, thus increasing the speed of trace processing. We feel that this is a reasonable trade off between space and time.

- *Trace accuracy.* Accuracy reflects how effectively the information contained in the trace captures the data necessary to evaluate system performance. For example, traces are often truncated because a full trace requires too much computation to process. Likewise, approximations may be made in the workload to simplify the generation of a trace. Our current goal with respect to accuracy is to provide a completely accurate single-user trace; our current work with multi-user traces requires that we make approximations that reduce the trace accuracy.

PTF is implemented in C++ and includes converters to translate a binary trace file to an ASCII trace file and vice-versa. In the case of the ASCII format, it also contains a verifier to check the correctness of an event type with respect to the number and type of each of its parameters.

The current version of PTF does not support dynamic resizing, embedded classes, multiple inheritance, and multiple versions of the object layouts. Furthermore, some objects and data structures are part of the implementation of the persistent object system (e.g., indexes, extents) and not a part of the application. These objects and their behaviors need to be represented in the PTF trace. Extents are captured by creating data structures to represent them in the application and then instrumenting the operations applied to these structures. Currently, although we realize the importance of indexes, we do not support the manipulation of indexes.

Chapter 4

Modeling Toolkit

In this chapter, we motivate the need for our persistent application modeling toolkit, AMPS, and then describe it in detail, providing a simple example of its use based on the binary tree example of the previous chapter.

4.1 Motivation and Overview

Experimental performance evaluation requires that performance be measured with respect to a particular workload. Unfortunately, in the field of persistent object systems, standardized experimental workloads have not been developed. Experimental results are presented based on a wide variety of benchmarks, including the OO7 benchmark suite [10], the Hypermodel benchmark [6], the OO1 benchmark suite [24], and the Trouble Ticket Benchmark [31].

As an example of this situation, consider that currently there exist no workloads specifically designed for use in analyzing storage reclamation techniques in persistent object systems [1]. Therefore, researchers have developed their own synthetic applications and workloads. Amsaleg et al. [2] used linked lists of 80-byte objects in their studies on efficient incremental garbage collection. Yong et al. [46] used a subset of the OO7 benchmark suite in their study of storage reclamation and reorganization of persistent object stores. Maheshwari and Liskov [30] used a homogeneous collection of 30-byte objects in their study of partitioned garbage collection of a large object store. Lastly, Cook et al. [15] used a forest of augmented binary trees of objects containing some number of non-tree edges.

We feel that the lack of standard workloads is based in part on two factors: first, the effort required to develop a good workload, and second, the lack of infrastructure to share workloads once developed. We have already described PTF, a format for sharing workload traces. We now describe AMPS, a toolkit facilitating the creation of such traces through the modeling of persistent object system workloads.

We assume that AMPS users have in mind a workload that consists of a schema describing the structure of the store and a collection of behaviors associated with an application that manipulates the store. Common behaviors include the creation of the persistent data, the reorganization of the data, and traversals that update and query the objects in the store. Behaviors are then combined together to create a complete workload. AMPS allows complex workloads to be created quickly, allows the user to rapidly script different combinations of application behaviors, and supports the generation of PTF trace files that result from executing the workload. Thus, the goal of AMPS is to provide a rich shared infrastructure for developers of persistent object systems to evaluate their designs.

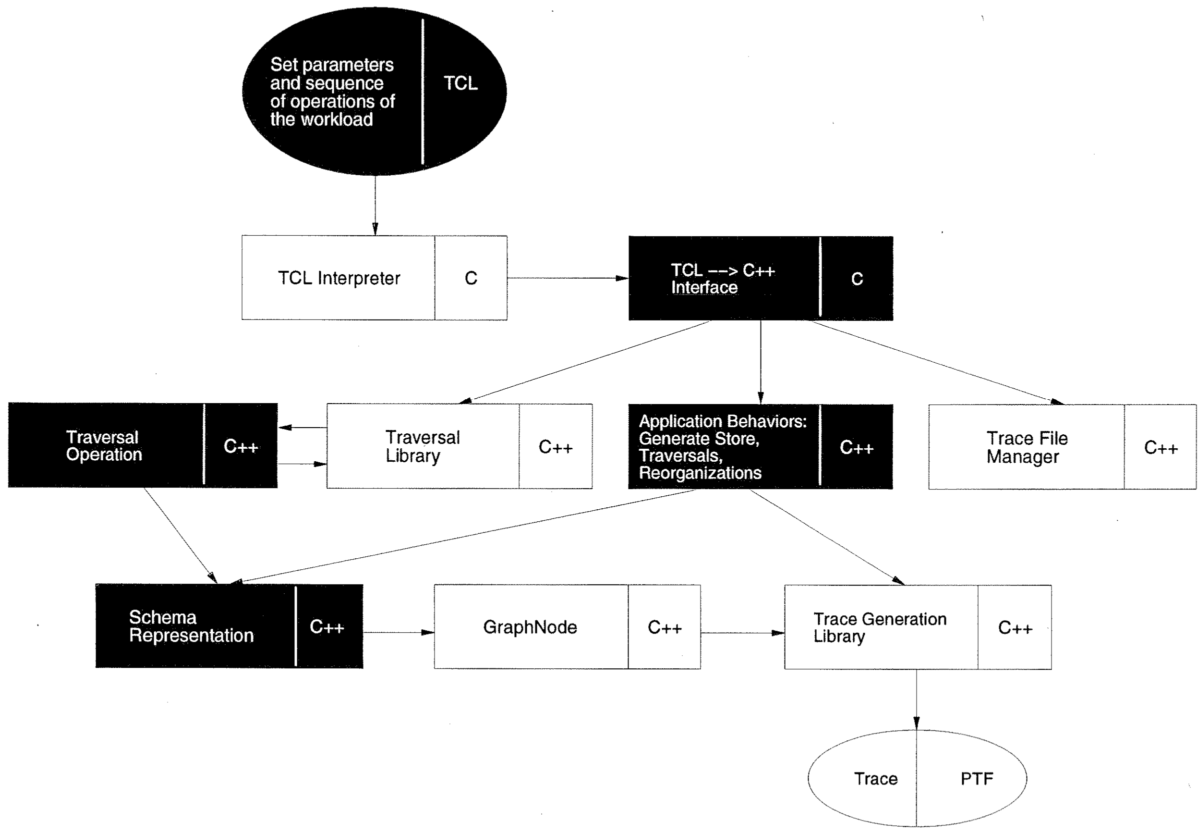


Figure 4.1: AMPS Architecture.

4.2 Architecture

Figure 4.1 presents the architecture of the AMPS toolkit. The user of AMPS is responsible for the dark portions of the diagram, while the remaining portions are provided by AMPS itself. As the diagram shows, AMPS consists of several components: a TCL interpreter, a collection of C++ classes for modeling objects of the persistent store and creating traversals, and a trace generator.[1] Let us examine Figure 4.1 from bottom to top.

The schema for the persistent store, depicted as the box Schema Representation, is provided by the user of the AMPS toolkit. The user takes the object types represented in this schema specification and converts them to C++ classes that inherit from an abstract class, `GraphNode`, provided by AMPS. This class interacts with other AMPS classes, depicted as the box Trace Generation Library, to generate the appropriate PTF events during execution of the application.

The box Application Behaviors depicts the implementations of the application behaviors (e.g., traversals, reorganizations, etc.) created by the user. A library of traversals, depicted as the box Traversal Library, is provided by AMPS to help the user in creating traversals. As a starting point,

[1]While the AMPS prototype currently requires the use of C++ for modeling applications, any class-based object-oriented language would be appropriate. For example, we anticipate that a port of AMPS to Java would be straightforward.

the library currently contains generic facilities for a pre-order depth-first traversal, a post-order depth-first traversal, and a breadth-first traversal. The library is designed to allow additional traversal types to be added by subclassing its type hierarchy. The actual actions taken during the traversal (e.g., read or write an object's data value), depicted as the box Traversal Operation, are specified by the user as traversal operations written in C++. The box Trace File Manager depicts facilities for managing the trace files created during execution of an application.

The user must design a workload and implement it so that it executes in a TCL environment, as indicated by the presence of the box TCL Interpreter in Figure 4.1. This workload consists of several TCL commands that are implemented in C, with an interface to C++ methods. Once the commands of the workload have been implemented, the user may interactively execute them, generating PTF trace files.

4.3 Modeling an Application Workload

AMPS provides support for modeling a persistent object application through C++ classes and a TCL environment. Through the use of AMPS, an in-memory version of the persistent object store can be generated and then manipulated. Here we illustrate the process of modeling a persistent application with AMPS using the simple binary tree object store from the example in Section 3.2. In our example application, each node of the tree contains an integer value. A diagram of the binary tree persistent store is shown in Figure 3.1.

We first discuss the process of translating the schema of the persistent store into AMPS. We then discuss the process of implementing some behaviors of the application, including populating the store and traversing it. Finally, we describe how an application workload is implemented and executed under the TCL environment.

4.3.1 Implementing the Persistent Store Schema

AMPS is implemented in C++. Prior to using the classes of AMPS, a developer of a persistent object application identifies the objects of the persistent store and the relationship among these objects. The developer then specifies these objects as C++ classes.

In AMPS, objects are treated as nodes of a graph with directed edges. By modeling the persistent store as a graph, AMPS captures the relationship between objects through the directed edges of the graph. AMPS provides the class `GraphNode` to support the manipulation of an object of the persistent store as a node of a graph. `GraphNode` uses other C++ classes from AMPS to instrument the application so that the structure of the persistent store and the behavior of the application are captured in a PTF trace.

Viewing the persistent store structure as a graph, the developer of the persistent object application translates C++ specifications of the classes representing objects of the persistent store so that they are treated as nodes of a graph. In this translation, all classes representing objects of the persistent store inherit from `GraphNode`. Also, all fields of the classes are accessed and updated through methods. Using methods to manipulate fields allows instrumentation of the application to be restricted to those methods. In addition to the above, edges are formed from fields that contain pointers to other objects. Thus, these fields are no longer explicitly declared within the class. All edges are read and updated through their offsets.

A portion of the specification for `GraphNode` is shown in Figure 4.2. We briefly mention some interesting aspects of it here. The constructor takes as input information that is needed to capture the structure of a persistent store. The constructor also allocates a structure to hold the edges of

```
class GraphNode {
     friend class GraphNodeIter;
   public:
     GraphNode(int objectType, int edges, char *noteBuf = NULL);
     int setEdge(GraphNode *toVertex, int edgeNum,
                 char *noteBuf = NULL);
     GraphNode *getEdge(int edgeNum, char *noteBuf = NULL);
     void setRoot(char *noteBuf = NULL);
     void getAdjacentVertexList(GraphNode **adjacentList);
     void readData(int position, char *noteBuf = NULL);
     void writeData(int position, char *noteBuf = NULL);
     int getNumberOfEdges();
     int getNodeIdentifier();
     ~GraphNode();

     ...
};
```

Figure 4.2: C++ Class Specification of `GraphNode`.

the object. The method `setEdge` writes an edge given an offset. The method `getEdge` returns the object associated with a given offset. The method `setRoot` records a trace event to indicate that the current object is a root in the persistent store. The method `getAdjacentVertexList` returns, through a vector, all of the objects that are connected to a given object. The method `readData` records the access of a field that does not contain a pointer value, and the method `writeData` records the updating of a field that does not contain a pointer value. The method `getNumberOfEdges` returns the number of edges associated with an object. The method `getNodeIdentifier` returns the object identifier that AMPS has associated with a given object. Each of the methods that create an event in the trace have an optional input parameter (`noteBuf`) that can be used to attach an ASCII string as an annotation on the trace entry for the event.

We now consider a specific example. Suppose we want to model the binary tree node of our simple example in which we need left and right children and a data value. The class specification for this binary tree node constructed using AMPS is shown in Figure 4.3. In this figure, the class `BinTreeNode` inherits from `GraphNode`. The only field of the class is `nodeValue`, so the methods `refNodeValue` and `setNodeValue` are added to the class specification to manipulate the value associated with the field `nodeValue`. In addition to the above, the references to the left and right children are implemented as edges referring to objects of class `GraphNode` manipulated through methods `setLeftChild`, `setRightChild`, `refLeftChild`, and `refRightChild`. Figure 4.3 also shows the implementation for the access method `refNodeValue`, which returns the value of `nodeValue`. In the method `refNodeValue`, there is an invocation of the method `readData`, which is used to record the reading of data from the location referred to by the field `nodeValue`. Furthermore, this figure shows the implementation of the access method `setRightChild`, which illustrates the use of the method `setEdge`.

```
class BinTreeNode: public GraphNode {
  public:
    BinTreeNode(int typeIdentifier,int edges, char *noteBuf = NULL):
        GraphNode(typeIdentifier, edges, noteBuf)
        { nodeValue = 0; }

    int refNodeValue(int position);
    void setNodeValue(int position, int val);
    void setLeftChild(GraphNode *parentNode);
    void setRightChild(GraphNode *parentNode);
    GraphNode *refLeftChild();
    GraphNode *refRightChild();

  private:
    int nodeValue;
};

// Example implementations of two of the access functions

// Method to reference the nodeValue data member
//
int BinTreeNode::refNodeValue()
{
                             // POSITION_IN_TYPE indicates attribute's
                             // position within type definition
     readData(POSITION_IN_TYPE, NULL);   // NULL => empty comment
     return(nodeValue);
}

// Method to update the RightChild reference
//
void BinTreeNode::setRightChild(GraphNode *parentNode)
{
   parentNode->setEdge(this, 1, NULL); // 1 => RightChild
}
```

Figure 4.3: C++ Class Specification of BinTreeNode.

4.3.2 Implementing Application Behaviors

A persistent object application consists of a collection of behaviors that manipulate the data of the application. These operations perform a variety of tasks, such as updating an object of the persistent store, referencing the data of an object, or updating the structure of the persistent store by adding or deleting an object. As mentioned, AMPS provides a library of traversals in order to support standard graph traversal algorithms.

In AMPS, traversals are implemented as C++ objects. The operations performed on each object in the course of traversing the persistent store are also implemented as C++ objects. Normally an operation of a persistent store would be implemented as a method of a class with optional formal parameters and an optional return value. By treating operations as objects, the formal parameters and return value become fields of the class representing the operation object. The fields representing the formal parameters are then initialized using the constructor of the class. The field representing the return value is accessed via a method of the class.

Now, suppose a developer is implementing an application that manipulates the binary tree persistent store of Figure 3.1. Also, suppose that the application consists of a breadth-first traversal to sum the integer values contained at each node of the binary tree. Using AMPS, the developer would implement an operation object that would add the integer value of a node to the sum. By combining the operation object with the breadth-first traversal object, the binary tree persistent store can be traversed using a breadth-first algorithm and the total of the integer values can be calculated.

By implementing traversals and operations as objects, we were able to design a set of generic traversals. In addition to the above, the implementation supports the development of complex traversals. Complex traversals are traversals that consist of several simple traversals, where the type of traversal employed is determined at run time while manipulating an object.

The traversal classes inherit from the virtual class `Traversal`, which sets up the interface for the traversal classes. At the top of Figure 4.4 is the specification for `Traversal`. The method `traversalApply` implements the traversal algorithm, such as breadth first or depth first. It takes as input the starting node for the traversal. In the middle of Figure 4.4 is the class specification for the breadth-first traversal object. The constructor for the breadth-first traversal object takes as input the number of objects in the persistent store and a pointer to the operation object to be performed at each object of the persistent store. The method `traversalApply` invokes this operation at each object that is visited while performing the breadth-first traversal on the persistent store structure. The method `getCurrentNode` returns a pointer to the object that is the last visited node during the processing of the traversal. The traversal class has three fields as shown in Figure 4.4. The fields represent the current node (`currentVertex`), an array to keep track of the visited nodes (`visitedArray`), and a pointer to the operation object (`queryOption`).

In the generic traversals supported by AMPS, an operation is performed at each node in the graph traversed. These operations are implemented as classes that inherit from the virtual class `TraverseOption`, which is shown at the bottom of Figure 4.4. In this specification, the method `apply` takes as input a pointer to a node of the persistent store and performs the task defined by the method on this node.

Using the binary tree persistent store, we illustrate how to define the operation on a node as an object. Recall that in our example, the operation on the node was to take the integer value of that node and add it to a total. The class representing this operation is called `SumNodes`. A C++ class specification for this operation is shown in Figure 4.5. `SumNodes` consists of a constructor that initializes the field `totalSize`, which is the result of applying this operation to each of the nodes of

```
class Traversal {
  public:
    virtual GraphNodePtr getCurrentNode() = 0;
    virtual void traversalApply(GraphNodePtr startVertex) = 0;
};

class BreadthFirst : public Traversal {
  public:
    BreadthFirst(int noOfNodes, TraverseOption *queryFunction);
    void traversalApply(GraphNodePtr startVertex);
    GraphNodePtr getCurrentNode()
        { return (currentVertex); }
    ~BreadthFirst( );

   protected:
     GraphNodePtr currentVertex;
     int *visitedArray;
     TraverseOption *queryOption;
};

class TraverseOption {
 public:
   virtual int apply (GraphNodePtr currentNode) = 0;
};
```

Figure 4.4: Class Specifications of `Traversal`, `BreadthFirst`, and `TraverseOption`.

```
class SumNodes: public TraverseOption {
  public:
    // initialization of the operation object
    SumNodes() { totalSize = 0; }

    // operation performed at each visited node of the Traversal
    int apply(GraphNodePtr Node) {
        totalSize = totalSize + ((BinTreeNode *)(Node))->RefValue();
    }

    // result of traversal, if there is one
    int getSum() { return(totalSize); }

  private:
    int totalSize;
};
```

Figure 4.5: C++ Class Specification of `SumNodes`.

the binary tree persistent store. The method `getSum` returns the value associated with `totalSize`.

By combining this operation object with the breadth-first traversal object, the binary tree persistent store is traversed using a breadth-first algorithm and the total of the integer values is calculated. In order to apply the `SumNodes` operation to each node of the binary tree persistent store, the breadth-first traversal constructor is invoked with a pointer to a `SumNodes` object as an actual parameter. Upon the completion of the execution of the method `traversalApply` for the breadth-first traversal, the method `getSum` can be invoked for the instance of the class `SumNodes` to obtain the sum of all the integer values of the binary tree persistent store.

4.3.3 Implementing the Workload

Prior to using AMPS, developers must have some idea of the workload that they wish to perform for a specific persistent object application. With AMPS, once the specific behaviors of the workload are implemented, the actual execution of the workload can be performed interactively using the TCL interpreter.

To use TCL, the developer must implement TCL commands that reflect the individual behaviors of the workload. For example, there might be a TCL command to generate the persistent store or there might be a TCL command that represents a specific query of the application.

AMPS provides an example application along with two TCL commands to illustrate how to design and implement the commands of the workload for use under TCL. These commands are `TG_DBbuild` and `TG_Traversalbuild`. The command `TG_DBbuild` builds a binary tree persistent store. The command `TG_Traversalbuild` invokes either a breadth-first or a depth-first traversal on a persistent store given a specific operation to be performed. These examples can easily be specialized to the needs of a particular persistent object application.

TCL allows user-level commands to be implemented with C functions. In Figure 4.6, the C implementation of the TCL command `TG_DBbuild` is shown. In this case, the TCL command `TG_DBbuild` is bound to the C function `TG_DBbuildCmd`. All inputs to the TCL commands are ASCII

```
int TG_DBbuildCmd(ClientData clientData, Tcl_Interp *interp,
                  int argc, char *argv[])
{
    // Error checking code omitted

    int numOfLevels = atoi(argv[1]);

    // Disable garbage collection during database generation
    //
    Traceobject_BeginNoGC( );
    DBptr = BinTree_new(numOfLevels);

    // Enable garbage collection after database generation
    Traceobject_EndNoGC( );

    // Setup result string and return
    interp->result = "Database generated";
    return TCL_OK;
}
```

Figure 4.6: Implementation of the TCL command `TG_DBbuild`.

character strings, which first must be converted to the proper type. For example, `TG_DBbuild` takes as an argument the depth of the binary tree in the persistent store. The argument representing the depth is converted to an integer value and stored in the variable `numOfLevels` as shown in Figure 4.6. In implementing the binary tree persistent store, we implemented a class to represent the binary tree structure. This class contains a method to create the persistent store. Also, notice that the output from the function that implements the command is a string value. The variable `interp->result` contains the ASCII value to be printed as the result.

In a manner similar to `TG_DBbuild`, a TCL command to invoke the operation `SumNodes` (called `TG_SumNodesCmd`) can also be constructed.

Finally, using the binary tree persistent store example, we illustrate in Figure 4.7 how to script a workload using AMPS. In this example, we create a seven-node binary tree persistent store. We then traverse the persistent store using the breadth-first traversal and the operation `SumNodes`, which sums the integer value at each node of the persistent store. In this scenario, we compute the sum and then close the trace file. The TCL commands that an AMPS user types are located next to the percent sign and the responses to these commands are shown on the following line.

4.4 Summary

This chapter described the AMPS toolkit, which is a framework for creating an instrumented application. A simple application was used to show how to use the toolkit to effectively model and implement a persistent object application. Through the C++ libraries of the toolkit, the effort to implement an instrumented model of an application is reduced in three ways. First, the generic traversals provided by the toolkit reduce the effort to implement and model traversals of

```
sheriff% TGenApp
   % TG_OpenTraceFile btree3
     Trace file opened
   % TG_DBbuild 3
     Database generated
   % TG_Traversalbuild breadthfirst SumNodes
     Traversal processing completed
   % TG_CloseTraceFile
     Trace file closed
   % exit
sheriff%
```

Figure 4.7: Scripting a Workload Using AMPS and TCL.

the application. Second, the methods of the toolkit can be used to generate trace events. Third, the effort to instrument an application is reduced by mimimizing the likelihood of errors because instrumentation is restricted to access methods of classes of the application.

This chapter also illustrates how to generate workloads using the TCL interpreter of the AMPS toolkit. With the TCL interpreter, a researcher can easily script a variety of workloads.

As mentioned at the end of Chapter 3, there are some objects and data structures that are part of the persistent object system implementation (e.g., indices, extents) and not the application itself. Conceptually, objects of the persistent object system are separate from and should be modeled separately from the application objects. AMPS currently does not allow objects of the persistent object system to be distinguished from application objects and, as a result, someone using AMPS must implement the objects of the persistent object system necessary to accurately model the situation. To be more concrete, in the next chapter we discuss the fact that extents were required in our modeling of the OO7 benchmark.

Chapter 5

Experience Modeling OO7

The OO7 benchmark [10] has been used in evaluation studies of persistent object storage management [13, 46]. Because of its availability and its use in these evaluation studies, we selected it as the primary application to study the feasibility of the AMPS toolkit. The OO7 benchmark is intended to mimic CAD/CAM applications, but does not model any specific application. In a study evaluating the suitability of the OO7 benchmark as an application benchmark, it was found that the data structures mapped reasonably well to those of a large mechanical CAD/CAM application [43].

The developers of the OO7 benchmark distribute several implementations, including an E version and a C++ version. Cook et al. [13] hand instrumented a subset of the E version, which is based on the Exodus storage manager, and used it to generate traces to drive their analyses. After developing the AMPS toolkit, we created an AMPS-based C++ version of the benchmark. Using these two implementations of the same benchmark we were able to assess the correctness of the automation provided by the toolkit simply by comparing the generated traces from each implementation. This chapter describes our experience building the the AMPS-based implementation and the results of the assessment.

5.1 OO7 Overview

The OO7 benchmark provides a schema for a persistent store together with several detailed scenarios for creating and accessing data. The largest logical unit of the OO7 schema is the *module*. A particular persistent store may contain one or more modules. Each module consists of a *manual* and a hierarchy of *assemblies*. Manuals are used to represent large objects in the store, and have associated with them a dynamically allocated amount of text. The hierarchy of assemblies consists of complex assemblies, and below that, base assemblies. There is a bi-directional association between complex assemblies and their subassemblies. A diagram of the structure of a module (absent the manual) is shown in Figure 5.1.

The complexity of assembly objects can be controlled through parameters provided by the benchmark, such as the number of subassemblies associated with an assembly, the degree of connectivity among components of the subassemblies, and the number of levels in the assembly hierarchy. The base assemblies are composed of composite parts, some of which are shared and some of which are unshared among base assemblies. Each base assembly has a bi-directional association with its composite parts. In Figure 5.1, the composite parts are represented by grids. The grid is meant to indicate that each composite part consists of a graph of *atomic parts*. Each atomic part can be connected to 3, 6, or 9 other atomic parts. There is also a bi-directional association between the atomic parts that make up a composite part and the composite part. Similar to the

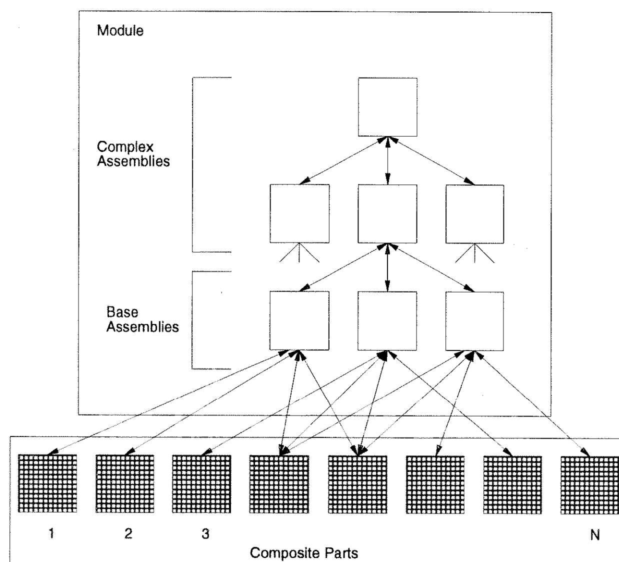


Figure 5.1: Diagram of the Module Object of the OO7 Benchmark.

manuals of whole modules, a composite part has associated with it a *document* object whose text is dynamically allocated.

The benchmark specifies that a single transaction is used to create a persistent store. Traversals that manipulate the store navigate through the graph of objects, invoking methods associated with the visited objects. The original specification of the benchmark did not include any behaviors that modified the store. Therefore, Yong et al. [46] defined several additional behaviors, including two reorganization functions that target some set of the atomic parts for deletion. New atomic parts are then created to replace them. In the remainder of this chapter, when we refer to the OO7 benchmark, we are referring to this enhanced version.

5.2 Using AMPS to Model OO7

Building an implementation of an experimental application is a significant undertaking. It involves defining the schema and programming the behaviors. Instrumentation adds a further burden. AMPS requires the definition of the schema to be built in terms of the AMPS class library, but with the benefit that instrumentation is achieved with little extra effort. Moreover, AMPS was designed so that the analyst can take advantage of any pre-existing C++ implementation code with little additional modification. This was the case for us in building the OO7 benchmark, since there was a publicly available C++ implementation.

To assess the quality of the AMPS-based implementation, we set the goal of generating traces whose contents were very similar, if not identical to, the traces produced through the hand instrumentation of the E version of the benchmark that was implemented by Cook et al. [13]. To achieve this goal, we slightly modified the publicly available C++ implementation so that it better mimicked the E version of the benchmark. These modifications were restricted mainly to the order of object creation.

Below, we describe our use of AMPS in modeling the OO7 schema, modeling class extents, and modeling one of the traversals of the OO7 benchmark, T1. Following that, we compare the PTF traces resulting from the two implementations of the benchmark.

5.2.1 Modeling the OO7 Benchmark Schema

Using the publicly available C++ implementation of the OO7 benchmark as a starting point, we made the following modifications to the class specifications constituting the schema. First, we restored the class `DesignObject` from the original benchmark specification. This class serves as an abstract base class for the application classes `Module`, `Assembly`, `CompositePart`, and `AtomicPart`. Second, root classes in the inheritance hierarchy, in particular `DesignObject`, were changed so that they inherited from the AMPS class `GraphNode`. Third, we added access methods to retrieve and modify fields of the application classes. This modification allowed us some level of transparency between instrumentation and the application by restricting instrumentation to the access methods. Finally, pointer fields within the implementations of the classes were reformulated as explicit AMPS edge objects.

Figure 5.2 shows the C++ specification of class `CompositePart` before our modifications for use with AMPS. Figure 5.3 shows the modified versions of the C++ class specifications for `DesignObject` and one of its subclasses, `CompositePart`. Each field, such as `id` in `DesignObject`, now has a pair of access functions, such as `refId` and `setId`. Each pointer field in the original version, such as `documentation` in `CompositePart`, has been replaced by a pair of access functions and, although not shown, now has its value maintained by an edge structure in `GraphNode` (see Figure 4.2). A depiction of an instance of `CompositePart` is shown in Figure 5.4. Notice that pointer fields `documentation`, `usedInPriv`, `usedInShar`, `parts`, and `rootPart` are treated as elements of a vector of pointers in `GraphNode`.

In addition to the above modifications, non-shared dynamic data were inlined, since they are not treated as persistent objects. Examples of non-shared data within the OO7 schema are the text of manuals and documents. The size of the manual and document objects were changed so that they included the size of the text associated with them.

5.2.2 Modeling Extents

As mentioned in Section 4.4, some objects that are modeled in AMPS are actually part of a POS implementation. Here we describe how we used AMPS to model extents in OO7.

Most persistent object systems support the concept of *extent*, which is the set of instances of a class. In the language E, sets of objects are represented as collections. Each collection is instantiated with a specific type that indicates the type of its members. The members of the collection can be either of the type or subtype of the type that was used to instantiate the template of the collection. The C++ implementation of OO7 models extents a bit differently, using a so-called *association* implemented as the class `Assoc`. In order to capture the behavior of extents in our traces, we modified `Assoc` to be a compliant subclass of `GraphNode`.

The original C++ specification for `Assoc` is shown in Figure 5.5, while the version implemented using AMPS is shown in Figure 5.6. In the original version, the pointers representing the association's elements are stored in a fixed-size array denoted by the variable `members`. `BaseSize` is a constant value that indicates the size of the array. In the AMPS implementation, the contents of this array are treated as edges managed by the superclass `GraphNode`.

```
class CompositePart {
  public:
    int id;
    char type[TypeSize];
    int buildDate;
    class Document *documentation;
    class Assoc *parts;
    class AtomicPart *rootPart;
    // list of assemblies in which part is used as a private component
    Assoc *usedInPriv;
    // list of assemblies in which part is used as a shared component
    Assoc *usedInShar;

    CompositePart(int cpId);
    ~CompositePart();
    int traverse(BenchmarkOp op);
    int traverse7();
    int reorg1();
    int reorg2();
};
```

Figure 5.2: Original C++ Class Specification of `CompositePart`.

5.2.3 Modeling a Traversal

While the OO7 benchmark contains a number of traversals, many are quite similar. Traversal T1 serves well as a representative of the group. In this traversal, the assembly hierarchy is traversed by visiting each of the base assemblies. Upon visiting a base assembly, each of its unshared composite parts is visited. A depth-first traversal is then performed on the graph of the atomic parts for each composite part. As an atomic part is visited, it is counted to produce a total number of atomic parts visited during the traversal.

We chose to implement traversal T1 in two ways. In our first approach, we used the generic traversal library that comes with AMPS to implement T1 fully. In traversal T1, the object to visit next is based on the type of the current object. As a result, the traversal operation applied to each node in AMPS has two distinct components, a computation component and a navigation component. The navigation component is defined through a boolean list that indicates which edges should be taken. Each object type contained in the schema specification of the OO7 benchmark is given such a list. As an object is visited during the processing of the directed depth-first traversal, the boolean list is retrieved and used to determine which objects should be visited next. In T1, computation is only performed when an atomic part is visited. Thus, in the apply method of the traversal operation, shown in Figure 5.7(a), the type of the node is checked to determine if it is an atomic part and, if so, the method `DoNothing` of the atomic part class is invoked. `DoNothing` references the `id` attribute of the atomic part object and checks for a negative value. Also, the apply method accumulates the sum of all atomic parts that are visited during the traversal. In the generic depth-first traversal, a list of all atomic parts is maintained per composite part to prevent cycles during the traversal.

```
class DesignObject: public GraphNode {
   public:
     DesignObject(int typeIdentifier, int edges, char *noteBuf = NULL):
          GraphNode(typeIdentifier, edges, noteBuf){ };
     int refId();
     void setId(int value);
     void setType(char *&typestring);
     void setType(char *typestring, int length);
     int refBuildDate();
     void setBuildDate(int value);


   private:
     int id;
     char type[TypeSize];
     int buildDate;
};


class CompositePart: public DesignObject {
  public:
    Document *refDocumentation();
    void setDocumentation(Document *value);
    Assoc *refParts();
    void setParts(Assoc *value);
    AtomicPart *refRootPart();
    void setRootPart(AtomicPart *value);
    Assoc *refUsedInPriv();
    void setUsedInPriv(Assoc *value);
    Assoc *refUsedInShar();
    void setUsedInShar(Assoc *value);

    CompositePart(int cpId);
    ~CompositePart();
    int traverse(BenchmarkOp op);
    int reorg1();
    int reorg2();
};
```

Figure 5.3: Class Specification of AMPS Versions of `DesignObject` and `CompositePart`.

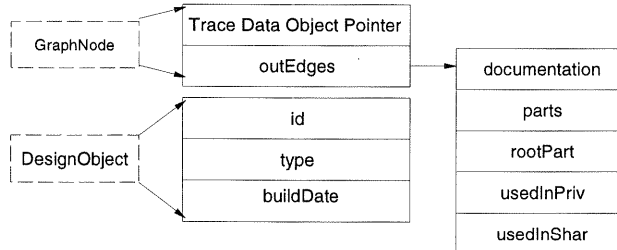


Figure 5.4: Graphical Representation of an Instance of `CompositePart`.

```
class Assoc {
  public:
    Assoc();
    ~Assoc();
    void add(void *member);
    void remove(void *member);

  private:
    int allocated;          // actual size
    int curSize;            // number of elements used
    int scanPtr;            // index into members array
    void *members[BaseSize]; // connection pointers
    Assoc *overflow;          // beginning of the overflow chain
};
```

Figure 5.5: Original C++ Class Specification of `Assoc`.

```
class Assoc: public GraphNode {
  public:
    Assoc(char *noteBuf = NULL);
    ~Assoc();
    void add(GraphNode *member);
    void remove(GraphNode *member);
    int refAllocated();
    void setAllocated(int value);
    int refCurSize();
    void setCurSize(int value);
    int refScanPtr();
    void setScanPtr(int value);
    Assoc *refOverflow();
    void setOverflow(Assoc *value);
    GraphNode *refMemberSubI(int index);
    void setMemberSubI(GraphNodePtr member, int index);

  private:
    int allocated;       // actual size
    int curSize;         // number of elements used
    int scanPtr;         // index into members ac:
};
```

Figure 5.6: C++ Class Specification of AMPS Version of `Assoc`.

```
int Traverse1::apply(GraphNodePtr currentNode)
{
    if (currentNode->getType() == ATOMICPARTTYPE){
        count++;

        // Perform the AtomicPart::DoNothing method

        ((AtomicPart *) currentNode)->DoNothing();
    }
    return(0);
}
```

(a)

```
int DFSTraverse::apply(GraphNodePtr currentnode)
{
    int count;

    // benchOp --> Traversal Type
    // benchOp is a data member of the DFSTraverse class
    // and is initialized in its constructor.

    count = ((Module *) currentnode)->traverse(benchOp);
    return (count);
}
```

(b)

Figure 5.7: AMPS-Native (a) versus Existing-Code (b) Invocations of the Traversal T1 Apply Method.

Table 5.1: Parameters for a Small OO7 Persistent Store.

Parameter	Value
NumAtomicPerComp	20
NumConnPerAtomic	3
DocumentSize (bytes)	2000
ManualSize (bytes)	100000
NumCompPerModule	150
NumAssmPerAssm	3
NumAssmLevels	6
NumCompPerAssm	3
NumModules	1

Using the traversal class provided by AMPS as the base class, new traversal classes can be implemented that reuse existing code of an application already available in C++. In the second implementation of traversal T1, we created a new traversal class that uses the original T1 C++ code. The apply method of the traversal operation is shown in Figure 5.7(b). This method is only invoked once, unlike its counterpart in the generic implementation of traversal T1. The root node, a module object, is supplied as input to the apply method. As shown in Figure 5.7(b), the apply method invokes the method `traverse` of class `Module` from the existing C++ implementation. This method returns the number of atomic parts visited. The variable `benchOp` indicates which variant of OO7 traversals to execute. The navigation of the traversal is handled directly by the methods that are called.

In order to reuse the existing C++ code in the second implementation of traversal T1, references to fields were changed to calls to access methods, which provide the instrumentation functionality. Secondly, the pointers to objects that are obtained through the iterator for the association class are cast to their proper type. It is important to note that the changes to the original code were very minor.

5.3 Comparison of PTF Traces

To demonstrate the degree to which traces gathered from an AMPS-instrumented implementation of an application can remain faithful to the traces gathered from a hand-instrumented implementation, we performed a comparison of traces from both implementations. Using the OO7 benchmark parameters listed in Table 5.1, we gathered traces from the hand-instrumented E version of the OO7 benchmark implementation and the AMPS-instrumented C++ version. (Although not shown here, we also executed both implementations of traversal T1, based on the original traversal code and the generic AMPS traversal code, and collected traces from each. The traces were identical with respect to navigation and the number of atomic parts that were visited during the execution of the traversal.)

To compare the traces, a post-processing program collected the number of occurrences of a given event type per object type. In Table 5.2, the number of occurrences per object type are shown for both the AMPS trace and the hand-instrumented trace. Notice that the number of occurrences of trace events are identical for all object types. The only exception is for association objects. This difference can be attributed to a small difference in the implementations of the association type in the two versions.

Table 5.2: Comparison of AMPS-Instrumented and Hand-Instrumented Traces.

OO7 Object Types	**Trace Event Types**	**AMPS/C++ Version**	**Hand-Instrumented E Version**
Module	co	1	1
	er	243	243
	ew	3	3
	dr	0	0
	dw	3	3
Complex Assembly	co	121	121
	er	363	363
	ew	242	242
	dr	0	0
	dw	484	484
Base Assembly	co	243	243
	er	1458	1458
	ew	729	729
	dr	0	0
	dw	972	972
Composite Part	co	150	150
	er	4458	4458
	ew	750	750
	dr	0	0
	dw	450	450
Atomic Part	co	3000	3000
	er	18000	18000
	ew	9000	9000
	dr	0	0
	dw	18000	18000
Connection	co	9000	9000
	er	0	0
	ew	18000	18000
	dr	0	0
	dw	18000	18000
Association	co	7252	7253
	er	35443	5576
	ew	104636	104647
	dr	166060	134135
	dw	71102	78358

5.4 Summary

Modeling the OO7 benchmark using the AMPS toolkit provided more insight into the modeling of persistent object applications in general. In the simple persistent application of Chapter 4, we did not use any extents and thus did not look at how to model extents until we were faced with modeling them to capture the behavior of the OO7 benchmark. In addition to extents, we were faced with issues of how to treat dynamically allocated data such as the text of documents. This led to enhancements to PTF (e.g., trace events to create and manipulate dynamic arrays). Last but not least, we found that the graph model, on which AMPS is based, was well suited in capturing the relationship of the objects of the persistent store generated by the OO7 benchmark.

In order to validate the traces generated by the AMPS-instrumented C++ implementation of the OO7 benchmark, we compared them to traces that were generated by a hand-instrumented E version. These traces were previously verified through the use of a simulator of a persistent object system. The comparison showed that for all of the objects of the application, the AMPS-instrumented version produced the same number of events as the hand-instrumented E version. From this, we concluded that the traces generated by the AMPS-instrumented version correctly captured the behavior of the OO7 benchmark.

Chapter 6

Infrastructure to Support Generation of Multi-user Workloads

Generally, real applications of persistent object systems are executed in a multi-user environment. Thus, in order to understand and evaluate the performance of a persistent object system, it is critical to understand how these systems perform under a multi-user environment. However, performance studies provide little insight into the performance characteristics of a multi-user environment. Although two efforts [9, 41] were made to develop a multi-user benchmark, the community of persistent object systems' researchers have not adopted either as a standard. To date, there is no consensus as to what constitutes a representative multi-user workload, which is a combined workload of multiple clients concurrently accessing a persistent store. This is especially true in performance studies of storage management algorithms of persistent object systems. Due to the lack of a standard benchmark, researchers continue to generate their own workloads to perform experimentation on storage management algorithms, thus limiting the sharing of data between researchers and hindering the ability to perform direct comparison of persistent object systems.

The work described in this chapter attempts to address the issue of sharing of data through a trace-driven approach that captures the behaviors within a multi-user workload. It is novel in that single user workloads are generated and captured in trace files and these trace files are later interleaved using a concurrency control model and a transaction model to generate a multi-user workload.

Multi-user environments in which multiple users access a persistent object system have several features that impact both the correctness and the ability to model a multi-user workload. Some of these features are as follows:

- **Concurrency**: Within multi-user environments, multiple users can access the persistent store concurrently under the transaction management of the persistent object system. An approach that models this environment must be able to capture an interleaving of the transactions in a manner that supports serializability. Each transaction must guarantee the ACID properties (e.g., atomicity, consistency, isolation and durability), as defined in the ODMG standard [11].

- **Data Sharing**: When multiple users access a persistent store, some data are typically accessed by more than one user and thus there must be some mechanism to prevent interference between competing users. When modeling a multi-user environment, a desirable feature is one that provides some control over the amount of sharing among users.

- **Atomic Transactions**: Transactions within a multi-user environment may abort. The

update events of aborting transactions are not captured in the trace representing the multi-user workload.

In this chapter, we describe a trace-driven approach to collecting data that captures the behavior of an application in a multi-user environment. The trace-driven approach has the following characteristics:

1. A trace file is generated for each client used to model the multi-user workload independently of a persistent object system as described in Chapter 4.
2. Events indicating transaction boundaries enclose trace events that are generated by operations manipulating the persistent store as part of a given transaction.
3. The traces are processed under a simulated multi-user environment to capture the combined behaviors of the multiple users. These traces contain flat transactions that are processed under a concurrency control policy.
4. Experimentation with alternate transactions model and concurrency control policies is possible since the transaction model and the concurrency control mechanism are orthogonal.
5. The final trace file represents the results from interleaving transactions of the participating clients and captures the behavior of a multi-user workload.

The remainder of this chapter is organized as follows. In the second section, the design goals of the instrumentation infrastructure are presented and a brief overview of the approach is provided. It concludes with a list of assumptions upon which the design of the infrastructure is based. The third section describes the design issues that affect the implementation of the infrastructure. The fourth section presents the PTF enhancements to capture the behavior of transactions. The fifth section presents scenarios to illustrate how the single-user workloads are merged to form an interleaved trace and presents issues related to the creation of the interleaved trace. The sixth section provides an overview of an example implementation of the trace merger.

6.1 Setting the Stage

In this section, we briefly describe the trace-driven approach that has been designed in this dissertation to create a trace file that captures the behavior of persistent object applications that are executed in a multi-user environment.

6.1.1 Design Goals

The design goals of this instrumentation infrastructure are as follows:

- To design a flexible technique for generating workloads that allows the number of clients to vary as well as the tasks that make up the workloads.
- To generate a multi-user workload without dependence on the concurrency control algorithm enforced by either an implementation of threads or by the underlying operating system on which the single-user workloads are generated.
- To generate workloads whose behavior is repeatable and reusable without re-running the application to produce the workloads.

- To build a flexible experimental environment in which alternative scheduling algorithms, concurrency control models, and transaction models can be tested.

- To generate a trace file that can be used in studying storage management policies.

6.1.2 Approach To Support Generation of Multi-user Workload

Figure 6.1 presents an approach to instrumentation of multi-user workloads to generate a trace for use in analysis of persistent object systems. In Figure 6.1, the new steps to the trace-driven approach that was shown in Figure 1.1 are highlighted through the use of light gray shading. As shown in Figure 6.1 there are two phases in the trace-driven approach: a collection phase and a processing phase. In the collection phase, AMPS is used to instrument a given persistent object application. From the instrumented application, a single-user workload, represented by `Client Workload` in Figure 6.1, is specified for each client. Several single-user workloads are then executed to produce PTF traces. These traces are then used in a two-step processing phase. In the first step of the processing phase, multiple traces representing single-user workloads are input to the trace merger. The trace merger simulates the execution of the single-user workloads under a multi-user environment. Transactions of the multiple client workloads are interleaved to form a trace consisting of events that manipulate the persistent store and provide directives to the persistent object system.

The trace merger consists of a scheduler, a swapper, a transaction manager, and a lock manager. Each client represented by a trace is simulated as a process. The scheduler determines the order of execution of these pseudo-processes. Once a process has been given a time slice, the swapper processes the events of the trace corresponding with the selected client. The transaction manager processes the events of a simulated transaction and interacts with the lock manager to enforce a given concurrency control model. As the simulated transactions are processed, an interleaved trace is generated. This interleaved trace is then used as input to a simulator for analysis of storage management algorithms.

6.1.3 Assumptions

Our design of an instrumentation infrastructure to support the generation of multi-user workloads depends on several assumptions. These assumptions are as follows:

- **Assumption A1:** All operations of an application that manipulate the persistent store are executed as part of a transaction under a concurrency control mechanism, guaranteeing atomicity and consistency.

- **Assumption A2:** The data value of an attribute of an object cannot be used to guide the behavior of the operations of a transaction. The data value also cannot determine whether a transaction executes or does not execute.

- **Assumption A3:** The set of transactions that are serially executed to form the single-user workload can be executed concurrently with the set of transactions of another client to form an interleaved schedule that is serializable.

- **Assumption A4:** Transactions that abort due to locking conflicts will eventually complete successfully.

- **Assumption A5:** Structural changes can only occur within a region of the persistent store that is only accessible to the client issuing the structural change.

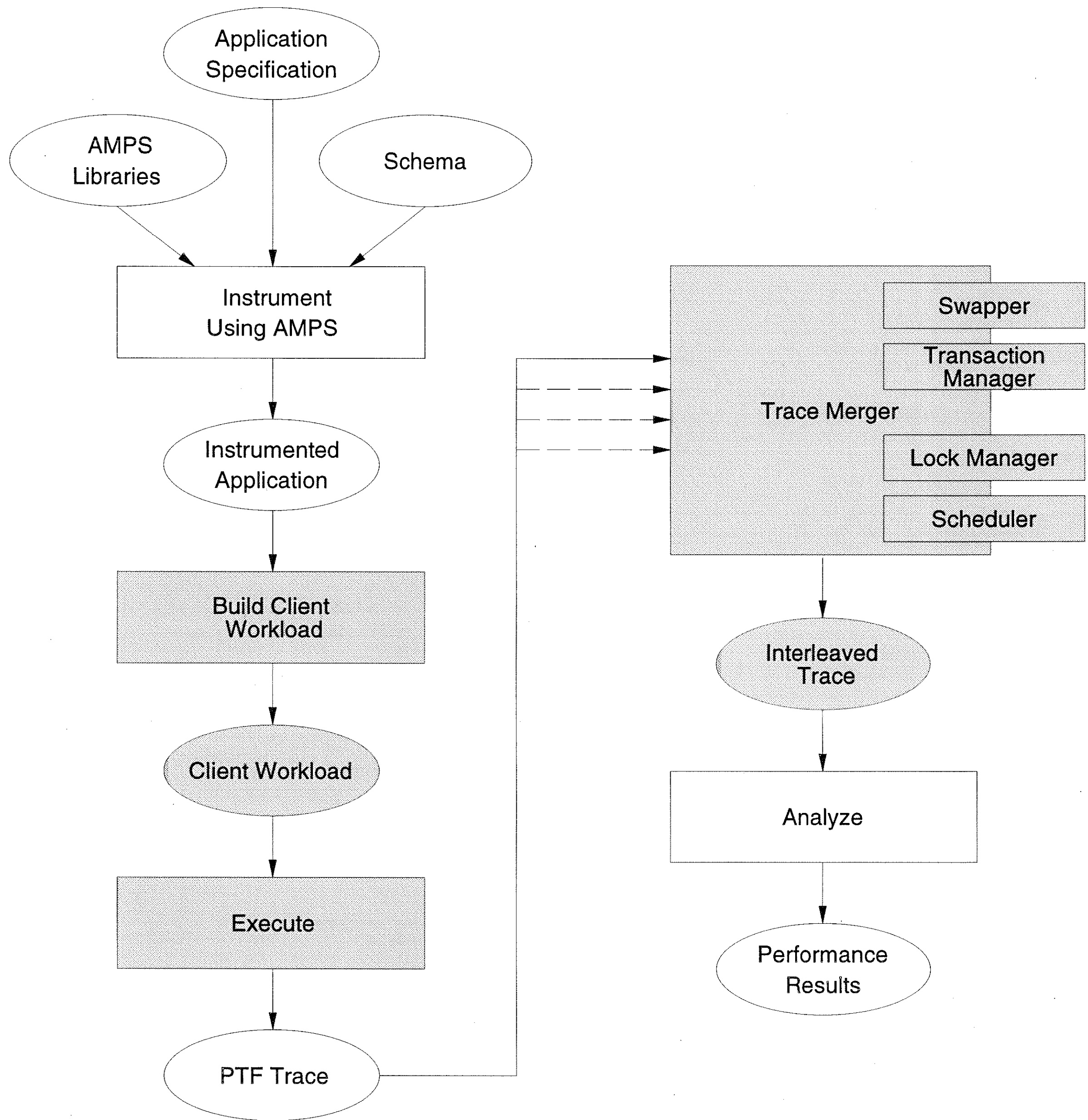


Figure 6.1: Approach to Generation of Multi-user Workloads.

Assumptions A1, A2, A3, and A4 guarantee the correctness of the interleaved trace that is produced by merging the single-user traces. They also guarantee that the persistent store generated by the trace is consistent. Assumption A5 prevents inconsistencies from occurring in the interleaved trace.

6.2 Design Issues

Several issues were investigated in the process of designing the multi-user workload instrumentation infrastructure. The major issue was determining how clients interact with one another. Key to this interaction is how to model concurrency and how to allow multiple clients to access shared data. These issues are addressed below.

6.2.1 Enforcing Correctness of Interleaved Traces

The interleaved traces must satisfy the following two invariants to provide semantic consistency. *Semantic consistency* means that no reference to an object occurs before an object is allocated or after an object is deleted in the final interleaved trace. The invariants are as follows:

- **Invariant I1:** Each client's operations must be executable with any transaction-granularity interleaving of the other client's operations (including possible retries of aborted transactions) with the result being semantically consistent.
- **Invariant I2:** Each client's operations within a transaction must be replayable, with the result being semantically consistent.

Invariant I1, in conjunction with a concurrency control policy, allows the interleaving of transactions of the single-user workloads by the trace merger such that they form a serializable schedule. This interleaving in turn produces a correct trace from the events of the participating transactions. Invariant I2 allows transactions of the single-user workloads that are aborted by the trace merger due to lock conflicts to be retried until they eventually complete successfully.

6.2.2 Data Sharing

There are two types of updates that must be considered when determining the amount of sharing that can occur among clients. The first type of update is applied to non-pointer data members. This kind of update only requires that a locking mechanism prevent interference from conflicting clients. The second type of update is a modification to the structure of a persistent store, such as a pointer overwrite. Within the design of the PTF trace, pointers are treated as edges of a graph. The object referenced by the pointer is then manipulated by its object identifier. With a pointer overwrite, the object referenced by a given edge changes.

Within our framework, a persistent store can be partitioned into regions. Regions that are accessed by only one client are referred to as *private regions* and those that are accessed by many clients as *shared regions*. When updates to edges are allowed in shared regions of the persistent store, all clients must be made aware of these updates since the single-user traces are generated sequentially. Thus, a barrier synchronization primitive is necessary to prevent interfering clients from traversing or even holding pointers into the shared region while the update occurs. This primitive creates a global ordering among the single-user workloads. Such an ordering is contrary to our goal of minimizing the dependencies between operations of the single-user workloads. Therefore,

a synchronization barrier is not supported in the implementation of PTF and structural changes are restricted to private regions (e.g, regions that are accessible by only one client).

6.2.3 Concurrency

Capturing the behavior of a concurrent environment is desired in order to study its effect on the performance of a persistent object system. As a result, several approaches have been designed to achieve concurrency in experimental studies of persistent object systems. These approaches range from actually generating data using a persistent object system under a client-server architecture as in the work of Amsaleg et al. [2] to generating concurrency synthetically as in the work of Maheshwari and Liskov [30] and the work of Schreiber [41].

In our instrumentation infrastructure, one of the goals as stated earlier is to create a multi-user environment independently of a persistent object system. In order to achieve this goal, we modeled concurrency through simulating a multi-user environment. Below, we describe some approaches other researchers have used to model concurrency followed by the approaches that we considered when designing the multi-user workload infrastructure.

Several researchers have modeled concurrency in a manner that prevents clients from interfering, thus avoiding issues related to locking conflicts and serialization of transactions. Maheshwari and Liskov [30] simulate the effect of concurrent applications through random selection of portions of the persistent store to create and modify objects. Schreiber [41] simulated the effect of concurrency within applications by running concurrent transactions on a persistent store that consisted of multiple structurally identical sections. Thus each client had its own copy of the application's persistent store.

Our design takes on the challenge of managing the processing of clients that may interfere with each other. As part of our design process, the following approaches to modeling or generating concurrency were considered.

One approach is to guarantee that there is no interference among clients accessing the persistent store. In this approach, at the collection phase, a stream of events can be generated by executing the workload of each client in sequential order. At the processing phase, a simulator can be used to create concurrency. With this approach, the collection of the data can be done independent of a lock manager, leaving locking and concurrency control to the processing phase of the trace-driven simulation. The advantage of this approach is that experiments can be done on different concurrency control policies using the same trace as input. The downside of this approach is that workloads are constructed with no interference among clients, which is not the case in real application workloads.

Another approach is to specify the kinds of operations that are going to be manipulated in an abstract way. For instance, one client might be allowed to produce ten reads while another client might be allowed an arbitrary combination of reads and writes. In order to apply this approach, the queries and traversals of an application must be thoroughly inspected to abstract information about the number of reads and writes. One advantage of this approach is that the synthetic operations would be generic. Another advantage of this approach is that the developer could have more control over the interactions between operations since the operations are modeled at a high level. A disadvantage of this approach is that a query language would have to be designed and implemented to support the creation of these synthetic operations.

A third approach is to execute the workload of clients using a lock manager to enforce concurrency control. The advantage of this approach is that no restrictions are placed on conflicting transactions among clients. The downside of this approach is that the traces are not generic; in

other words, they contain information specific to the concurrency control policy that was used to generate the traces, thus introducing a dependency between the collecting and processing phases of the traces. This approach would also require us to include a mechanism to support the logging of data values as the transactions were executing so that updates to data would be applied properly. To adequately support the third approach, we would end up implementing a lock manager and a transaction manager. Thus, if we selected this approach, we would have to implement one or more components of a particular persistent object system.

A fourth approach, which is a variation of the first approach, allows limited interference among clients. Limited interference means that clients will share access to the persistent store in a controlled way. At the application level, events generated by operations are enclosed in events that identify the boundaries of transactions. During the processing phase, a concurrency control protocol is applied to the single-user workloads to simulate execution in a multi-user environment. The advantage of this approach, as in the first approach, is that the traces are independent of a concurrency control policy and thus can be used with greater flexibility. Another advantage of this approach is that some sharing is allowed among clients. After considering the alternatives described above, the fourth approach was adopted in our implementation of the multi-user workload instrumentation infrastructure.

6.3 PTF Extensions

In order to support the generation of a multi-user workload, PTF was enhanced to capture the structure of a flat transaction. According to the ODMG standard [11], a transaction is a unit of logic that guarantees the ACID properties. At the application level, this unit of logic consists of a group of operations that are atomic, that is, the effect of the operations is visible to the persistent store only if the transaction finishes.

The structure of a flat transaction consists of the operations that indicate its boundaries and simple actions that manipulate the persistent store. In order to capture the structure of a flat transaction within the trace format, there must be events to represent the operations of the transactions (i.e., begin, commit, abort, and checkpoint). For example, using the `transaction start` and the `transaction end` events as brackets allows one to capture the effects of a transaction without placing any restriction on the concurrency control model that is used to mediate concurrent access to objects of the persistent store.

The simple actions of the transaction are represented by the PTF events that manipulate objects of the persistent store. To correlate these events with a transaction, each of the other PTF events (excluding the format object event), contains two new parameters, `TransId` and `ClientId`. These new parameters have been added to the front end of the list of parameters for each event type. The parameters of each trace event with the additional parameters to support transactions are shown in the Appendix A. Below, the trace formats for the events that represent the operations of a transaction are described.

6.3.1 Trace Formats for Transaction Events

The trace formats for the transaction events are as follows:

Transaction Start

Event: `<transaction_start>`

Parameters:	(int)	TransId	transaction identifier
	(int)	ClientId	logical client identifier

The `transaction start` event demarcates the beginning of a new transaction. A transaction identifier is assigned to the transaction and recorded in the parameter TransId. The client number is also recorded in the ClientId.

Transaction CheckPoint

Event:	`<transaction_checkpoint>`		
Parameters:	(int)	TransId	transaction identifier
	(int)	ClientId	logical client identifier

The `checkpoint` event directs the transaction manager to commit the updates of the transaction without requesting that the lock manager release the locks acquired during the execution of the transaction.

Transaction Abort

Event:	`<transaction_abort>`		
Parameters:	(int)	TransId	transaction identifier
	(int)	ClientId	logical client identifier

This event indicates that the updates should be aborted and the locks associated with this transaction should be released. It also indicates that the abort operation was issued at the application level.

Transaction End

Event:	`<transaction_end>`		
Parameters:	(int)	TransId	transaction identifier
	(int)	ClientId	logical client identifier

The `transaction end` event indicates that the transaction indicated by the field `TransId` has successfully completed and thus all updates of the transaction are eligible to be committed.

6.4 Generation of the Workload

There is very little variation in the process of generating single-user workloads from that discussed in Chapter 4. In this section, we describe modifications to AMPS to support the generation of single-user workloads that can later be processed under a transaction model by modeling the structure of a transaction within the application. We then describe how to generate several single-user workloads

using the TCL interpreter of the AMPS toolkit. We use the binary tree example of Chapter 4 to illustrate how to generate several single-user workloads.

6.4.1 Enhancement to AMPS Toolkit

As described in Section 4.2, the AMPS toolkit consists of a collection of C++ classes for modeling objects of the persistent store and creating traversals. To capture the structure of a transaction, a new class is added to the collection of C++ classes, the class `TransactionGenerator`. The specification of `TransactionGenerator` is shown in Figure 6.2. The constructor takes as input the client identifier and assigns a unique transaction identifier to the transaction generator object. The method `transactionStart` records the transaction start event in the trace file. The method `transactionEnd` records the transaction end event in the trace file. The abort transaction event is recorded in the trace file by the method `transactionAbort`. The method `transactionCheckPoint` records the transaction checkpoint event in the trace file. The method `getTransactionId` returns the transaction identifier of the instance of the `TransactionGenerator` class.

The other C++ classes of the AMPS toolkit have been modified to generate the new version of PTF as described in the previous section. The modifications consist of adding two new parameters to the methods of the `GraphNode` class and the constructors of the traversal classes, `transId` to represent the transaction identifier and `clientId` to represent the client identifier.

6.4.2 Implementing Several Single-user Workloads

The persistent store is implemented as described in Chapter 4 to generate the single-user workloads. The class specifications for each object of the schema must include the input parameters `transId` and `clientId` within each of its methods that manipulates the persistent object store. The class specifications of the operations of the application must also include these two input parameters as shown in the apply method of the class representing the operation `RemoveLeafNodes`, of Figure 6.3. The `RemoveLeafNodes` operation removes the leaf nodes of the binary tree persistent store and then replaces them with newly created nodes. The `RemoveLeafNodes` is used with the `DepthFirst` traversal of the AMPS library. As shown in Figure 6.3, the values of `transId` and the `clientId` are used as input to the methods of the `BinTreeNode` class that are invoked in the apply operation.

We now consider how to generate multiple single-user workloads using the TCL interpreter. As described in Chapter 4, it is assumed that the developer has some idea of the workloads that are to make up the single-user workloads of a multi-user workload. Prior to scripting the workloads, the developer must implement the operations of application as TCL commands. Once all the desired commands are implemented, several combinations of these commands can be used to generate single-user workloads using the TCL interpreter.

In Figure 6.4, we illustrate how to script several single-user workloads using AMPS. In this example, there are two clients and we generate a single-user workload for each client. We begin by assigning a number to the first client and then invoking the TCL command `TG_TransactionBegin`. The command `TG_TransactionBegin` creates a `TransactionGenerator` and then invokes the method `getTransactionId` to return the transaction identifier as its result. The workload of the first client consist of two transactions. The first transaction consists of an operation that creates a seven-node binary tree persistent store. After all of the operations of the transaction have been executed, the TCL command `TG_TransactionEnd` is invoked. The command `TG_TransactionEnd` invokes the method `transactionEnd` of the `TransactionGenerator` class followed by its destructor. The second transaction of the workload contains a breadth-first traversal that performs the operation `ReadXValue` as it visits each node of the binary tree. Although all transactions of this example

```
class TransactionGenerator {

   public:
     TransactionGenerator(int traceFd, int clientNum);
     void transactionStart();
     void transactionEnd();
     void transactionAbort();
     void transactionCheckPoint();
     int getTransactionId();
     ~TransactionGenerator();

   private:
     int traceFileFd;
     int clientNumber;
     static int currentTrId;

     ...
};
```

Figure 6.2: C++ Class Specification of `TransactionGenerator`.

only contain one operation, a transaction can consist of multiple operations. Upon generating the workload of the first client, the trace file associated with that client is closed. The workload of the second client is generated in a similar manner as shown in Figure 6.4.

6.5 Scenarios

In this section, scenarios are presented to show how single-user workload traces are interleaved. Each scenario was chosen to demonstrate a specific point. To aid in the illustration, traces are shown from the execution of a simple persistent object application that was introduced in Chapter 3. The persistent store consists of a binary tree whose nodes consist of an integer value. The operations of this application are as follows: `CreateTree`, a create operation that builds the binary tree structure, `UpdateXValue`, a breadthfirst traversal that updates the integer value of each node of the tree, `ReadXValue`, a breadth-first traversal that reads the integer value of each node of the tree, and `RemoveLeafNodes`, a depth-first traversal that creates new leaf nodes for the binary tree.

6.5.1 Scenario 1: Treatment of Creation of Objects

In this scenario, there are two clients, Client 1 and Client 2, that are manipulating the persistent store. The first client creates the persistent store and then references the integer value at each node, and the second client updates the integer value of each node. Table 6.1 illustrates the traces that would be generated for Client 1 and Client 2 along with a possible interleaving of their workloads. The first field of each trace event is the transaction identifier. The workload of Client 1 consists of two transactions, T1 and T2, and the workload of Client 2 consists of one transaction T3.

The trace merger receives a list of clients that participate in the creation of the multi-user

```
class RemoveLeafNodes: public TraverseOption {

    ...

    // operation performed at each visited node of the Traversal
    int apply(int transId, int clientId, GraphNodePtr currentNode) {
        int noOfEdges;
        static GraphNodePtr parentNode;
        static int edgeCount = 0;
        BinTreeNode *newChild;
        int edges = 0;

        noOfEdges = (* currentNode).getNoOfAdjEdges( );

        if (noOfEdges != 0)
        {
            // processing the non-leaf nodes

            parentNode = currentNode;
        }
        else
        {
            //  processing the leaf nodes
            //  typeNode is an attribute the RemoveLeafNodes Class

            newChild = new BinTreeNode(transId, clientId, typeNode, edges, NULL);
            (* newChild).setValue(transId, clientId, 1, 1);

            if (edgeCount == 0)
            {
                // got the left child
                (* newChild).setLeftChild(transId, clientId, parentNode);
                edgeCount = 1;
            }
            else
            {
               // got the right child
               (* newChild).setRightChild(transId, clientId, parentNode);
                edgeCount = 0;
            }
        }

        return (0);
    }

};
```

Figure 6.3: Implementation of the Apply Method of the RemoveLeafNodes Class.

```
sheriff% TGenApp
   % TG_OpenTraceFile Client1Trace
     Trace file opened
   % TG_SetClientId 1
   % TG_TransactionBegin
     1
   % TG_DBbuild 3
     Database generated
   % TG_TransactionEnd 1
   % TG_TransactionBegin
     2
   % TG_Traversalbuild breadthfirst ReadXValue
     Traversal processing completed
   % TG_TransactionEnd 2
   % TG_CloseTraceFile
     Trace file closed
   % TG_OpenTraceFile Client2Trace
     Trace file opened
   % TG_SetClientId 2
   % TG_TransactionBegin
     3
   % TG_Traversalbuild depthfirst pre RemoveLeafNodes
     Traversal processing completed
   % TG_TransactionEnd 3
   % TG_CloseTraceFile
     Trace file closed
   % exit
sheriff%
```

Figure 6.4: Scripting a Multi-user Workload Using AMPS and TCL.

workload. Suppose that Client 2 appears first in the list. Using a round-robin scheduling algorithm, Client 2 is selected for processing. In order for the transaction T3 of Client 2 to execute, the persistent store has to exist. Since the persistent store has not been created at this point, transaction T3 cannot run. The trace merger creates a synchronization barrier to prevent transactions from running until the objects manipulated by the transaction are available in the persistent store. Thus, Client 1 will be selected for processing. Now, suppose that the time slice only allows for the processing of T1. T1 is processed and Client 1 is placed back on the ready queue. At this point, Client 2 is processed. Since the persistent store has been created, T3 can now be processed. Upon completion of T3, the workload of Client 2 is completely processed so Client 1 will be processed to completion. Thus, the order of the transactions is the following:

- BOT(T1) EOT(T1) BOT(T3) EOT(T3) BOT(T2) EOT(T2)

where BOT stands for *beginning of transaction* and EOT stands for the *end of transaction.* The resulting interleaved trace is shown in Table 6.1. The following points should be noted about the interleaved trace.

1. The no garbage collection events `ngs/nge` protect the newly created objects from being collected.

2. As shown in T1, the newly created objects are written to the interleaved trace along with the updates to them as they are encountered in the single-user trace. The fact that create events are processed in this manner does not affect the consistency of the trace or the persistent store that is created from the trace. If at some point T1 had aborted, the objects created by T1 up to the abort would eventually be garbage collected, making them invisible to any future transactions because they are not attach to the reachability graph. Also, note that the set root event, which attaches the objects to the reachability graph, is the last event written from the processing of T1. This ensures that the newly created objects do not get attached to the reachability graph until T1 successfully completes.

6.5.2 Scenario 2: Example of an Interfering Transaction

The workloads of Client 1 and Client 2 are the same as described above in Scenario 1. However, in this scenario, the duration of the time slice of Client 1 is longer so that T2 is allowed to start processing but not complete. Now, suppose that after processing the event, `er 2 1 41 43 1`, the time slice of Client 1 expired. Client 2 is then selected for processing. The first event of T3 is processed and then a locking conflict occurs. The second event, `dw 3 2 41 42 1`, requires that T2 acquire a write lock on the object represented by the object identifier 42. However, a write lock cannot be granted to T3 because it is already locked in read mode by T2. Thus, Client 2 is placed in a wait state until T2 committed. The execution order of the transactions is the following:

- BOT(T1) EOT(T1) BOT(T2) BOT(T3) EOT(T2) EOT(T3)

The interleaved trace is shown in Table 6.2.

Table 6.1: Example of an Interleaving of the Workload of Two Clients.

Client 1 Trace	Client 2 Trace	Interleaved Trace
ts 1 1	ts 3 2	ngs 1 1
co 1 1 41 42	gr 3 2	co 1 1 41 42
dw 1 1 41 42 1	dw 3 2 41 42 1	dw 1 1 41 42 1
sr 1 1 41 42	er 3 2 41 42 0	co 1 1 41 43
co 1 1 41 43	er 3 2 41 42 1	dw 1 1 41 43 1
dw 1 1 41 43 1	dw 3 2 41 43 1	ew 1 1 41 42 0 43
ew 1 1 41 42 0 43	er 3 2 41 43 0	co 1 1 41 44
co 1 1 41 44	er 3 2 41 43 1	dw 1 1 41 44 1
dw 1 1 41 44 1	dw 3 2 41 44 1	ew 1 1 41 42 1 44
ew 1 1 41 42 1 44	er 3 2 41 44 0	co 1 1 41 45
co 1 1 41 45	er 3 2 41 44 1	dw 1 1 41 45 1
dw 1 1 41 45 1	dw 3 2 41 45 1	ew 1 1 41 43 0 45
ew 1 1 41 43 0 45	dw 3 2 41 46 1	co 1 1 41 46
co 1 1 41 46	dw 3 2 41 47 1	dw 1 1 41 46 1
dw 1 1 41 46 1	dw 3 2 41 48 1	ew 1 1 41 43 1 46
ew 1 1 41 43 1 46	te 3 2	co 1 1 41 47
co 1 1 41 47		dw 1 1 41 47 1
dw 1 1 41 47 1		ew 1 1 41 44 0 47
ew 1 1 41 44 0 47		co 1 1 41 48
co 1 1 41 48		dw 1 1 41 48 1
dw 1 1 41 48 1		ew 1 1 41 44 1 48
ew 1 1 41 44 1 8		sr 1 1 41 42
te 1 1		nge 1 1
ts 2 1		gr 3 2
gr 2 1		er 3 2 41 42 0
dr 2 1 41 42 1		er 3 2 41 42 1
er 2 1 41 42 0		er 3 2 41 43 0
er 2 1 41 42 1		er 3 2 42 43 1
dr 2 1 41 43 1		er 3 2 41 44 0
er 2 1 41 43 0		er 3 2 41 44 1
er 2 1 41 43 1		ngs 3 2
dr 2 1 41 44 1		dw 3 2 41 42 1
er 2 1 41 44 0		dw 3 2 41 43 1
er 2 1 41 44 1		dw 3 2 41 44 1
dr 2 1 41 45 1		dw 3 2 41 45 1
dr 2 1 41 46 1		dw 3 2 41 46 1
dr 2 1 41 47 1		dw 3 2 41 47 1
dr 2 1 41 48 1		dw 3 2 41 48 1
te 2 1		nge 3 2
		gr 2 1
		dr 2 1 41 42 1
		er 2 1 41 42 0
		er 2 1 41 42 1
		dr 2 1 41 43 1
		er 2 1 41 43 0
		er 2 1 41 43 1
		dr 2 1 41 44 1
		er 2 1 41 44 0
		er 2 1 41 44 1
		dr 2 1 41 45 1
		dr 2 1 41 46 1
		dr 2 1 41 47 1
		dr 2 1 41 48 1

Table 6.2: Example of an Interleaving as a Result of Conflicting Transactions.

Client 1 Trace	Client 2 Trace	Interleaved Trace
ts 1 1	ts 3 2	ngs 1 1
co 1 1 41 42	gr 3 2	co 1 1 41 42
dw 1 1 41 42 1	dw 3 2 41 42 1	dw 1 1 41 42 1
sr 1 1 41 42	er 3 2 41 42 0	co 1 1 41 43
co 1 1 41 43	er 3 2 41 42 1	dw 1 1 41 43 1
dw 1 1 41 43 1	dw 3 2 41 43 1	ew 1 1 41 42 0 43
ew 1 1 41 42 0 43	er 3 2 41 43 0	co 1 1 41 44
co 1 1 41 44	er 3 2 41 43 1	dw 1 1 41 44 1
dw 1 1 41 44 1	dw 3 2 41 44 1	ew 1 1 41 42 1 44
ew 1 1 41 42 1 44	er 3 2 41 44 0	co 1 1 41 45
co 1 1 41 45	er 3 2 41 44 1	dw 1 1 41 45 1
dw 1 1 41 45 1	dw 3 2 41 45 1	ew 1 1 41 43 0 45
ew 1 1 41 43 0 45	dw 3 2 41 46 1	co 1 1 41 46
co 1 1 41 46	dw 3 2 41 47 1	dw 1 1 41 46 1
dw 1 1 41 46 1	dw 3 2 41 48 1	ew 1 1 41 43 1 46
ew 1 1 41 43 1 46	te 3 2	co 1 1 41 47
co 1 1 41 47		dw 1 1 41 47 1
dw 1 1 41 47 1		ew 1 1 41 44 0 47
ew 1 1 41 44 0 47		co 1 1 41 48
co 1 1 41 48		dw 1 1 41 48 1
dw 1 1 41 48 1		ew 1 1 41 44 1 48
ew 1 1 41 44 1 48		sr 1 1 41 42
te 1 1		nge 1 1
ts 2 1		gr 2 1
gr 2 1		dr 2 1 41 42 0
dr 2 1 41 42 1		er 2 1 41 42 0
er 2 1 41 42 0		er 2 1 41 42 1
er 2 1 41 42 1		dr 2 1 41 43 1
dr 2 1 41 43 1		er 2 1 41 43 0
er 2 1 41 43 0		er 2 1 41 43 1
er 2 1 41 43 1		gr 3 2
dr 2 1 41 44 1		dr 2 1 41 44 1
er 2 1 41 44 0		er 2 1 41 44 0
er 2 1 41 44 1		er 2 1 41 44 1
dr 2 1 41 45 1		dr 2 1 41 45 1
dr 2 1 41 46 1		dr 2 1 41 46 1
dr 2 1 41 47 1		dr 2 1 41 47 1
dr 2 1 41 48 1		dr 2 1 41 48 1
te 2 1		er 3 2 41 42 0
		er 3 2 41 42 1
		er 3 2 41 43 0
		er 3 2 41 43 1
		er 3 2 41 44 0
		er 3 2 41 44 1
		ngs 3 2
		dw 3 2 41 42 1
		dw 3 2 41 43 1
		dw 3 2 41 44 1
		dw 3 2 41 45 1
		dw 3 2 41 46 1
		dw 3 2 41 47 1
		dw 3 2 41 48 1
		nge 3 2

6.5.3 Scenario 3: An Example of a Non-Conflicting Transaction

The workload of Client 2 now consists of the ReadXValue traversal. Thus, transaction T3 consists of the events shown in Table 6.3. As in Scenario 2, the duration of the time slice of Client 1 is longer so that T2 is allowed to start processing but not complete. Now, suppose that after processing the event, `er 2 1 41 43 1`, the time slice of Client 1 expired. Client 2 is selected for processing. Since T3 does not conflict with T2, it is processed to completion. The execution order of the transactions is the following:

- BOT(T1) EOT(T1) BOT(T2) BOT(T3) EOT(T3) EOT(T2)

The interleaved trace is shown in Table 6.3.

6.5.4 Scenario 4: Example of Structural Change to Shared Region

The workload of Client 2 consists of the operation RemoveLeafNodes. The events of transaction T3 are shown in Table 6.4. In this scenario, the execution order of the transactions is the following:

- BOT(T1) EOT(T1) BOT(T3) EOT(T3) BOT(T2) EOT(T2)

The resulting interleaved trace is shown in Table 6.4. In T3, the objects associated with object identifiers, 45, 46, 47, and 48, were replaced with objects whose object identifiers are 49, 50, 51, and 52, respectively. Since the workload of Client 1 was generated prior to the the workload of Client 2, the interleaving results in a semantically inconsistent trace. Therefore, structural modifications to shared regions of the persistent store are not allowed.

6.6 An Example Trace Merger

In this section, we present an example implementation of the trace merger. The trace merger simulates the execution of the single-user workloads in a multi-client environment. Before describing the implementation of the trace merger, we discuss the intuition behind the design of the trace merger.

6.6.1 The Notion of Time

Clients normally execute their workloads at varying rates. This may be due to some idle time because some data are being examined interactively or because the processor on which one client is executing is slower than another client's processor. Regardless of the reason, there is a need to simulate varying rates of execution among clients. Thus, the term *tick* represents the rate of execution of a client. A tick is equivalent to one or more executed events. Thus, one client whose tick rate is 1 event might process N events during a time slice while a slower client whose tick rate is 2 might process $N/2$ events during its time slice (e.g., the higher the tick rate the slower the client is running).

6.6.2 Synchronization Algorithm

In most traditional database management systems as well as some of the older persistent object systems (e.g., Exodus), serialization of concurrent transactions is enforced using the two-phase locking protocol. The two-phase locking protocol is defined as follows [19]:

Table 6.3: Example of an Interleaving of Two Non-Conflicting Transactions.

Client 1 Trace	Client 2 Trace	Interleaved Trace
ts 1 1	ts 3 2	ngs 1 1
co 1 1 41 42	gr 3 2	co 1 1 41 42
dw 1 1 41 42 1	dr 3 2 41 42 1	dw 1 1 41 42 1
sr 1 1 41 42	er 3 2 41 42 0	co 1 1 41 43
co 1 1 41 43	er 3 2 41 42 1	dw 1 1 41 43 1
dw 1 1 41 43 1	dr 3 2 41 43 1	ew 1 1 41 42 0 43
ew 1 1 41 42 0 43	er 3 2 41 43 0	co 1 1 41 44
co 1 1 41 44	er 3 2 41 43 1	dw 1 1 41 44 1
dw 1 1 41 44 1	dr 3 2 41 44 1	ew 1 1 41 42 1 44
ew 1 1 41 42 1 44	er 3 2 41 44 0	co 1 1 41 45
co 1 1 41 45	er 3 2 41 44 1	dw 1 1 41 45 1
dw 1 1 41 45 1	dr 3 2 41 45 1	ew 1 1 41 43 0 45
ew 1 1 41 43 0 45	dr 3 2 41 46 1	co 1 1 41 46
co 1 1 41 46	dr 3 2 41 47 1	dw 1 1 41 46 1
dw 1 1 41 46 1	dr 3 2 41 48 1	ew 1 1 41 43 1 46
ew 1 1 41 43 1 46	te 3 2	co 1 1 41 47
co 1 1 41 47		dw 1 1 41 47 1
dw 1 1 41 47 1		ew 1 1 41 44 0 47
ew 1 1 41 44 0 47		co 1 1 41 48
co 1 1 41 48		dw 1 1 41 48 1
dw 1 1 41 48 1		ew 1 1 41 44 1 48
ew 1 1 41 44 1 48		sr 1 1 41 42
te 1 1		nge 1 1
ts 2 1		gr 2 1
gr 2 1		dr 2 1 41 42 1
dr 2 1 41 42 1		er 2 1 41 42 0
er 2 1 41 42 0		er 2 1 41 42 1
er 2 1 41 42 1		dr 2 1 41 43 1
dr 2 1 41 43 1		er 2 1 41 43 0
er 2 1 41 43 0		er 2 1 41 43 1
er 2 1 41 43 1		gr 3 2
dr 2 1 41 44 1		dr 3 2 41 42 1
er 2 1 41 44 0		er 3 2 41 42 0
er 2 1 41 44 1		er 3 2 41 42 1
dr 2 1 41 45 1		dr 3 2 41 43 1
dr 2 1 41 46 1		er 3 2 41 43 0
dr 2 1 41 47 1		er 3 2 41 43 1
dr 2 1 41 48 1		dr 3 2 41 44 1
te 2 1		er 3 2 41 44 0
		er 3 2 41 44 1
		dr 3 2 41 45 1
		dr 3 2 41 46 1
		dr 3 2 41 47 1
		dr 3 2 41 48 1
		dr 3 2 41 44 1
		er 2 1 41 44 0
		er 2 1 41 44 1
		dr 2 1 41 45 1
		dr 2 1 41 46 1
		dr 2 1 41 46 1
		dr 2 1 41 47 1
		dr 2 1 41 48 1

Table 6.4: Example of a Semantically Inconsistent Interleaved Trace.

Client 1 Trace	Client 2 Trace	Interleaved Trace	
ts 1 1	ts 3 2	ngs 1 1	gr 2 1
co 1 1 41 42	gr 3 2	co 1 1 41 42	dr 2 1 41 42 1
dw 1 1 41 42 1	er 3 2 41 42 0	dw 1 1 41 42 1	er 2 1 41 42 0
sr 1 1 41 42	er 3 2 41 43 0	co 1 1 41 43	er 2 1 41 42 1
co 1 1 41 43	co 3 2 41 49	dw 1 1 41 43 1	dr 2 1 41 43 1
dw 1 1 41 43 1	dw 3 2 41 49	ew 1 1 41 42 0 43	er 2 1 41 43 0
ew 1 1 41 42 0 43	ew 3 2 41 43 0 49	co 1 1 41 44	er 2 1 41 43 1
co 1 1 41 44	er 3 2 41 43 1	dw 1 1 41 44 1	dr 2 1 41 44 1
dw 1 1 41 44 1	co 3 2 41 50	ew 1 1 41 42 1 44	er 2 1 41 44 0
ew 1 1 41 42 1 44	dw 3 2 41 50 1	co 1 1 41 45	er 2 1 41 44 1
co 1 1 41 45	ew 3 2 41 43 1 50	dw 1 1 41 45 1	dr 2 1 41 45 1
dw 1 1 41 45 1	er 3 2 41 42 1	ew 1 1 41 43 0 45	dr 2 1 41 46 1
ew 1 1 41 43 0 45	er 3 2 41 44 0	co 1 1 41 46	dr 2 1 41 47 1
co 1 1 41 46	co 3 2 41 51	dw 1 1 41 46 1	dr 2 1 41 48 1
dw 1 1 41 46 1	dw 3 2 41 51 1	ew 1 1 41 43 1 46	
ew 1 1 41 43 1 46	ew 3 2 41 44 0 51	co 1 1 41 47	
co 1 1 41 47	er 3 2 41 44 1	dw 1 1 41 47 1	
dw 1 1 41 47 1	co 3 2 41 52	ew 1 1 41 44 0 47	
ew 1 1 41 44 0 47	dw 3 2 41 52 1	co 1 1 41 48	
co 1 1 41 48	ew 3 2 41 44 1 52	dw 1 1 41 48 1	
dw 1 1 41 48 1	te 3 2	ew 1 1 41 44 1 48	
ew 1 1 41 44 1 48		sr 1 1 41 42	
te 1 1		nge 1 1	
ts 2 1		gr 3 2	
gr 2 1		er 3 2 41 42 0	
dr 2 1 41 42 1		er 3 2 41 43 0	
er 2 1 41 42 0		ngs 3 2	
er 2 1 41 42 1		co 3 2 41 49	
dr 2 1 41 43 1		dw 3 2 41 49 1	
er 2 1 41 43 0		ew 3 2 41 43 0 49	
er 2 1 41 43 1		er 3 2 41 43 1	
dr 2 1 41 44 1		co 3 2 41 50	
er 2 1 41 44 0		dw 3 2 41 50 1	
er 2 1 41 44 1		ew 3 2 41 43 1 50	
dr 2 1 41 45 1		er 3 2 41 42 1	
dr 2 1 41 46 1		er 3 2 41 44 0	
dr 2 1 41 47 1		co 3 2 41 51	
dr 2 1 41 48 1		dw 3 2 41 51 1	
te 2 1		ew 3 2 41 44 0 51	
		er 3 2 41 44 1	
		co 3 2 41 52	
		dw 3 2 41 52 1	
		ew 3 2 41 44 1 52	
		nge 3 2	

1. Before operating on any object, a transaction must acquire a lock on that object.
2. After releasing a lock, a transaction must never go on to acquire any more locks.

A transaction following this protocol thus has two phases. In the first phase, the locks required by the transaction are acquired. The second phase is completed in an atomic operation of the transaction, that is, either a commit operation or a rollback operation.

Within our prototype of the trace merger, the two-phase locking protocol is implemented. When a transaction is encountered, a lock request is issued for each object manipulated within the transactions. Upon acquiring locks for all objects, the transaction is committed by copying all update events (e.g., edge write events and data write events) to the interleaved trace.

There can be a variety of locking granularities associated with locks. For instance, an example of coarse granularity is the entire persistent store or a page of the persistent store. With the notion of object-oriented data management systems, data can be viewed as objects and thus the granularity of a lock can be in terms of an object. Access to an object can be in terms of shared access and exclusive access (e.g., read access and write access, respectively). At the level of the trace, manipulations to objects can be identified by the trace event type. These events can be classified as either a read access or a write access. Thus, we have chosen only to support read and write locking. We looked at the possibility of supporting update locks to avoid the common deadlock problem of two processes requesting a read lock on an object and then later requesting a write lock on the same object to update it. However, we did not implement the update lock mode.

6.6.3 Garbage Collector Interaction

In analyzing garbage collection algorithms, in the context of a multi-user environment, the following two invariants have been identified by Amsaleg et al. [2]:

1. When a transaction cuts a reference to an object, the object is not eligible for reclamation until the first garbage collection that is initiated after the completion (i.e., commit or abort) of that transaction.
2. Objects that are created by a transaction are not eligible for reclamation until the first garbage collection that is initiated after the completion of that transaction.

These invariants help to protect objects that have been disconnected and reconnected to the reachability graph as well as newly created objects. The interleaved traces produced by the trace merger must prevent premature garbage collection of objects. The garbage collection events, `no garbage collection begin` and the `no garbage collection end`, are incorporated into the interleaved trace produced by the trace merger to protect objects that cannot participate in a garbage collection. These events are used as "weak transaction begin/end" indicators. Thus, if a garbage collector is executing, it cannot collect partitions that contain objects protected by the `no garbage collection` events.

In addition to the above, Amsaleg et al. [2] found that the two-phase locking protocol guaranteed the correctness of the persistent store in the presence of a garbage collector that ran as an independent process without holding locks. That is, all accesses to pointers in the persistent store are performed under strict two-phase locking and this prevents the garbage collector from reclaiming objects prematurely.

6.6.4 Overview of Implementation

The current implementation of the trace merger is a proof-of-concept implementation. It supports only round-robin scheduling of the pseudo-processes representing clients manipulating a persistent object store. It also supports only two-phase locking. The implementation supports the insertion of alternative transaction models, concurrency control models, and scheduling algorithms. Below, we present an overview of the trace merger implementation.

Architecture of the Trace Merger

Figure 6.5 presents the architecture of the trace merger. As the diagram shows, the trace merger consists of several components: a Main, a Scheduler, a Swapper, a ClientCoordinator, and a LockManager. We consider Figure 6.5 from top to bottom. The file `Workload.asc` is created by the user and contains the number of clients participating in the generation of the workload, the names of the trace files generated from single-user workloads, a client number for each client, and the tick rate at which clients are to execute. It also contains the maximum number of events that make up a time slice. This file is used as input to the Main executable. `Main` creates a ClientCoordinator, serving as a process in the simulation, for each client participating in the generation of the multi-user workload. The ClientCoordinator keeps track of the state of the pseudo-process (e.g., state of current transaction, locks owned by the transaction, state of the process, ready or blocked). Associated with each ClientCoordinator is the trace file that was generated while executing the workload in a single-user environment. `Main` creates a list of the ClientCoordinators and invokes the Scheduler. The Scheduler selects an eligible client to be processed and keeps track of clients whose workload has been completely processed.

Upon selecting a client to be processed, the Scheduler invokes the Swapper executable. The Swapper also serves as the Transaction Manager. The events of the traces are processed by their type. When a transaction start event is encounter, all events until the transaction end event or the transaction abort event are processed as part of the transaction. The Swapper requests a lock for each event of the transaction that manipulates an object (e.g., events such as a read data event, a read edge event, a write data event, or a write edge event). Once a transaction commits, the Swapper copies all update events to the interleaved trace.

The LockManager grants a lock request made by the transaction manager if there is no conflict. If a lock request conflicts with an existing lock request but does not cause deadlock, the LockManager delays the lock request. Requests that may end up in deadlock are denied by the LockManager. The LockManager returns the status of the lock request to the Swapper. Depending on the status of the lock, the Swapper will either continue processing its current pseudo-process or discontinue processing it by placing it in a wait state.

The Swapper updates the state of the ClientCoordinator when transactions are encountered during the processing of the trace file associated with the ClientCoordinator. Upon completion of a time slice, the Swapper again updates the state of the ClientCoordinator and returns to the Scheduler to get its new pseudo-process.

Below, we discuss the following activities of the trace merger in more detail: scheduling a process, transaction processing, and conflict detection by the lock manager.

Selecting a ClientCoordinator for Processing

The scheduler uses three criteria in selecting a ClientCoordinator to be processed. These criteria are as follows:

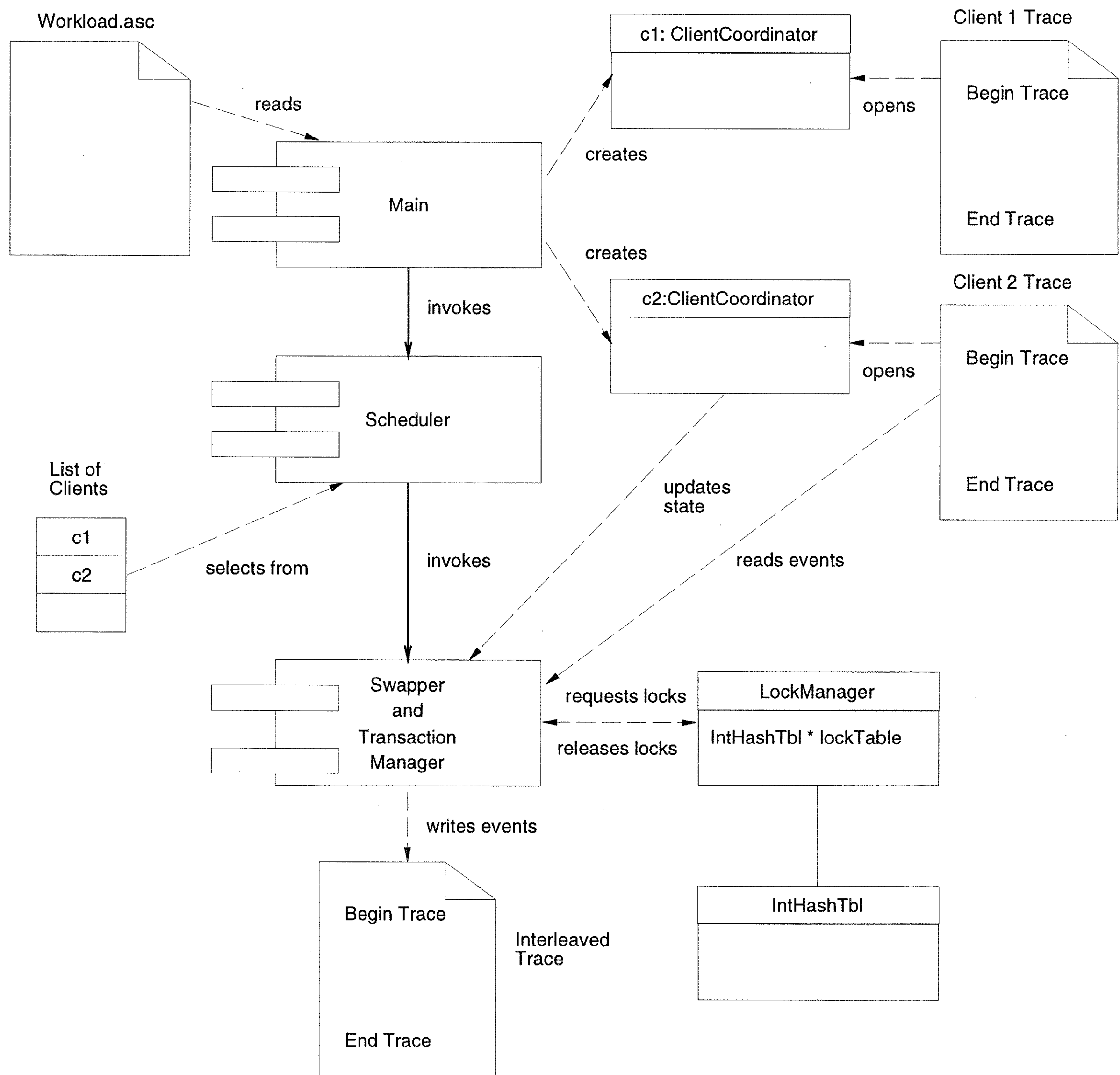


Figure 6.5: Architecture of the Trace Merger.

- *The order of the ClientCoordinator in the original list that was sent to the Scheduler.* Thus, the ClientCoordinator is first selected based on a round-robin algorithm.
- *The runable state of the ClientCoordinator.* The ClientCoordinator may be in a ready or a wait state. If the ClientCoordinator is waiting on a lock request, then it can not run until the lock has been granted.
- *The status of the current transaction of the ClientCoordinator.* If the lock manager has indicated that a ClientCoordinator must abort its current transaction and it is not the active ClientCoordinator, the Scheduler will select the ClientCoordinator of the aborted transaction when the Swapper requests a new ClientCoordinator to process.

Once a ClientCoordinator has completed processing, the Scheduler removes it from the list of active ClientCoordinators. Scheduling of ClientCoordinators continues until all are completely processed.

Process of Acquiring Locks

Objects are associated with locks through the use of the logical object identifier of the object. The object identifier serves as the name of the lock. When an operation is applied to an object, a lock must be acquired for the object. For example, the edge read event requires that a lock with a locking mode *read* be acquired for the object that contains the edge.

The lock manager is responsible for determining which lock requests are granted. In order to determine whether a lock request is granted, the lock manager uses the conflict detection algorithm that is illustrated in Figure 6.6. The conflict detection algorithm is a simplified version of the conflict detection algorithm implemented in the operation system for the Wang Laboratories main frames. The first step of this algorithm is to check whether the lock request is compatible with the current mode of the lock associated with the object of the request. To check the request's compatibility, the lock manager compares the mode of the lock request with the existing mode of the lock using the compatibility matrix of Figure 6.6. This compatibility matrix was developed by Daynes [20] in their implementation of a concurrency control model for persistent Java. As shown in Figure 6.6, a compatibility matrix is a two dimensional table. One dimension of the table represents the current mode of the lock associated with an object and the other dimension represents the mode of the lock request. If there is no conflict, the lock request is granted. However, if there is a conflict, the lock manager then checks to see if there is only one owner of the lock and that owner is also the requester of the lock. If this is the case, the lock manager then grants the lock. However, if there is more than one owner or the owner is not equivalent to the requester, the lock manager checks to see if the requester can wait on the lock. In order to determine if the requester can wait on the lock, the lock manager initiates a deadlock detection algorithm. If as a result of waiting, the requester will not cause a deadlock, the requester is allowed to wait. However, if a deadlock will occur, one of the conflicting ClientCoordinators is selected to abort its current transaction.

Once a transaction is committed or aborted, the lock manager removes all of the lock requests associated with the transaction by requesting the list of locks from the ClientCoordinator. If the only owner of the lock is the ClientCoordinator of the committed transaction, the lock is removed from the lock table.

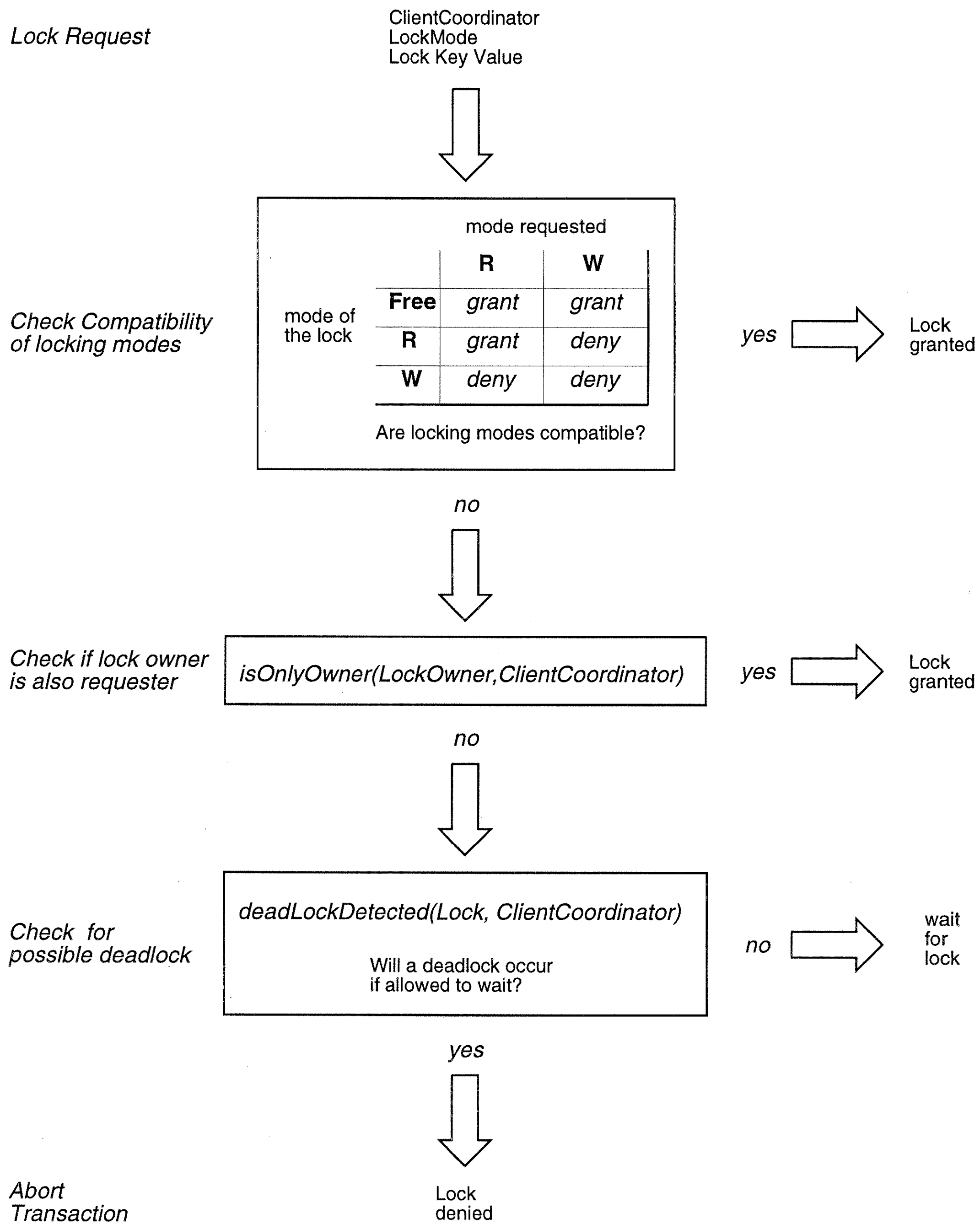


Figure 6.6: The Conflict Detection Algorithm of the Lock Manager.

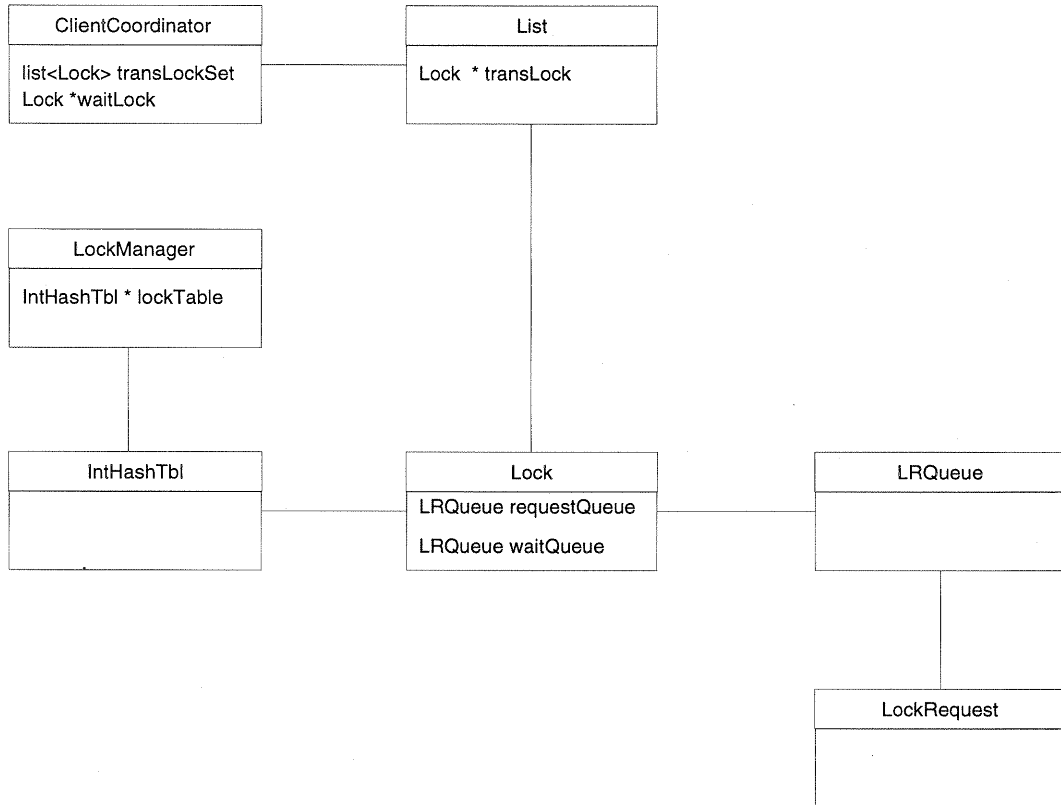


Figure 6.7: Diagram of the Relationships of the Lock Object.

Lock Associations

Several objects of the trace merger are associated with a lock as shown in Figure 6.7. The lock manager maintains a table of locks. These classes and their relationships are based on Gray's presentation of an implementation of a lock manager [25]. Each lock has two lock request queues, a queue of granted requests and a queue of requests that are waiting to be granted access to the lock. If a ClientCoordinator is allowed to wait on a lock, the pointer to the lock is kept until the lock has been granted. All locks requested by an active transaction of a ClientCoordinator are saved in a transaction lock list.

Processing of Aborted Transactions

A transaction may be aborted in two cases. First, the transaction is aborted due a decision by an application of the single-user workload. While processing the trace file of a client, if an abort trace event is encountered, the Swapper initiates the abort processing. If there are any events that caused modifications to the persistent store, these events are discarded. The lock manager then unlocks all of the locks on objects manipulated by the aborted transaction.

A transaction that is aborted by the lock manager due to lock conflicts is retried until it completes successfully. In this case, the lock manager unlocks all of the locks associated with the aborted transaction. Then the events of the transaction are setup to be processed again by the

Swapper.

6.7 Summary

This chapter described a new approach to generating multi-user workloads by executing instrumented applications to generate any number of single-user workloads and then merge them to generate a trace that captures their combined behavior. The applications are instrumented using the AMPS toolkit, described in Chapter 4, and PTF, presented in Chapter 3. The instrumented applications are then executed under the TCL interpreter to generate several traces of single-user workloads. Different combinations of these traces are then merged using a trace merger that combines the traces by simulating the concurrent execution of the clients under a concurrency control policy and a transaction model.

The goals and issues with their resolutions were also discussed. One of the most difficult issues of the design of the instrumentation infrastructure was how to allow sharing of data of the persistent store. With the goal of generating single-user workloads that were not dependent on one another, we chose to limit the type of update operations to those that did not reorganize the structure of the persistent store in regions that were manipulated by more than one client.

The chapter concluded with a description of an example implementation of a trace merger that was written in C++. The trace merger supports flat transactions and implements the two-phase locking protocol.

In the next chapter, we demonstrate the utility of the instrumentation infrastructure described in this chapter through the use of a C++ implementation of the OO7 multi-user benchmark.

Chapter 7

Experience with OO7 Multi-user Workload Specification

As part of this dissertation, a study of the OO7 multi-user benchmark was completed. In this chapter, we describe our experiences using the infrastructure described in Chapter 6 with the OO7 multi-user benchmark as the application.

This chapter begins with a discussion of the design of the persistent store that is used in the multi-user OO7 benchmark followed by a description of the parameterized workload. The second section describes the modifications that were made to operations of the OO7 benchmark to guarantee that the interleaved trace produced by the trace merger is correct semantically. In the third section, we present some preliminary experiments based on single-user workloads that were generated from an instrumented version of the multi-user OO7 implementation. We conclude this chapter with observations made as a result of this study.

7.1 Overview of the Multi-user OO7 Benchmark

The designers of the original OO7 benchmark realized that there was a gap in the information provided by the existing performance studies because very few provided insight into multi-user performance [9]. To address this gap, they developed a multi-user persistent object system benchmark. This benchmark is a customizable benchmark generator that consists of an extended design of the OO7 persistent store and a parameterized workload. By combining the primitives of the workload, a variety of workloads can be generated. The benchmark generates symmetric workloads, which is defined by Carey et al. [9] as a workload in which all the clients behave similarly with respect to accessing data (e.g., all clients generate their workloads with the same set of input parameters). According to Carey et al. [9], asymmetric workloads can also be generated using the parameterized workloads. An example of an asymmetric workload is a workload consisting of two clients in a producer/consumer relationship.

7.1.1 Description of the OO7 Persistent Store

The OO7 multi-user persistent store consists of modules as described in Section 5.1 with a few modifications. In the original design, the base assembly of a module contained two associations: a private association and a shared association. However, the composite parts that were selected for each association were contained in one collection of composite parts. The private composite parts were associated with only one module while the shared composite parts were randomly selected

from the collection of composite parts. Thus, there were no truly private parts. As stated in [9], it (e.g., the design of the persistent store) provided no way to have a client transaction read from its private composite parts and update shared composite parts with out interfering with the private reads of other clients.

To address this problem, the designers of the OO7 multi-user workload removed the shared association from the base assembly and created a *shared* module for shared access. The shared module then introduced another set of composite parts that were referenced by the base assemblies.

A major goal of the designers of the multi-user OO7 persistent store was to design a scalable extension to the single-user OO7 persistent store. In order to achieve this goal, the persistent store contains one private module for each client and the shared module grows proportionately with the number of clients. For each client, the shared module contains a base assembly that consists of 200 composite parts.

In our case study of the OO7 multi-user benchmark, the persistent store is slightly different from that described above. The base assembly of modules still contains a shared association of composite parts as described in the original OO7 benchmark. However, there are two sets of composite parts: a private set and a shared set as shown in Figure 7.1. The composite parts of a private association of a base assembly are assigned from the private set of composite parts. The shared composite parts of a base assembly are randomly assigned from the shared composite part set. As shown in Figure 7.1, the number of composite parts is represented by I where I equals $3/4(N)$ and N is the number of private composite parts. The number of shared composite parts created is almost a $1:1$ ratio to the number of private composite parts. It is slightly less to increase the possibility of contention between clients accessing the shared composite parts.

As described above, the persistent store of our study consists of a private module for each client. However, there is no one entity that is the *shared module* in the persistent store of our study. Instead, it is formed from the shared associations of the base assemblies of all the private modules. As in the original OO7 multi-user specifications of the persistent store, our design of the persistent store is also scalable with the addition of more clients. In other words, the *logical* shared module, consisting of the all the shared composite parts of the base assemblies of each private module as well as their respective atomic parts, grows as the number of clients increases.

Lastly, the persistent store consists of only three connections per atomic part. This is consistent with the specifications of original design of the multi-user OO7 benchmark specification.

7.1.2 Description of the OO7 Multi-User Workloads

The multi-user OO7 workload consists of multiple clients running a series of parameterized transactions. Each transaction consists of a combination of basic operations. These operations can be categorized as follows: private reads, private writes, shared reads, and shared writes. The private operations are operations that access the private modules of the persistent store and the shared operations manipulate the shared module of the persistent store. In the study, the shared operations access the shared base assemblies of the private modules. Thus, a shared operation can access shared data from any of the private modules.

The read-only operation is formed from traversal T1. A composite part of a base assembly is selected and a depth-first traversal of the atomic part subgraph is performed on the selected composite part. The read-write operation is formed from traversal T2b of the original OO7 benchmark. In Traversal T2b, a depth-first traversal similar to T1 is performed on the atomic part subgraph and as each atomic part of the subgraph is visited, the `X` and `Y` attributes of the atomic part are swapped.

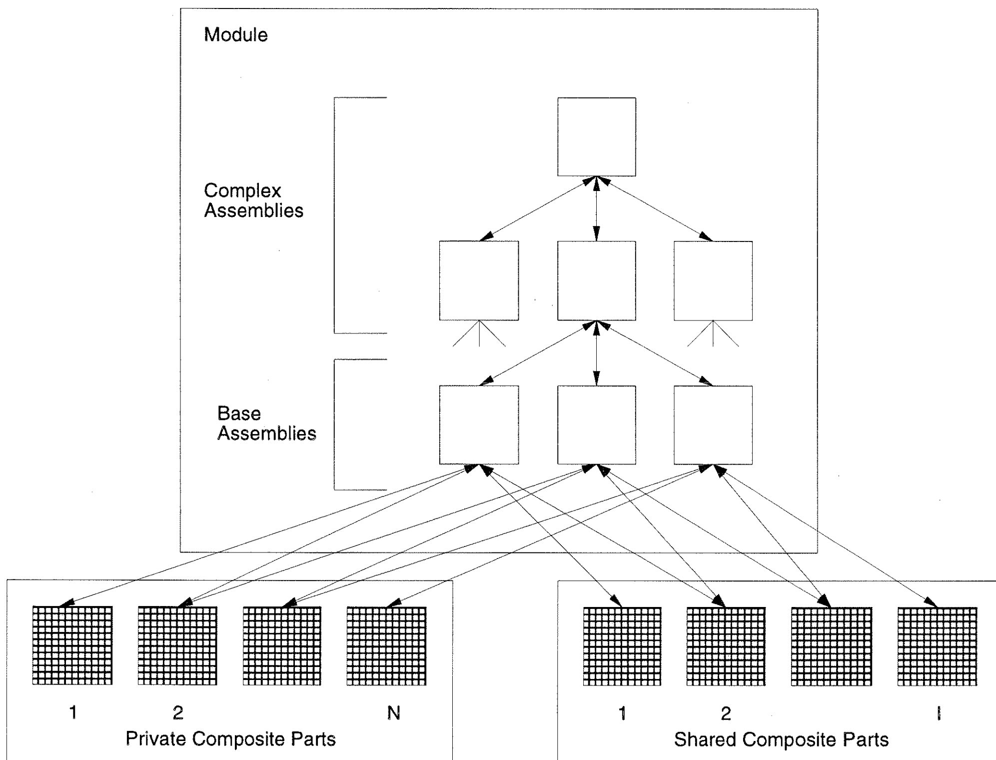


Figure 7.1: Diagram of the Module Object with Private and Shared Composite Parts.

```
beginTransaction;

for RepeatCount do

    if this is a shared transaction
       start at the root of the assembly hierarchy of the shared module;

     else
       start at the root of the assembly hierarchy of the private
               module for the specific client;

    Follow a single random path down the hierarchy to a base assembly;

    From the base assembly, perform some operation on a base assembly;

    Sleep(SleepTime);

end;

endTransaction;
```

Figure 7.2: Generic Multi-user Transaction.

A generic version of the pseudo-code for a traversal transaction for a given client, which was extracted from [9], is shown in Figure 7.2. (This generic version is based on the original OO7 multi-user benchmark specifications.) In Figure 7.2, **RepeatCount** indicates how many composite parts will be visited during a given transaction. It can also be thought of as the number of basic operations per transaction. The **Sleep Time** is used to control the level of activity during the transaction. By assigning a value greater than zero to the **Sleep Time**, one can model transactions that reflect the behavior of an interactive session. If its value is zero, the transaction does not consist of any idle time.

To generate a workload for a client, input parameters are supplied to the workload generator. The four input parameters are a vector of percentages representing each basic operation, a RepeatCount, a Sleep Time parameter, and a parameter to represent the number of operations per transaction.

Operations of a workload are determined based on the input percentages supplied for a given client. Each basic operation is represented by this vector,(rp,rs,wp,ws) where `rp` represents the percentage of private reads, `rs` represents the percentage of shared reads, `wp` represents the percentage of private writes, and `ws` represents the percentage of shared writes. For example, suppose that the vector is (50,0,50,0), the workload generated would consists of transactions that contain 50% read-only operations and 50% update operations applied to the private module of a given client.

These percentages can be thought of as the probability that an operation is generated. A random number is generated modulo 100 and the resulting value along with the input percentages determines which of the basic operations is executed.

7.2 Modifications to OO7 Multi-user Specification

The multi-user workload does not include an operation for reorganizing the persistent store. In our study, we have added the two reorganization functions, Reorg1 and Reorg2, that were added by Yong et al. [46]. Thus, there are six operation percentages, two additional ones to generate transactions that contain one of the reorganization functions. The new operation vector is as follows: (rp,rs,wp,ws,r1,r2) where `r1` represents Reorg1() and `r2` represents Reorg2(). The vector of percentages must add up to 100%.

The original reorganization functions affected all of the composite parts of the OO7 persistent store. For example, the function Reorg1() iterates through all of the composite parts, deleting/inserting half of the atomic parts of the subgraph and counting up the number of atomic parts processed in this manner. All modules are processed during the execution of this function. Thus, one invocation of the original function affects the generation of all clients' workloads regardless of which client originated the reorganization operation.

To eliminate the dependency among client workloads that would occur by this function, the reorganization functions have been modified so that structural modifications are applied only to the atomic parts subgraph of private composite parts of the module owned by the client that issued the reorganization function. To accomplish the same effect as the original reorganization functions, each client's workload would consist of a reorganization operation.

In our study, the **SleepTime** parameter is not used to generate the client workload. To model idle time within transactions, a SleepTime parameter can be included on a per client basis to the trace merger. The SleepTime parameter is in terms of events. It cannot be greater than the number of events used as the value for a time slice. The Swapper subtracts the value of the SleepTime parameter from the value of a time slice and then determines how many events of the trace to process when a sleep time is associated with a Client.

7.3 Experiments

In this section, we describe two experiments conducted using workloads generated from an instrumented version of the OO7 multi-user application. The first experiment is designed to examine the level of contention of three clients that are accessing the shared region of the persistent store. The second experiment is used to compare the events generated by the original OO7 Reorg1() function and the events generated by the reorganization function that was implemented to meet the needs of our infrastructure.

7.3.1 Experiment 1: Three Client Producer/Consumer Multi-user Workload

In this experiment, a very small persistent store was generated using the input parameters shown in Table 7.1. The persistent store was created as a separate transaction and its contents are shown in Table 7.2. Table 7.3 provides a summary of the events generated during the creation of this persistent store.

In this experiment, the clients interact with one another in a producer/consumer relationship. The workload of Client 1 consists of 10% read-only operations to the shared composite parts of the base assemblies and 90% update operations to the shared composite parts of the base assemblies. The workload of Client 2 contains transactions whose operations consist of 90% read-only operations to the shared region of the persistent store and 10% update operations to the shared region. The workload of Client 3 contains transactions whose operations consist of 40%

Table 7.1: Parameters for a Very Small 007 Persistent Store with 3 Modules.

Parameter	Value
NumAtomicPerComp	10
NumConnPerAtomic	3
DocumentSize (bytes)	2000
ManualSize (bytes)	100000
NumCompPerModule	15
NumAssmPerAssm	3
NumAssmLevels	3
NumCompPerAssm	5
NumModules	3

Table 7.2: Total Number of Objects by Type Contained in the Persistent Store.

Type of Object	Total Number
Module	3
Complex Assembly	12
Base Assembly	27
Private Composite Part	45
Shared Composite Part	33
Atomic Part	780

Table 7.3: Number of Events Captured in Workload to Create Persistent Store.

Event Types	Client 0
co	5196
ew	34385
er	12337
dw	27926
dr	39001
sr	1
grt	0
ts	1
te	1

Table 7.4: Parameters to Generate Client Workloads.

Workload Parameters	Client 1	Client 2	Client 3
Private Read Percentage	0	0	0
Private Write Percentage	0	0	0
Shared Read Percentage	10	90	40
Shared Write Percentage	90	10	60
Reorg1 Percentage	0	0	0
Reorg2 Percentage	0	0	0
Number of Transactions	5	5	5
Number of Ops per Transaction	2	2	2

Table 7.5: Operations Executed to Generate Each Client's Workload.

Client	Transaction Number	Private Read	Shared Read	Private Write	Shared Write	Reorg1	Reorg2
Client 1	1	0	0	0	2	0	0
	2	0	0	0	2	0	0
	3	0	0	0	2	0	0
	4	0	0	0	2	0	0
	5	0	0	0	2	0	0
Client 2	1	0	2	0	0	0	0
	2	0	1	0	1	0	0
	3	0	1	0	1	0	0
	4	0	2	0	0	0	0
	5	0	2	0	0	0	0
Client 3	1	0	0	0	2	0	0
	2	0	1	0	1	0	0
	3	0	0	0	2	0	0
	4	0	2	0	0	0	0
	5	0	2	0	0	0	0

read-only operations to the shared region and 60% write operations to the shared region. The input parameters to generate the workloads of the three clients are shown in Table 7.4. The first six input parameters form the operation vector for each of the three clients. The repeat count parameter is represented by the number of operations per transaction. These input parameters produce the combination of operations shown in Table 7.5. The basic operations of Table 7.5 were then used to generate a workload for each of the three clients. These workloads were captured in traces, that are characterized by the number of events by type in Table 7.6.

These three traces were then used as input to the trace merger. The time interval input had a value of 1000 events. The SleepTime parameter for each of these clients had a value of zero. Client 1 had a tick rate of one. Client 2 has a tick rate of 2. Client 3 had a tick rate of 1. The trace merger used this input to generate an interleaved trace.

In this experiment, the trace merger was run several times. Some of the runs varied the number of clients participating in the generation of the multi-user workload, (e.g., two clients or three clients). The time interval input was also changed with the runs. The average number of lock requests per workload was 2957. When three clients participated, the total number of lock requests was 8870 when there was no contention and 8871 when there was one wait. In this experiment,

Table 7.6: Number of Events in the 3 Client Producer/Consumer Workload Traces.

Event Types	Client 1	Client 2	Client 3
co	0	0	0
ew	0	0	0
er	2513	1581	1588
dw	320	40	100
dr	2081	1221	1258
sr	0	0	0
grt	5	5	5
ts	5	5	5
te	5	5	5

Table 7.7: Parameters for a Small 007 Persistent Store with 2 Modules.

Parameter	Value
NumAtomicPerComp	20
NumConnPerAtomic	3
DocumentSize (bytes)	2000
ManualSize (bytes)	100000
NumCompPerModule	150
NumAssmPerAssm	3
NumAssmLevels	6
NumCompPerAssm	3
NumModules	2

there was almost no contention for locks among the concurrent clients.

Further experimentation was then completed to determine why the contention between clients was so little. New traces were generated using a persistent store that was produced with a set of shared composite parts that was half the size of the set of private composite parts. This change had little effect on the the contention. In an experiment in which three clients were used to generate a multi-user workload, two of the clients waited once for a lock.

Thus, we have concluded that the lack of contention is mainly due to the actual OO7 multi-user application. When traversing the hierarchy for each operation, the complex assemblies, the base assemblies, and the chosen composite part of the shared subassemblies is randomly chosen. In addition to a random selection of objects to visit, the shared subassemblies of the based assemblies are randomly chosen. The combination of random selections is enough to reduce the amount of contention to a minuscule amount.

7.3.2 Experiment 2: Two Client Multi-user Workload with Reorganizations

The small persistent store was generated using the input parameters shown in Table 7.7. The persistent store was created as a separate transaction. The content of this persistent store is shown in Table 7.8. Table 7.9 provides a summary of the events generated during the creation of this persistent store.

In this experiment, the two clients generate workloads based on the operation vectors and other

Table 7.8: Total Number of Objects by Type Contained in the Persistent Store.

Type of Object	Total Number
Module	2
Complex Assembly	242
Base Assembly	486
Private Composite Part	300
Shared Composite Part	225
Atomic Part	10500

Table 7.9: Number of Events Captured in Workload to Create Persistent Store.

Event Types	Client 0
co	68218
ew	450111
er	179967
dw	36534
dr	546302
sr	1
grt	0
ts	1
te	1

input parameters shown in Table 7.10. Both operation vectors of Table 7.10 indicate that only one type of operation, Reorg1(), is to be performed. The other input parameters indicate that the workload should consist of one transaction containing one operation. The operations that were used to generate the workloads for both clients are shown in Table 7.11.

The combined number of object creations for Client 1 and Client 2 are presented in Table 7.12 along with the number of object creations generated on the same persistent using the original OO7 Reorg1() function. As stated above, the numbers confirm that applying the reorganization function of our study to all of the modules will give you results similar to that of the original Reorg1() function. For example, the combined total of new atomic parts created during the execution of the workloads of Client 1 and Client 2 is 2970 and the number of newly created atomic parts generated through the execution of the original Reorg1() is 3000. For completeness, Table 7.13, a table of all the events generated by the three workloads, is included.

7.4 Observations

As result of studying the OO7 multi-user workload benchmark, the following observations were made:

- **Observation 1:** The infrastructure designed and developed in this dissertation is well suited for the OO7 multi-user benchmark. Using AMPS to instrument the OO7 multi-user application allows us to capture the workload of a client in a trace. Each of the single client traces can then be combined in a variety of ways to generate a trace that reflects a multi-user

Table 7.10: Parameters to Generate the Reorganization Workloads.

Workload Parameters	Client 1	Client 2
Private Read Percentage	0	0
Private Write Percentage	0	0
Shared Read Percentage	0	0
Shared Write Percentage	0	0
Reorg1 Percentage	100	100
Reorg2 Percentage	0	0
Number of Transactions	1	1
Number of Ops per Transaction	1	1

Table 7.11: Operations Executed to Generate Each Client's Workload.

Client	Transaction Number	Private Read	Shared Read	Private Write	Shared Write	Reorg1	Reorg2
Client 1	1	0	0	0	0	1	0
Client 2	1	0	0	0	0	1	0

Table 7.12: Number of Events Per Object Type Created by Reorg1() Functions.

Event Types	Client 1	Client 2	Client 1 + Client2	Original Reorg1()
Atomic Part	1470	1500	2970	3000
Connection	6009	6177	12186	12307
Assoc	2940	3000	5940	6000

Table 7.13: Number of Events Generated by the Reorganization Workloads.

Event Types	Client 1	Client 2	Original Reorg1()
co	10419	10677	21307
ew	69735	71385	142535
er	173687	177459	346164
dw	68478	70235	140100
dr	238422	243916	477927
sr	0	0	0
grt	1	1	1
ts	1	1	1
te	1	1	1

behavior without rerunning the OO7 multi-user application.

- **Observation 2:** Although the time to generate an in-memory version of the persistent store as well as the single-user workloads increases with the size of the persistent store, the flexibility afforded during the processing phase compensates for the length of time taken during the collection phase.

- **Observation 3:** Some researchers have based their experimentation on the assumption that access to persistent object stores is random and thus there is rarely any contention in some persistent object applications [36]. Our limited experimentation with the OO7 multi-user benchmark that was performed in our study supports this assumption.

- **Observation 4:** Prior to experimenting with the multi-user workload, all workloads were generated with the `no garbage collection begin` and `no garbage collection end` events as "weak transactions". With the multi-user workloads, we introduced the flat transaction model that caused us to look at the creation of events more closely. Through the design process, we became aware of the inadequacy of the events that were designed to capture the creation of objects of the persistent store. Since we capture persistent objects at the application level, we start recording the manipulations of objects as soon as an object is created. This is partially due to the fact that the applications used in our study consist of objects that are all persistent. These applications also have no operation to indicate when an object becomes persistent.

  Although PTF does not adequately capture object creation, it does produce semantically correct traces. However, the traces contain manipulations of the object prior to its persistence. Thus, a persistent object system would need to keep track of which events occurred before it made an object persistent and discard the extra events.

In summary, the OO7 multi-user benchmark has provided us with an application to illustrate the feasibility of our instrumentation infrastructure for generating multi-user workloads. Analysis of this application with respect to contention among clients and storage management of persistent object systems can provide some initial insights that must be further tested in real applications.

Chapter 8

Related Work

Trace-driven simulation has been used effectively in many different areas within computer systems, including the evaluation of persistent object systems. We begin this chapter with a short review of this work. Next, we consider how this prior work relates to ours. Specifically, we consider prior work in trace formats and in application benchmarking and modeling.

8.1 Trace-Driven Simulation

In addition to Cook et al. [13, 15, 16], whose work is described in Chapter 2, one other group has used trace-driven simulation for performance studies of persistent object systems. Scheuerl et al. [40] used event traces to analyze the I/O performance of various recovery mechanisms. Their analytical I/O cost model, MaStA, estimates performance for a given configuration, and consists of an application workload, a recovery mechanism, and execution machine architectures. Using their system, accesses are recorded as trace events during the executions of synthetic workloads. The traces are then used in the following ways [39]: to analyze and validate assumptions of the actual MaStA model; to examine real and simulated devices to calibrate the device simulators; and to compare I/O costs of the devices.

8.2 Trace Formats for Performance Evaluation

Trace formats have been developed to capture information about the behavior of applications in various areas of computer systems design and evaluation, such as in studies of memory management [26, 28] and storage management in persistent object systems.

Our PTF design is most closely related to the work of Scheuerl et al., who developed the MaStA I/O trace format [39] to study the cost of various recovery mechanisms with respect to I/O. The trace events of MaStA record information on reading, writing, and synchronization. The format for an event is a fixed size of two 32-bit words. Each event consists of 7 fields. The 7 fields of a trace entry represent the following: an operation, an I/O category, a pattern, a region number, a block number, and a stream number. Integer values are stored in bigendian format. In the trace format, the use of different logical areas of storage are represented by regions. A synchronization operation can occur over more than one of these regions. The stream field of the entry is used to capture concurrency. A stream follows threads of I/O accesses; thus, reads and writes are recorded with respect to a thread. The developers have defined four operations: read, write, sync (the synchronization operation), and block. The trace format is designed to allow up to 250 new events.

Since the developers are categorizing I/O, the second byte for read and write entries is an I/O category. There are four I/O categories: data, recovery, installation, and commit.

As shown in the description of MaStA provided above, the MaStA format captures device-level physical behavior. Our traces capture workloads at a logical level. In designing PTF, we recognize the need for capturing behavior in traces at many different levels. However, we have focused on an implementation-independent representation to provide a trace that can be used in a wider variety of contexts.

8.3 Performance Evaluation Based on Workload Models

In our approach to modeling application workloads, we facilitate modeling by providing a C++ framework for implementing and instrumenting a model of a persistent application, and by providing a TCL interface for rapidly constructing model workloads. Relatively little related work in this area has focused on providing explicit support for workload modeling. Here we mention three efforts of which we are aware.

Missikoff and Toiati [32] have developed a system, MOSAICO, that supports the design, conceptual modeling, and rapid prototyping of an object database application. The system consists of a graphical user interface to model the application. The model is then encoded in the language TQL++. This system also consists of a subsystem that compiles the conceptual model to generate executable code. MOSAICO differs from AMPS in the level of support it provides. While we support users with a C++ library and scripting interface, MOSAICO provides much higher-level tools, such as a programming language and visual interface. Our approach, while more modest, still has advantages. For example, we feel that AMPS is more likely to be adopted by users having pre-existing applications that they would like to model.

Another approach to modeling an object application has been developed by Schreiber [41] in the development of the JUSTITIA benchmark. Schreiber models an object application by categorizing the objects of the applications into one of three types: static objects, simple dynamic objects, and complex dynamic objects. Using these three types, Schreiber models the database of the application as a tree structure in which the number of leaves of the tree can vary at different levels. The major disadvantage of this approach is that there are no clearly defined methods for transforming a persistent store structure into a corresponding tree-like structure. AMPS differs from JUSTITIA in that it allows application data to be modeled as an arbitrary graph structure.

The last approach that we discuss was developed by Darmont et al. [17]. Darmont et al. developed a generic benchmark, OCB (the Object Clustering Benchmark), to evaluate the performance of clustering policies. The persistent store of the benchmark is modeled using several input parameters: `NC`, `MAXNREF`, `BASESIZE`, `NREFT`, and `NO`. The parameter `NC` indicates the number of classes that are to be contained in a persistent store. The parameter `MAXNREF` represents the maximum number of references contained in an instance of a class, and the parameter `BASESIZE` represents the base size of an instance. The parameter `NREFT` represents the number of different types of references. By allowing different references to point to the same class, several kinds of relationships can exist between the objects of the persistent store, zero-to-many, one-to-many, and many-to-many. The types of references can be chosen randomly or they can be set by the developer. The parameter `NO` represents the number of objects of the persistent store. The generation of the persistent store begins with defining the various class structures up to `NC`. Once the classes are defined, a consistency check is performed to eliminate cycles and discrepancies in the inheritance graph. Using a random distribution to choose a class structure, objects are created up to `NO`. As with our modeling, the persistent store is structured as a graph. In our model, we capture the specific information about

data members of a class; however Darmont et al. are only concerned with the pointers of a class and the size of the object. One advantage of their approach to modeling is that it allows flexibility in the preciseness of the model of classes of a given schema and the relationships between these classes.

Chapter 9

Conclusion

The work of this dissertation addresses the problem of a lack of tools to perform research in the area of performance evaluation of persistent object systems. This dissertation also provides a framework in which data can be shared among researchers in the form of traces that capture the behavior of the generated workloads.

Specifically, we described an infrastructure for generating and sharing experimental workloads for the purpose of evaluating persistent object systems. The infrastructure is used to generate both single-user workloads and multi-user workloads. Our approach consists of three components: PTF, a common trace format, AMPS, a toolkit to aid in the modeling and instrumentation of persistent object applications, and a processing component to merge single-user workloads to form a multi-user workload. The development of an instrumentation infrastructure as described in this dissertation provides the following benefits: the process of building new experiments for analysis is made easier; experiments to evaluate the performance of implementations can be conducted and reproduced with less effort; and pertinent information can be gathered in a cost-effective manner.

PTF captures the structure of a persistent application's data and the time-varying behavior of an application. PTF is novel in that it captures that behavior at the application level. In particular, the events capture information about the manipulation of objects at a logical level independent of the physical level. Because we used this approach, a single PTF trace can be used to evaluate any number of different persistent object system implementations.

AMPS is designed to aid in the creation and instrumentation of a model of a persistent object application. Through its C++ libraries, the effort required to implement a model of an application is reduced in three ways. First, the process of instrumentation is not as error prone as it is with hand instrumentation because instrumentation is localized to methods that access fields and to classes provided by the toolkit. Second, modeling and instrumentation of traversals of an application are reduced using the generic traversals provided by the toolkit. Finally, through the TCL interpreter environment, once the schema and behaviors of an application have been modeled, workloads can be easily scripted.

A trace merger is used to simulate a multi-user environment in order to create a multi-user workload under a particular concurrency control policy. These workloads allow some sharing of the data of the persistent store. The infrastructure for generating multi-user workloads is novel in that traces of single-user workloads are processed by the trace merger to produce a trace that captures the multi-user behavior of the combined single users. This infrastructure enriches the semantics of multi-user workloads, thus allowing for a broader range of experimentation.

Both PTF and AMPS represent only the beginning of a more complete system. There are still some issues that need to be addressed in developing a general, yet effective, trace format. In the

remainder of this chapter, we pose some questions that reflect open issues and future work.

9.1 Future Work

One of the major issues we have carefully considered is how to separate the logical and physical workloads. Nevertheless, an important question that remains to be answered is at what application level the workload should be captured. For example, are operations on collections, such as "add an element", represented with a single high-level trace event or with a sequence of low-level events reflecting a particular implementation of the collection? In our current format, we take the latter approach, which means that the implementation of collections is implicit in our trace. As a result, our format is not appropriate for studying different collection implementations directly.

Similarly, for indexed collections, the data structure representing the index and the manipulations involved in doing a lookup are not captured in our format. Thus another issue that might be addressed is the support in both PTF and AMPS to capture the structure and manipulations of indexes. A basic question that remains is whether the behavior of indexes can be captured at the application level. The answer to this question depends on the level of transparency provided by the persistent object system. For instance, in the O2 system [34], the implementation of the index is transparent at the application level. However, with the Exodus system indexes for collections of an application are implemented at the application level. In systems that require indexes to be created at the application level, it is possible to model the class specification of the index object and then produce trace events. The question then becomes at what level of abstraction should the manipulations to the index be captured within trace events.

Other, even more difficult issues arise when one attempts to capture the behavior of multi-user workloads. One of the simplifying assumptions that was made in order to generate a multi-user workload by merging single-user workloads was that program control was not dependent on data values. An open issue that remains is whether this assumption can be relaxed and does it affect the type of applications that can be modeled by the infrastructure. Futhermore, we have proposed a model of sharing of data of the persistent store that restricts restructuring updates. Although we allow some form of sharing that allows us to perform experimentation with interfering clients, we do not know if our sharing model is sufficient for capturing multi-user workloads generated from non-CAD-like applications.

In addition to enriching the sharing model, other transaction models and concurrency control policies can be explored as part of future work. The optimistic concurrency control policy is used by some researchers in analysis of distributed persistent stores and is a good candidate for an alternate concurrency control policy for the trace merger.

Lastly, a major issue is the question of how to get other analysts to adopt PTF and AMPS. The issue of path-to-adoption is very important in many designs (e.g., consider how Java went from a design to a widely used language), but is often not considered at all. The two main ways to get users to adopt a new technology are to make it so valuable that they are willing to take the time to learn the technology, at some cost, or to make the new technology easier to use than what they are currently using (thus giving a benefit at no cost). In the context of PTF, the first approach would involve convincing analysts to modify their current performance evaluation frameworks to generate PTF traces. In our research, we have chosen the second path, which is to make PTF very easy to generate. We do this by providing the AMPS toolkit to simplify the creation of workloads and to make the generation of PTF traces almost automatic.

Bibliography

[1] L. Amsaleg, M. Franklin, P. Ferreira, and M. Shapiro. Evaluating Garbage Collectors for Large Persistent Stores. In *OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance*, Austin, TX, October 1995.

[2] L. Amsaleg, M. Franklin, and O. Gruber. Efficient Incremental Garbage Collection for Client-Server Database Systems. Technical Report CS-TR-3370, Department of Computer Science, University of Maryland, College Park, MD, October 1994.

[3] M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott. PS-Algol: An Algol with a Persistent Heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1982.

[4] M.P. Atkinson, L. Daynes, M.J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *ACM SIGMOD Record*, December 1996.

[5] S. Banerjee and C. Gardner. Towards An Improved Evaluation Metric For Object Database Management Systems. In *OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance*, Austin, TX, October 1995.

[6] A.J. Berre, T.L. Anderson, and M. Mallison. The HyperModel Benchmark. Technical Report CS/E 88-031, Oregon Graduate Center, Beaverton, Oregon, 1988.

[7] M. Butler. Storage Reclamation in Object-oriented Database Systems. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 410–423, San Fancisco, CA, 1987.

[8] M.J. Carey, D.J. Dewitt, M.J. Franklin, N.E. Hall, M.L. McAuliffe, J.F. Naughton, D.T. Schuh, M.H. Solomon, C.K. Tan, O.G. Tsatalos, S.J. White, and M.J. Zwilling. Shoring Up Persistent Applications. In *Proceedings of the ACM SIGMOD International Confernece on the Management of Data*, pages 383–394, Minneapolis, MN, May 1994.

[9] M.J. Carey, D.J. Dewitt, C. Kant, and J.F. Naughton. A Status Report on the OO7 OODBMS Benchmarking Effort. *ACM SIGPLAN Notices*, 29(10):414–426, October 1994.

[10] M.J. Carey, D.J. DeWitt, and J.F. Naughton. The OO7 Benchmark. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 12–21, Washington, DC, June 1993.

[11] R.G.G. Cattell and D.K. Barry. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1997.

[12] W.P. Cockshot, M.P Atkinson, and K.J Chisholm. Persistent Object Management System. *Software-Practice and Experience*, pages 49–71, 1984.

[13] J.E. Cook, A.W. Klauser, A.L. Wolf, and B.G. Zorn. Semi-automatic, Self-adaptive Control of Garbage Collection Rates in Object Databases. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 377–388. SIGMOD, June 1996.

[14] J.E. Cook, A.L. Wolf, and B.G. Zorn. The Design of a Simulation System for Persistent Object Storage Management. Technical Report CU-CS-647-93, CUCS, Boulder, CO, March 1993.

[15] J.E. Cook, A.L. Wolf, and B.G. Zorn. Partition Selection Policies in Object Database Garbage Collection. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 371–382, Minneapolis, MN, May 1994.

[16] J.E. Cook, A.L. Wolf, and B.G. Zorn. A Highly Effective Partition Selection Policy for Object Database Garbage Collection. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):153–172, January 1998.

[17] J. Darmont, B. Petit, and M. Schneider. OCB: A Generic Benchmark to Evaluate the Performances of Object-Oriented Database Systems. In *Proceedings of the Sixth International Conference on Extending Database Technology (EDBT'98)*, pages 326–340, Valencia, Spain, March 1998. LNCS Vol. 1377 (Springer).

[18] J. Darmont and M. Schneider. Model to Evaluate the Performance of OODBs. In *Proceedings of the 25th Very Large Data Base Conference*, pages 254–265, Edinburgh, Scotland, 1999.

[19] C.J. Date. *An Introduction to Database Systems*, chapter 4, pages 391–414. Addison-Wesley System Programming Series, 1995.

[20] L. Daynes. Extensible Transaction Management in PJava. In *The First International Workshop on Persistence and Java (PJWI)*, Drymen, Scotland, September 1996.

[21] Exodus Project Document, Computer Sciences Department, University of Wisconsin, Madison, WI. *An Introduction to GNU E*, 1993.

[22] Exodus Project Document, Computer Sciences Department, University of Wisconsin, Madison, WI. *Using the EXODUS Storage Manager V3.1*, 1993.

[23] J. Gray. *The Benchmark Handbook*, chapter 2, pages 21–127. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[24] J. Gray. *The Benchmark Handbook*, chapter 7, pages 397–432. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[25] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, chapter 8, pages 449–488. Morgan Kaufmann Publishers, San Franciso, CA, 1993.

[26] M.D. Hill. *Man Pages for Dinero.* Computer Science Department, University of Wisconsin, Madison, WI.

[27] M. Holliday. Techniques for Cache and Memory Simulation Using Address Reference Traces. *International Journal of Computer Simulation*, 1:129–151, 1991.

[28] E.E. Johnson and J. Ha. PDATS Lossless Address Trace Compression for Reducing File Size and Access Time. In *Proceedings of IEEE International Conference on Computers and Communications*, pages 213–219, Phoenix, AZ, May 1994.

[29] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

[30] U. Maheshwari and B. Liskov. Partitioned Garbage Collection of a Large Object Store. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 313–323, Minneapolis, MN, May 1994.

[31] W.J. McIver and R. King. Self-Adaptive, On-Line Reclustering of Complex Object Data. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 407–418, Minneapolis, MN, May 1994.

[32] M. Missikoff and M. Toiati. MOSAICO—A System for Conceptual Modeling and Rapid Prototyping of Object-Oriented Database Application. *SIGMOD Record*, 23(2):508, June 1994.

[33] J.E.B. Moss. Design of the Mneme Persistent Object Store. *ACM Transactions on Information Systems*, 8(2):103–139, April 1990.

[34] O2 Technology, Inc, Palo Alto, CA. *O2 System Administration Guide*, 1995.

[35] A. Ralston. *Encyclopedia of Computer Science and Engineering*, page 167. Van Nostrand Reinhold, New York, NY, 1983.

[36] P. Ranganathan, K. Charachorloo, S.V. Adve, and L.A. Barroso. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *Proceedings of the 8th International Conference on Architectural Support of Programming Languages and Operating Systems*, pages 307–318, October 1998.

[37] K. Rotzell and M.E.S. Loomis. Benchmarking an ODBMS. *Journal of Object-Oriented Programming*, 1(4):66–72, April 1991.

[38] D. Samples. Mache: No-loss Trace Compaction. In *Proceedings of the 1989 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 89–97, May 1989.

[39] S.J.G. Scheuerl, R.C.H. Connor, R. Morrison, J.E.B. Moss, and D.S. Munro. The MaStA I/O Trace Format. Technical Report CS/95/4, School of Mathematical and Computational Sciences, University of St. Andrews, North Haugh, St Andrews, Fife, Scotland, 1995.

[40] S.J.G. Scheuerl, R.C.H. Connor, R. Morrison, and D.S. Munro. The MaStA I/O Cost Model and its Validation Strategy. In *Proceedings of the Second International Workshop Advances in Databases and Information Systems (ADBIS'95)*, pages 165–175, Moscow, June 1995.

[41] H. Schreiber. Evaluating Garbage Collectors for Large Persistent Stores. In *OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance*, Austin, TX, October 1995.

[42] V. Singhal, S.V. Kakkad, and P.R. Wilson. Texas: An Efficient, Portable Persistent Store. In *5th International Workshop on Persistent Object Systems*, 1992.

[43] A. Tiwary, V.R. Narasayya, and H.M. Levy. Evaluation of OO7 as a System and an Application Benchmark. In *OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance*, Austin, TX, October 1995.

[44] R.A. Uhlig and T.N. Mudge. Trace-Driven Memory Simulation: A Survey. *ACM Computing Surveys*, 29(2):128–170, June 1997.

[45] P.R. Wilson, M.S. Lam, and T.G. Moher. Caching Considerations for Generation Garbage Collection. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 32–42, San Franciso, CA, June 1992.

[46] V.-F. Yong, J. Naughton, and J.-B. Yu. Storage Reclamation and Reorganization in Client-Server Persistent Object Stores. In *Proc. of the 10th International Conference on Data Engineering*, pages 120–131, February 1994.

[47] B.G. Zorn. The Effect of Garbage Collection on Cache Performance. Technical Report CU-CS-528-91, Department of Computer Science, University of Colorado, Boulder, CO, 1991.

Appendix A

Parameters of PTF Trace events

In this appendix, the input parameters for each trace event of PTF is provided. These input parameters are shown in the context of the C++ definition for each of the trace events of the file `Trace.h`.

A.1 Trace.h File

The file `Trace.h` contains the definitions for the trace events of PTF. It was originally implemented by Artur Klauser as part of the Trace Converter package. Below is the modified version of `Trace.h` which includes the revisions that were made to the trace specification as a result of the work of this dissertation.

```
//*************************************************************
//
//  Trace Converter
//  converts trace files between Ascii and binary representation
//
//
//*************************************************************

#ifndef TraceConverterH
#define TraceConverterH

//*************************************************************
// definition of trace events
// new events *must* be added at the end of the appropriate
// section to keep compatible with old traces
//*************************************************************
enum EventTypes {
  //---high level events ---
  EventTypeMake = 0,
  EventTypeRemove,
  EventTypeVisit,
  EventTypeRandom,
  EventTypeCollect,
```

```
  EventTypeCheckpoint,
  EventTypeRestart,
  EventTypePrintConnectivity,
  EventTypeSimulationDone,
  //--- low level events (from DB traces) ---
  EventTypeTraceBegin = 64,
  EventTypeTraceEnd,
  // only following events can have comments
  EventTypeCreateObject,
  EventTypeDeleteObject,
  EventTypeEdgeWrite,
  EventTypeEdgeRead,
  EventTypeDataWrite,
  EventTypeDataRead,
  EventTypeSetRoot,
  EventTypeGetRoot,
  EventTypeNoGCStart,
  EventTypeNoGCEnd,
  EventTypeTransactionStart,
  EventTypeTransactionEnd,
  EventTypeTransactionAbort,
  EventTypeTransactionCheckPt,
  EventTypeCreateArrayObject,
  EventTypeArrayDataWrite,
  EventTypeArrayDataRead,
  EventTypeFormatObject,
  EventType_FinalSentinel  // must be last entry in this enum
};

//*****************************************************************
// declaration of binary trace file structs and lengths;
// length are not defined by sizeof() due to terminal struct
// padding
//*****************************************************************

typedef unsigned char   EventTypeDef;
typedef unsigned int    int32;
typedef unsigned short  int16;
typedef unsigned char   int8;

struct EventDefCreateObject {
  int32 TransId;
  int32 ClientId;
  int32 FormatId;
  int32 OId;
};
const int EventLenCreateObject = 16;
```

```
struct EventDefDeleteObject {
  int32 TransId;
  int32 ClientId;
  int32 FormatId;
  int32 OId;
};
const int EventLenDeleteObject = 16;

struct EventDefEdgeWrite {
  int32 TransId;
  int32 ClientId;
  int32 FormatId;
  int32 FromOId;
  int32 ToOId;
  int8  Edge;
};
const int EventLenEdgeWrite = 21;

struct EventDefEdgeRead {
  int32 TransId;
  int32 ClientId;
  int32 FormatId;
  int32 FromOId;
  int8  Edge;
};
const int EventLenEdgeRead = 17;

struct EventDefDataWrite {
  int32 TransId;
  int32 ClientId;
  int32 FormatId;
  int32 OId;
  int32 Offset;
};
const int EventLenDataWrite = 20;

struct EventDefDataRead {
  int32 TransId;
  int32 ClientId;
  int32 FormatId;
  int32 OId;
  int32 Offset;
};
const int EventLenDataRead = 20;

struct EventDefCreateArrayObject {
  int32 TransId;
```

```
  int32 ClientId;
  int32 FormatId;
  int32 OId;
  int32 ContainerOId;
  int32 NoOfElements;
};
const int EventLenCreateArrayObject = 24;

struct EventDefArrayDataRead {
  int32 TransId;
  int32 ClientId;
  int32 FormatId;
  int32 OId;
  int32 Offset;
  int32 Index;
  int32 Length;
};
const int EventLenArrayDataRead = 28;

struct EventDefArrayDataWrite {
  int32 TransId;
  int32 ClientId;
  int32 FormatId;
  int32 OId;
  int32 Offset;
  int32 Index;
  int32 Length;
};
const int EventLenArrayDataWrite = 28;

struct EventDefSetRoot {
  int32 TransId;
  int32 ClientId;
  int32 FormatId;
  int32 OId;
};
const int EventLenSetRoot = 16;

struct EventDefFormatObject {
  int32 FormatId;
  int32 SuperFormatId;
  int32 NumberOfPointers;
  int32 NumberOfDataMembers;
  int32 NumberOfArrayMembers;
  int32 LengthOfClassName;
};
const int EventLenFormatObject = 24;
```

```
struct EventDefGetRoot {
  int32 TransId;
  int32 ClientId;

};
const int EventLenGetRoot = 8;

struct EventDefNoGCStart {
  int32 Trid;
  int32 ClientId;
};
const int EventLenNoGCStart = 8;

struct EventDefNoGCEnd {
  int32 Trid;
  int32 ClientId;
};
const int EventLenNoGCEnd = 8;

struct EventDefTransactionStart {
  int32 TransId;
  int32 ClientId;
};
const int EventLenTransactionStart = 8;

struct EventDefTransactionEnd {
  int32 TransId;
  int32 ClientId;
};
const int EventLenTransactionEnd = 8;

struct EventDefTransactionAbort {
  int32 TransId;
  int32 ClientId;
};
const int EventLenTransactionAbort = 8;

struct EventDefTransactionCheckPt {
  int32 TransId;
  int32 ClientId;
};
const int EventLenTransactionCheckPt = 8;


/*****************************************************************
/ Note: The actual length of the Format Object event is equal to
```

```
/       EventLenFormatObject + 4 * NumberOfDatatMembers +
/       8 * NumberOfArrayMembers + LengthOfClassName
/*******************************************************************

union EventDef {
  EventDefCreateObject          CreateObject;
  EventDefDeleteObject          DeleteObject;
  EventDefEdgeWrite             EdgeWrite;
  EventDefEdgeRead              EdgeRead;
  EventDefDataWrite             DataWrite;
  EventDefDataRead              DataRead;
  EventDefSetRoot               SetRoot;
  EventDefCreateArrayObject     CreateArrayObject;
  EventDefArrayDataRead         ArrayDataRead;
  EventDefArrayDataWrite        ArrayDataWrite;
  EventDefFormatObject          FormatObject;
  EventDefGetRoot               GetRoot;
  EventDefNoGCStart             NoGCStart;
  EventDefNoGCEnd               NoGCEnd;
  EventDefTransactionStart      TransactionStart;
  EventDefTransactionEnd        TransactionEnd;
  EventDefTransactionAbort      TransactionAbort;
  EventDefTransactionCheckPt    TransactionCheckPt;

};

#endif TraceConverterH
```

Appendix B

Interface Specifications for Components of the Example Trace Merger

This appendix contains the interface specification for components of the example trace merger that was implemented as part of this dissertation. The trace merger takes traces generated for each individual client and forms a trace that captures the combined behavior of the participating clients.

B.1 Functions of the Trace Merger

In this section, the interface specification for the three functions of the trace merger are provided. The functions are as follows: `Main`, `Scheduler`, and `Swapper`.

B.1.1 Main Function

Signature

```
void main(int argc,
          char **argv)
```

Functionality

Takes as input the maximum number of events to process during one time interval, a file `Workload.asc` that contains the number of clients, the names of each of the trace files, the names of each client, the tick rate, and the sleep time, and a debug flag, to indicate whether debugging information should be printed while it is executing. `Main` creates a ClientCoordinator for each of the clients provided in `Workload.asc`. It allocates a LockManager. It opens a file to hold the interleaved trace events and then invokes the Scheduler.

B.1.2 Scheduler Function

Signature

```
void Scheduler(int numberofClients,
               ClientCoordinator **clientList,
               int eventNumber,
               LockManager *lockMN,
               int traceFileFd)
```

Functionality

Keeps track of the number of active simulated client sessions. It also implements a round-robin algorithm when all of the ClientCoordinators are at normal priority. If a ClientCoordinator's priority has been changed to high, then it will be allowed to run before a ClientCoordinator with normal priority. It invokes the Swapper providing it with a pointer to a ClientCoordinator and the number of events to process during the current time interval for the client represented by the ClientCoordinator.

B.1.3 Swapper Function

Signature

```
Bool Swapper(ClientCoordinator *client,
             LockManager *lockMN,
             int numberOfEvents,
             int traceFileFd)
```

Functionality

The Swapper implements the processing of flat transactions. It makes request to the lock manager based on the type of event that it is processing. It also manipulates the events of a transaction based on the whether it ends with a `transaction end` event or a `transaction abort` event. When a `transaction end` event is encountered, the Swapper writes the update events to the interleaved trace file. It also makes a request to the lock manager to release all the locks of the transaction. The Swapper returns `true` when it has completely processed the workload file associated with the ClientCoordinator; otherwise it returns `false`.

Return Values

1	:	completed processing trace file
0	:	trace file not completely processed

B.2 The C++ Classes of the Trace Merger

The interface specifications for the C++ Classes of the trace merger are presented in this section.

B.2.1 The LockManager Class

The LockManager class manages all of the locks requested by the different clients. It determines if the lock request is compatible with the current lock mode of a requested lock. If there is no conflict, the requester of the resource is added to the set of lock requests and the lock is granted. If there is a conflict and deadlock will not occur as a result of the requester waiting, the requester is allowed to wait.

The methods of the LockManager class are as follows:

```
LockManager::LockManager(int noOfLocks);

             creates a lock table containing noOfLocks and
             initializes the compatibility matrix.


LockStatus LockManager::requestLock(ClientCoordinator *clientCoor,
                                   LockMode mode, int resourceId);

                        determines whether clientCoor is granted the
                        lock request, allowed to wait, or must abort
                        its current transaction because the lock
                        request is denied.  Upgrades the lock mode
                        when appropriate.  Creates a new lock when
                        the lock request is for a resource which does
                        not have a lock associated with it.


LockStatus LockManager::requestUnLock(ClientCoordinator *clientCoor,
                                      Lock *heldLock);

                        removes the lock request corresponding to
                        clientCoor from  heldLock and the pointer
                        to the lock tht is stored in the Client-
                        Coordinator's lock set.  If the request is
                        the only request for the lock, the lock is
                        removed from the lock table.


void LockManager::requestUnLockAll(ClientCoordinator *clientCoor);

                  removes all the lock requests for locks associated
                  with the current transaction of clientCoor.
```

```
bool LockManager::deadlockDetected(Lock *lock,
                                   ClientCoordinator *requestingClient);
```

    determines whether requestingClient will cause a
    deadlock if allowed to wait on lock. If not,
    return false. If a deadlock will occur, determine
    which one of the ClientCoordinator's must abort. If
    requestingClient is chosen to abort, true is
    returned; otherwise false is returned.

```
LockManager::~LockManager();
```

    deallocates the lock table.

B.2.2 ClientCoordinator Class

The `ClientCoordinator` class represents a pseudo-process. It keeps trace of the status of the current transaction, the reading point of the working buffer, and the management of any temporary files and the trace file representing the workload of a given client. It also keeps track of the locks acquired during the current transaction and state information associated with the current transaction.

The methods of the `ClientCoordinator` class is as follows:

```
ClientCoordinator::ClientCoordinator(int clientNumber,
                                     int processingRate,
                                     int sleepTime,
                                     char * traceFileName);
```

    creates a ClientCoordinator to represent the client
    indicated by the argument clientNumber and
    initializes its data members.

```
void ClientCoordinator::setTransStatus(TransStatus status);
```

    sets the transaction status for the current
    transaction. The status of a transaction is as
    follows: NonActive, Active, Prepared, Aborting,
    Committing, Aborted, Committed.

```
TransStatus ClientCoordinator::getTransStatus();
```

    returns the transaction status.

```
char* ClientCoordinator::createTempBuf(int tempBufDataLen);
```

```
                         creates a working buffer of length
                         tempBufDataLen for processing
                         trace events of an aborted transaction.

char* ClientCoordinator::getTempBuf();

                         returns the pointer to the working buffer.


void ClientCoordinator::deleteTempBuf();

                        deletes the working buffer.

char* ClientCoordinator::getProcessingBuffer();

                         returns pointer to processing buffer for
                         trace events.

int ClientCoordinator::getClientTraceFile();

                       returns the file pointer of the workload
                       trace file.

FILE* ClientCoordinator::getTempFile();

                         creates a temporary file for processing
                         trace events and returns a pointer to
                         the file.


ProcessingLocation ClientCoordinator::getTransLocation();

                                      returns the location of the
                                      transaction which is either
                                      a buffer or a temporary file.


void ClientCoordinator::setTransLocation(ProcessingLocation transloc);

                        sets the data member transactionLocation to
                        indicate that the transaction is either in
                        the processing buffer or a temporary file.


int ClientCoordinator::getTickRate();
```

returns the tick rate associated with the client.

```
int ClientCoordinator::getSleepTime();
```

returns the number of events to use as SleepTime.

```
bool ClientCoordinator::getRefreshBufFlag();
```

returns a flag to indicate whether the processing buffer needs to be refreshed (e.g, fill the buffer with additional trace events from the workload trace file).

```
void ClientCoordinator::setRefreshBufFlag(bool flagValue);
```

sets the value of the RefreshBuf flag.

```
int ClientCoordinator::getCurrentPosInBuf();
```

returns the offset within the processing buffer to start processing new trace events.

```
void ClientCoordinator::setCurrentPosInBuf(int position);
```

sets the new starting position within the processing buffer.

```
int ClientCoordinator::getBufDataLen();
```

returns the length of the processing buffer.

```
void ClientCoordinator::setBufDataLen(int dLength);
```

sets the length of the processing buffer.

```
int ClientCoordinator::getTransReplayCount();
```

returns the number of times that the current transaction has re-started.

```
void ClientCoordinator::setTransReplayCount(int count);
```

updates the number of times that the current transaction has re-started.

```
void ClientCoordinator::setTransIdentifier(int trid);
```

sets the value of the current transaction identifier.

```
int ClientCoordinator::getTransIdentifier();
```

returns the identifier of the current transaction.

```
void ClientCoordinator::addTransLock(Lock *transHeldLock);
```

adds a lock pointer to the transaction lock set.

```
void ClientCoordinator::removeTransLockSet();
```

removes all the lock pointers of the transaction lock set.

```
void ClientCoordinator::removeTransLock(Lock *transLock);
```

removes the lock pointer transLock from the transaction lock set.

```
int ClientCoordinator::getTransLocksNumber();
```

returns the number of locks in the transaction lock set.

```
void ClientCoordinator::getTransLockSet(int numberOfLocks,
                                        Lock **transLockSet);
```

returns a list of the pointers of locks
contained in the transaction lock set
through the input argument transLockSet.

```
void ClientCoordinator::setStartTransPosition(int startPosition);
```

sets the data member transBeginPosition
to the offset within the processing
buffer that represents the starting
position of the current transaction.

```
int ClientCoordinator::getStartTransPosition();
```

returns the offset in the processing buffer
that represents the starting position of
the current transaction.

```
int ClientCoordinator::getTempBufDataLen();
```

returns the length of the temporary buffer.

```
char* ClientCoordinator::getUpdateBuffer();
```

returns a pointer to the buffer that
contains the update events of the
transaction.

```
char* ClientCoordinator::resizeUpdateBuffer();
```

creates a larger buffer to hold the
update events of a transaction and
returns a pointer to the new buffer.

```
int ClientCoordinator::getPositionInUpdateBuffer();
```

returns the offset to start writing new
update events.

```
void ClientCoordinator::setPositionInUpdateBuffer(int position);
```

sets the data member updateCursorPos

```
                        to the offset within the update buffer
                        to start writing new update events.

int ClientCoordinator::getUpdateBufferSize()

                       returns the size of the update buffer.

Lock* ClientCoordinator::getWaitLock();

                          returns a pointer to the lock that
                          the current transaction is waiting for;
                          otherwise returns NULL.

void ClientCoordinator::setWaitLock(Lock *lockValue);

                        sets the data member waitLock to the pointer
                        of the lock that the current transaction
                        is waiting to gain access.

bool ClientCoordinator::getReadHeaderFlag();

                        returns the value of the flag that indicates
                        whether to read the header of the trace
                        binary.

void ClientCoordinator::setReadHearderFlag(bool flagValue);

                        sets flag that indicates whether to read the
                        header of the binary trace file.

bool ClientCoordinator::getAbortingFlag();

                        returns the value of the flag that indicates
                        that the transaction has aborted and now
                        in the process of retrying the transaction.

void ClientCoordinator::setAbortingFlag(bool flagValue);

                        sets the flag that indicates that the current
                        transaction has just been aborted and that
                        the current transaction is to be retried.
```

```
bool ClientCoordinator::getLMAbortFlag();
```

returns the value of the flag that indicates whether the lock manager requested an abort of the current transaction.

```
void ClientCoordinator::setLMAbortFlag(bool flagValue);
```

sets the value of the flag that indicates whether the lock manager requested an abort of the current transaction.

```
int ClientCoordinator::getClientNumber();
```

returns the number of the client represented by the ClientCoordinator.

```
ProcessState ClientCoordinator::getProcessingState();
```

returns the processing state.

```
void ClientCoordinator::setProcessingState(ProcessState pvalue);
```

sets the data member processingState to either Waiting, Ready, Running, or Stopped.

```
Priority ClientCoordinator::getProcessingPriority();
```

returns the priority at which the ClientCoordinator is to be run.

```
void ClientCoordinator::setProcessingPriority(Priority pvalue);
```

sets the data member processingPriority to either Normal or High.

```
ClientCoordinator::~ClientCoordinator()
```

destructor for the ClientCoordinator.

B.2.3 Lock Class

The Lock class represents the lock that is used to protect an object during a transaction. The Lock class consists of a mode, a resource identifier that is the object identifier of the object protected by the lock, a pointer to the queue of request for the lock, a pointer to the queue of wait request, and a count of the number of owners of the resource. The lock modes are Free, Read, and Write.

The methods the Lock class as follows:

```
Lock::Lock(int resource, LockMode mode);
```

creates a lock and initializes the data members.

```
void Lock::setMode(LockMode mode);
```

set the mode of the lock.

```
LockMode Lock::getMode();
```

returns the mode of the lock.

```
LRQueue* Lock::getLockRequestQueue();
```

returns a pointer to the lock request queue.

```
LRQueue* Lock::getLockRequestWaitQueue();
```

returns a pointer to the lock wait request queue.

```
Lock::~Lock();
```

deallocates the request and wait request queues.

B.2.4 LRQueue Class

The LRQueue class implements a queue containing lock request objects. The methods of the LRQueue are as follows:

```
LRQueue::LRQueue();
```

creates a lock request queue.

```
LockRequest* LRQueue::popLockRequest()

                        removes the first lock request from the queue.


void LRQueue::pushLockRequest(LockRequest *requestPtr)

                adds a lock request to the queue.


LockRequest* LRQueue::front()

                        returns a pointer to the lock request at the
                        beginning of the queue.


LockRequest* LRQueue::removeLockRequest(ClientCoordinator *client)

                        removes the lock request owned by client
                        and returns a pointer to the lock request.


LockRequest* LRQueue::lookupLockRequest(ClientCoordinator *client)

                         looks for a lock request that is owned by
                         client.  If a lock request exists, it
                         returns a pointer to the lock request;
                         otherwise it returns NULL.
```

B.2.5 LRQueueItr Class

The LRQueueItr class is an iterator for the LRQueue class. The methods of this class are as follows:

```
LRQueueItr::LRQueueItr(LRQueue *lockReqOueue);

            constructor for the iterator.


LockRequest* LRQueueItr::next();

                           returns the next lock request in queue.


LRQueueItr::~LRQueueItr();
```

destructor for the iterator.

B.2.6 LockRequest Class

The LockRequest class represents a lock request. The data members of the LockRequest class are a lock status, a lock mode, and a pointer to the ClientCoordinator requesting the lock.

The methods of the LockRequest class are as follows:

```
LockRequest::LockRequest(LockStatus lstatus, LockMode lmode,
                         ClientCoordinator *clientRef)
```

creates a lock request and initializes its data members.

```
LockStatus LockRequest::getStatus();
```

returns the status of the lock request.

```
LockMode LockRequest::getMode();
```

returns the mode of the lock request.

```
void LockRequest::setConvertMode(LockMode lmode);
```

sets the new mode of the lock request. For example, a lock request with a mode Read may be upgraded to a Write mode.

```
LockMode LockRequest::getConvertMode();
```

returns the upgraded mode if it was upgraded.

```
ClientCoordinator* LockRequest::getClient();
```

returns a pointer to the ClientCoordinator that owns the request.