An Infrastructure for Generating and Sharing Experimental Workloads for Persistent Object Systems

Thorna Humphries, Artur Klauser,
Alexander L. Wolf, and Benjamin G. Zorn

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

University of Colorado
Department of Computer Science
Technical Report CU-CS-883-99 April 1999

A version of this report to appear in
Software—Practice and Experience

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. In this paper, we describe an infrastructure for generating and sharing experimental workloads to be used in evaluating the performance of persistent object system implementations. The infrastructure consists of a toolkit that aids the analyst in modeling and instrumenting experimental workloads, and a trace format that allows the analyst to easily reuse and share the workloads. Our infrastructure 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. We describe the two major components of this infrastructure, the trace format and the toolkit. We also describe our experiences using these components to model, instrument, and experiment with the OO7 benchmark.

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

1 Introduction

Benchmarking is an important technique for assessing the performance of persistent object systems, whether existing or proposed. Conceptually, a benchmark consists of two elements: the structure of the persistent data, and 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.

The benchmark data and behavior can be derived either from a synthetic application or from a real application. Underlying both approaches is a clear need to elicit and represent application data and behavior independently of any particular persistent object system that might undergo the assessment. In our work, for example, we are interested in evaluating the storage management features of persistent object systems, concentrating most recently on garbage collection algorithms [9, 10, 11]. To evaluate such algorithms, we require information about the time-varying nature of data connectivity, which means we need information about the occurrence of application operations that result in the creation of new objects and the modification of inter-object references. The sequence of such events is independent of the persistent object system upon which the application might have been or will be implemented. How well a persistent object system responds to the sequence, of course, is the measure of its performance.

The sequence of application events is called a *trace*, and is at the heart of an assessment technique called *trace-driven simulation* [23]. A trace-driven simulation consists of three phases: collection, reduction, and processing [12]. Because we are concerned with dynamic behavior, 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.

Two questions 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?

The work described in this paper is aimed at addressing these two questions.

First, we have developed a general-purpose trace format, called PTF (POSSE Trace Format), that is the specification of a set of events characterizing application operations on persistent object stores. PTF traces can be used for a variety of purposes, including simulation studies, application visualizations, debugging, and statistical summaries of application behavior.

Second, we have developed a library and associated tool, called AMPS (Application Modeling for Persistent Systems), 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 in terms of persistent store behaviors such as generation, traversals, and updates.

Two approaches to collecting application traces are illustrated in Figure 1. The approach on the left side of the figure involves the hand instrumentation of an application, followed by the execution of the instrumented application using an actual persistent object system. The execution results in a trace file that can then be fed into an analyzer. The right side of the figure shows the approach based on AMPS. Using AMPS, 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 then runs without the need for an actual persistent object system. As in the first approach, a trace file is produced that serves as input to an analyzer.

We have gained substantial experience with both approaches. In fact, we have gone through three generations of work with instrumentation-based experimentation. In the first generation, we built a synthetic application that made direct procedure calls to a simulation system for persistent object storage management. PTF arose from a desire to separate the application from the simulator so that we could have better control over the running (and rerunning) of experiments, as well as to provide a way to make our experimental input (not just output) available to other researchers. In the second generation, we performed a hand instrumentation of an implementation of an application to produce PTF traces. Our experience with hand instrumentation led in the third generation to the development of AMPS, which allowed us to create an instrumented application more easily and flexibly.

There is, of course, a third approach to collecting traces. That approach involves modifying the underlying persistent object system engine to generate and collect trace events. While it is possible to collect trace data in this way, we do not consider it to be a viable alternative for the following two reasons. First, the instrumentation would be performed at a very low level of abstraction, so recovering the higher-level application semantics could be difficult. Second, the lower the level of instrumentation, the more likely that platform dependencies are introduced into the trace, which is something we want to avoid.

While PTF can be used independently of AMPS, using them together gives substantial leverage to an analyst 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.

In the current versions of PTF and AMPS, we make some simplifying restrictions. First, PTF only captures single-user workloads and AMPS only models single-user workloads. Second, there is no support in PTF and AMPS for capturing or modeling the behavior of an application with respect to concurrency, transaction processing, and manipulation of indices. Our current work is aimed at relaxing these restrictions. Nevertheless, PTF and AMPS have proven their utility in supporting meaningful experimentation despite their current limitations.

This paper describes PTF and AMPS. We begin by introducing PTF and AMPS using a simple application as an example. Following that, we review our experience in using PTF and AMPS to capture both the structure of the data and the application behavior specified by the OO7 benchmark. We then review related work in the areas of trace formats, trace-driven simulation, and performance evaluation. We conclude with a summary and discussion of future work.

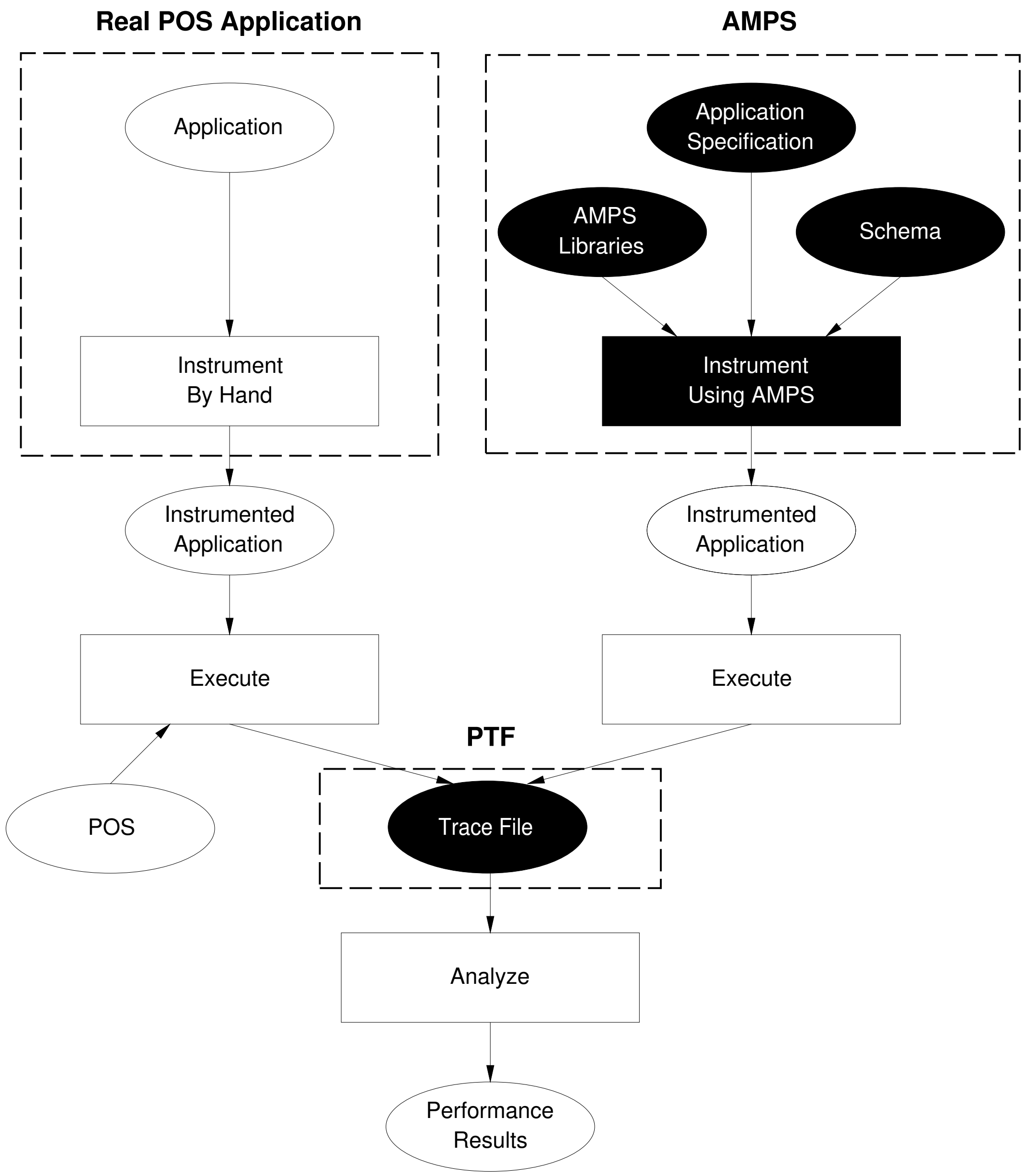

Figure 1: Two Approaches to Application Instrumentation.

2 POSSE Trace Format

PTF is used to capture events that occur during the execution of an application manipulating a persistent object store. By *application* we mean any number of threads of execution that access and manipulate the object store over a period of time. By *behavior* we mean a thread of execution operating on the object store with a specific high-level purpose (such as populating it, reorganizing it, traversing it, etc.) We call the combination of the behaviors of the application together with the structure of the store the *workload* captured by the trace. For simplicity, we always assume that the object store is initially empty, and that the application begins by populating the store.

Our main purpose for designing PTF was to develop a representation of a workload that was independent of any particular persistent object system implementation. With such a representation, 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. We refer to information about the application that is not specific to a particular persistent object system implementation as the *logical workload*, and the same information augmented with details of a particular persistent object system implementation as the *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. As much as we can, our intent is to capture the logical workload in PTF events. This goal is not unlike the design of the Java Virtual Machine [14], which also abstracts away physical information in its representation.

2.1 PTF Design

Here we provide a conceptual overview of the contents of a PTF trace, leaving full details of the format to be described elsewhere [13]. For example, details related to dynamically resizing objects or annotating trace events with arbitrary information are provided in the full description.

PTF uses a logical object identifier (OID) 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.

PTF contains events that reflect operations to create, delete, access, and modify persistent objects. Table 1 outlines the events in PTF, placing them into four 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` and `data write`, where each event indicates that a single value 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 type of the object created, specifically the OID of a "type" object. The type object describes the fields of an object in terms of their types and their relative positions in the representation of the object. Traces that delete an object and then later read from or write to that object are considered erroneous. Our persistence model uses the mechanism of persistence by reachability [3]. The event `set root` indicates a root of the reachability analysis. Any number of objects can be designated as roots.

Category	Event Name	Abbreviation	Arguments
Object	`create object`	`Co`	class ID, OID
	`delete object`	`Do`	class ID, OID
	`set root`	`Sr`	class ID, OID
Atomic Data	`data read`	`Dr`	class ID, OID, offset
	`data write`	`Dw`	class ID, OID, offset
Connections	`edge read`	`Er`	class ID, OID, offset
	`edge write`	`Ew`	class ID, from OID, offset, to OID
Directives	`begin no collection`	`Ts`	
	`end no collection`	`Te`	

Table 1: PTF Trace Events.

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 new object is linked into a persistent structure (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 transaction. The current definition of PTF does not support transaction events in their most general form, but we are in the process of defining appropriate support in the next version of PTF.

2.2 PTF Example

To illustrate the use of PTF in capturing a workload, we present a simple example. Figure 2 depicts a state of a simple persistent store organized as a binary tree, in which each node contains a data value and pointers to left (offset 0) and right (offset 1) subtrees. Also shown are the logical OIDs used in the PTF trace to identify each object. Figure 3 contains the PTF trace for a simple two-behavior application that first builds the binary tree of Figure 2 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 `Ts` and `Te`, 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.

Every trace event that manipulates an object contains the OID of the type of the object. In

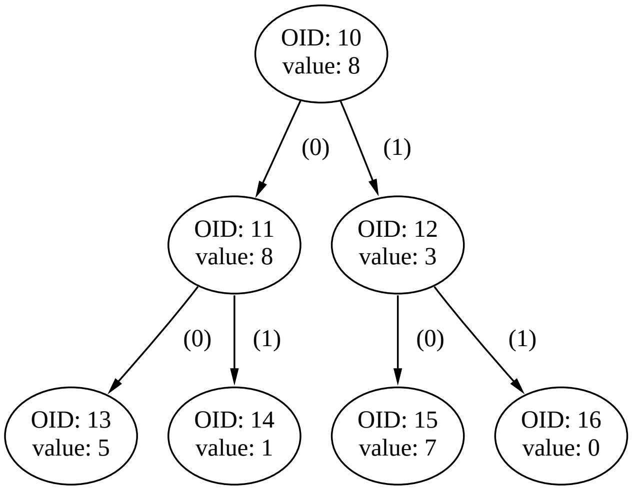

Figure 2: A Simple Persistent Store Organized as a Binary Tree.

the binary tree example, the objects are all of the same type, and the OID of this type object is 20 in Figure 3. 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 type information. The advantage of including the type information with each event is that tools processing the trace do not have to always look up the type of each object, thus increasing the speed of trace processing. We feel that this is a reasonable trade off between space and time.

3 AMPS

In this section, 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 section.

3.1 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 [6], the Hypermodel benchmark [4], the OO1 benchmark suite [7], and the Trouble Ticket Benchmark [16].

As an example of this situation, consider that there currently 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. [25] used a subset of the OO7 benchmark suite in their study of storage reclamation and reorganization of persistent object stores. Maheshwari and Liskov [15] used a homogeneous collection of 30-byte objects in their study of partitioned garbage collection of a large object store.

```
Trace begin
Ts                Disallow garbage collection until after Te event
Co 20 10          Create object with OID 10 whose class OID is 20
Dw 20 10 1        Write data value to position 1 in object 10 of class 20
Sr 20 10          Set object 10 of class 20 to be a persistent root
Co 20 11          Create object with OID 11 whose class OID is 20
Dw 20 11 1        Write data value to position 1 in object 11 of class 20
Ew 20 10 0 11     Write edge 0 from object 10 of class 20 to object 11
Co 20 12          Create object with OID 12 whose class OID is 20
Dw 20 12 1        Write data value to position 1 in object 12 of class 20
Ew 20 10 1 12     Write edge 1 from object 10 of class 20 to object 12
Co 20 13          Create object with OID 13 whose class OID is 20
Dw 20 13 1        Write data value to position 1 in object 13 of class 20
Ew 20 11 0 13     Write edge 0 from object 11 of class 20 to object 13
Co 20 14          Create object with OID 14 whose class OID is 20
Dw 20 14 1        Write data value to position 1 in object 14 of class 20
Ew 20 11 1 14     Write edge 1 from object 11 of class 20 to object 14
Co 20 15          Create object with OID 15 whose class OID is 20
Dw 20 15 1        Write data value to position 1 in object 15 of class 20
Ew 20 12 0 15     Write edge 0 from object 12 of class 20 to object 15
Co 20 16          Create object with OID 16 whose class OID is 20
Dw 20 16 1        Write data value to position 1 in object 16 of class 20
Ew 20 12 1 16     Write edge 1 from object 12 of class 20 to object 16
Te                Allow garbage collection to occur
Dr 20 10 1        Read data value from position 1 in object 10 of class 20
Er 20 10 0        Read value of edge 0 from object 10 of class 20
Er 20 10 1        Read value of edge 1 from object 10 of class 20
Dr 20 11 1        Read data value from position 1 in object 11 of class 20
Er 20 11 0        Read value of edge 0 from object 11 of class 20
Er 20 11 1        Read value of edge 1 from object 11 of class 20
Dr 20 12 1        Read data value from position 1 in object 12 of class 20
Er 20 12 0        Read value of edge 0 from object 12 of class 20
Er 20 12 1        Read value of edge 1 from object 12 of class 20
Dr 20 13 1        Read data value from position 1 in object 13 of class 20
Dr 20 14 1        Read data value from position 1 in object 14 of class 20
Dr 20 15 1        Read data value from position 1 in object 15 of class 20
Dr 20 16 1        Read data value from position 1 in object 16 of class 20
Trace end
```

Figure 3: Annotated PTF Trace Generated from a Binary Tree Application.

In our own work, we have used a forest of augmented binary trees of objects containing some number of non-tree edges [10].

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 richer shared infrastructure for developers of persistent object systems to evaluate their designs.

3.2 Architecture

Figure 4 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 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, 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. 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.

[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.

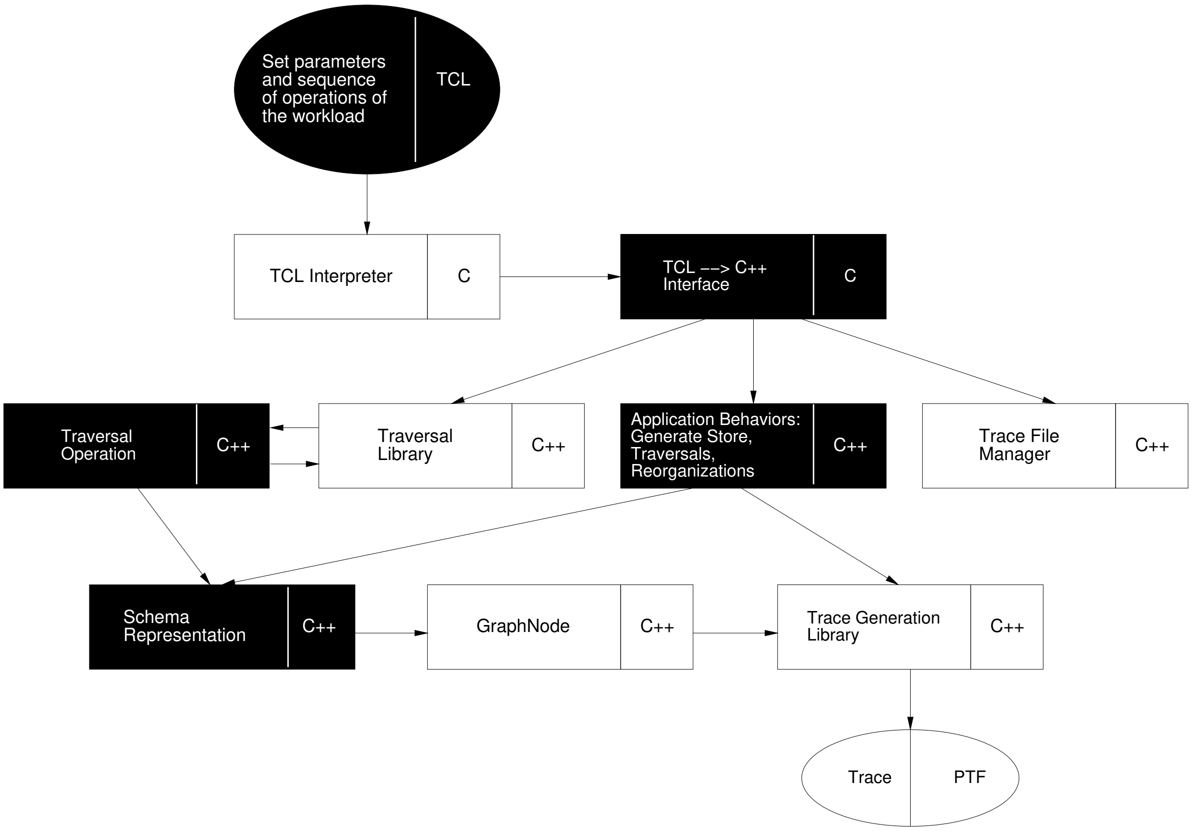

Figure 4: AMPS Architecture.

3.3 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 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 2.

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.

3.3.1 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 5. 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 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 OID 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 6. In this figure, the class

```
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);
     int resizeEdge(ResizeType tvalue, int noOfEdges, char *noteBuf = NULL);
     void readData(int position, char *noteBuf = NULL);
     void writeData(int position, char *noteBuf = NULL);
     int getNumberOfEdges();
     int getNodeIdentifier();
     ~GraphNode();

     ...
};
```

Figure 5: C++ Class Specification of `GraphNode`.

`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 6 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`.

3.3.2 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

```
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 6: C++ Class Specification of `BinTreeNode`.

persistent store of Figure 2. 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 7 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 7 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 7. 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 7. 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 8. `SumNodes` consists of a constructor that initializes the field `totalSize`, which is the result of applying this operation to each of the nodes of 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.

3.3.3 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.

```
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 7: C++ 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 currentNode) {
        totalSize = totalSize + ((BinTreeNode *)(currentNode))->RefValue();
    }

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

  private:
    int totalSize;
};
```

Figure 8: C++ Class Specification of `SumNodes`.

```
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 9: Implementation of the TCL command `TG_DBbuild`.

In order 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 9, the C implementation of the TCL command `TG_DBbuild` is shown. All inputs to the TCL commands are ASCII 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 9. 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 10 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`,

```
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 10: Scripting a Workload Using AMPS and TCL.

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.

3.4 Non-Application Persistent Objects

Some objects and data structures are part of the POS implementation (e.g., indices, extents) and not the application itself. These objects need to be represented in the PTF trace, but are not conceptually separate from the POS implementation away from which the trace is attempting to abstract. Conceptually, POS objects are separate from and should be modeled separately from the application objects. AMPS currently does not allow POS objects to be distinguished from application objects and as a result, someone using AMPS must implement the POS objects necessary to accurately model the situation. To be more concrete, in the next section we discuss the fact that extents were required in our modeling of the OO7 benchmark. We are currently considering possible approaches to handling POS objects in a more appropriate way. For example, we are currently exploring the possibility of introducing new POS-specific high-level trace events (such as `create_index` and `update_index`) that can be converted to PTF events via a POS-specific preprocessor.

4 Experience Modeling OO7

In our efforts to evaluate new policies for persistent object storage management, we selected the OO7 benchmark [6] because of its availability and its use in other similar evaluation studies. 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 [22].

The developers of the OO7 benchmark distribute several implementations, including an E version and a C++ version. In one of our studies of garbage collection policies, we hand instrumented a subset of the E version, which is based on the Exodus storage manager, and used it to generate the PTF traces that drove our analyses [8]. After developing the AMPS toolkit, we created

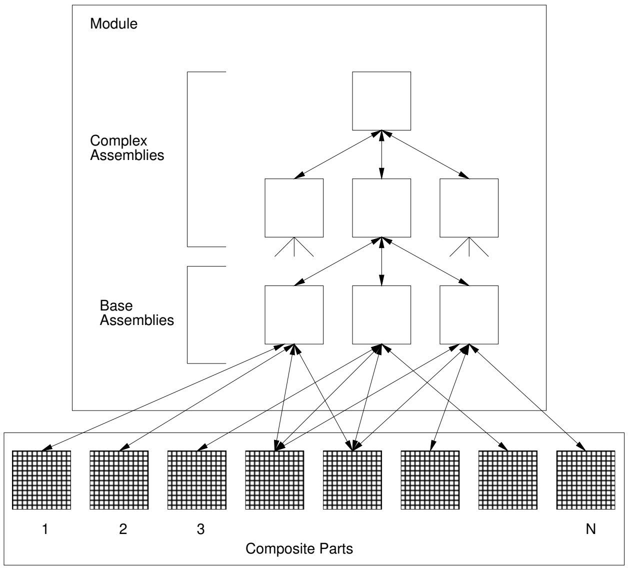

Figure 11: Diagram of the Module Object of the OO7 Benchmark.

a second, 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 PTF traces from each implementation. This section describes our experience building the two implementations and the results of the assessment.

4.1 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 11.

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 11, 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 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. [25] 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 section, when we refer to the OO7 benchmark, we are referring to this enhanced version.

4.2 Early Experience with OO7

As mentioned above, our first experience with the OO7 benchmark involved the hand instrumentation of the E implementation to gather PTF trace information. To perform this instrumentation correctly, we had to completely understand the source code of the implementation to identify the appropriate places to insert code. During this process, if a single event was overlooked, the state recorded in the trace would not reflect the state of the persistent store created by the implementation. In order to assess correctness, we created very small stores by altering the parameters of the benchmark. The traces were then submitted to our simulator 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. The process required a significant amount of manual labor and the hand instrumentation was error prone.

Beyond the effort in producing the instrumented version of the OO7 implementation, we found that hand instrumenting a particular implementation gave us no additional leverage when we wanted to instrument a new application, since each application's source code had to be reinstrumented. Furthermore, the OO7 implementation that we were using was developed on a MIPS-based DECStation, and in subsequent years our local MIPS machines were replaced by DEC Alphas, rendering the earlier instrumentation unusable because at that time there was no version of Exodus available on the DEC Alphas.

Although hand instrumentation proved to be problematic, the generation of PTF traces proved invaluable. We were easily able to organize and document the experiments, rerun the same experiments multiple times (e.g., in the face of resource limitations or simulator errors), and gather statistical information about the various workloads.

4.3 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 our hand instrumentation of the E version of the benchmark. 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 our two implementations of the benchmark.

4.3.1 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 12 shows the C++ specification of class `CompositePart` before our modifications for use with AMPS. Figure 13 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 5). A depiction of an instance of `CompositePart` is shown in Figure 14. 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.

4.3.2 Modeling Extents

As mentioned in Section 3.4, 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 15, while the version implemented using AMPS is shown in Figure 16. 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 12: Original C++ Class Specification of `CompositePart`.

4.3.3 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 17(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 13: C++ Class Specification of AMPS Versions of `DesignObject` and `CompositePart`.

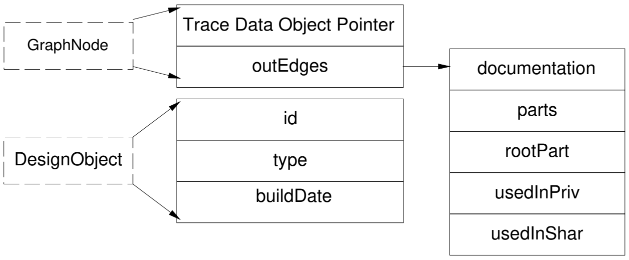

Figure 14: Graphical Representation of an Instance of `CompositePart`.

```
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 15: Original C++ Class Specification of `Assoc`.

```
class Assoc {
  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 16: 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 17: AMPS-Native (a) versus Existing-Code (b) Invocations of the Traversal T1 Apply Method.

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

Table 2: Parameters for a Small 007 Persistent Store.

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 17(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 17(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.

4.3.4 Comparison of PTF Traces

Let us now 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. Using the OO7 benchmark parameters listed in Table 2, 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 3, 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.

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

Table 3: Comparison of AMPS-Instrumented and Hand-Instrumented Traces.

5 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 section 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.

5.1 Trace-Driven Simulation

For years, trace-driven simulation has been a popular approach to 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 [23] 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 [26] and Wilson [24] 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 us to apply it to the related study of performance of storage management in persistent object system implementations.

Specifically, in prior work we used trace-driven simulation to investigate the performance of storage management algorithms in persistent object systems [8, 10, 11]. We developed a simulator, ODBsim, that uses traces as input [9]. Through trace-driven simulations, we 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. Scheuerl et al. [20] 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: 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 [19].

5.2 Trace Formats for Performance Evaluation

Many different 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 [18]) 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 the PTF has been to ensure that it supplies all the information necessary for our storage management performance studies.

- *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 future work with multi-user traces requires that we make approximations that reduce the trace accuracy.

Our PTF design is most closely related to the work of Scheuerl et al., who developed the MaStA I/O trace format [19] to study the cost of various recovery mechanisms with respect to I/O. While our traces capture workloads at the logical level, the MaStA format captures device-level physical behavior. We recognize the need for capturing behavior in traces at many different levels, but we have focused on an implementation-independent representation to provide a trace that can be used in a wider variety of contexts.

5.3 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 two efforts of which we are aware.

Missikoff and Toiati [17] 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 object applications has been developed by Schreiber [21] 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.

6 Summary

In this paper, we have described an infrastructure for generating and sharing experimental workloads for the purpose of evaluating persistent object systems. Our approach consists of two components: PTF, a common trace format, and AMPS, a toolkit to aid in the modeling and instrumentation of persistent object applications. We believe that research in the area of performance evaluation of persistent object systems is hindered by a lack of tools. Specifically, there is little published information about the structure and time-varying behavior of persistent object applications. The development of an instrumentation infrastructure as described in this paper 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.

We have described PTF and AMPS and provided a high-level understanding of our approach using a simple example. We also described our experience using PTF and AMPS to model parts of the OO7 benchmark. While a number of issues remain, our experience with these tools gives us confidence that, at least for the purpose of evaluating the performance of storage management implementations, our tools are effective.

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.

We have only started investigating all the valuable additional tools made possible by a common format such as PTF. One such tool, which we have considered in some detail, would allow users to visualize the time-varying changes made to the persistent objects by a particular application [5].

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 two 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.

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

Other, even more difficult issues arise when one attempts to capture traces of multi-user workloads. Traces of such workloads cannot separate the physical and logical workload completely without making simplifying assumptions; this is an area we are actively investigating.

Another 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.

Our immediate future work has two major focuses. One focus will be to enhance the AMPS toolkit and the PTF trace format to support the generation of trace events that capture the concurrency within a multi-user workload. The other focus is to extend our instrumentation infrastructure so that traces can be used to analyze the impact of indices on the performance of persistent object systems.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. This work was supported, in part, by the National Science Foundation under Grant IRI-95-21046.

References

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

[5] H. Bryant. A Visualization Design for Linked Data Structures Comprising an Object-Oriented Database System. Master's thesis, University of Colorado, Boulder, CO, May 1998.

[6] 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.

[7] R.G.G. Cattell. *The Benchmark Handbook for Database and Transaction Processing Systems*, chapter 7, pages 397–432. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] T.O. Humphries, A.W. Klauser, A.L. Wolf, and B.G. Zorn. POSSE Trace Format. Technical report, Department of Computer Science, University of Colorado, Boulder, CO, 1999. Version 1.0.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.