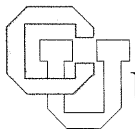
# POSSE Trace Format
# Version 1.0

Thorna O. Humphries
Artur W. Klauser
Alexander L. Wolf
Benjamin G. Zorn

CU-CS-897-00



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

# POSSE Trace Format
# Version 1.0

Thorna O. Humphries, Artur W. 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-897-00 January 2000



## ABSTRACT

*The POSSE Trace Format (PTF) is our attempt to develop a standard trace format for use in the study of persistent object systems (POS) through trace-driven simulations. A major component of this approach involves analyzing traces that capture the structure of a persistent store and the time-varying behavior of persistent object applications. In this report, we present an overview of the PTF design. A detailed syntactic layout of PTF is then described, together with an explanation of the semantics of each trace event type defined. PTF is our effort to develop a standard trace format that captures a wide range of information about a persistent object application. It also provides a means to share information among the community of POS researchers.*

# 1 Introduction

Trace-driven simulation has been a popular and cost-effective approach to evaluating the performance of proposed cache and paging designs. It has been used for a variety of purposes, including the evaluation of dynamic storage management implementations. For example, in the early 90s, Zorn [11] and Wilson [10] used trace-driven simulations to study the performance impact of garbage collection on caches. As a result of the effectiveness of this technique in the domain of primary memory, it was applied to a related domain, that of storage management in persistent object system implementations. Specifically, in prior work, trace-driven simulation was used to investigate the performance of storage management algorithms in persistent object systems, concentrating on garbage collection algorithms [3, 4, 5]. In the initial study [4], a synthetic application that made direct procedure calls to a simulation system for persistent object storage management was used to generate trace events. From this study, two important observations lead to the development of the POSSE Trace Format (PTF). The first observation was that the application needed to be separated from the simulator to gain better control over experiments. The second observation was that there needed to be a mechanism to generate experimental input that could be made available to other researchers. Thus the initial goals of PTF are as follows:

1. 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.

2. To develop a system-independent representation of a workload.

3. To promote the creation of a collection of representative application behaviors in a common trace format.

Although there are several ways in which to collect information from an application, including modifying the underlying persistent object system engine, the approach we have adopted involves instrumenting the persistent object system application and then executing the instrumented application as shown in Figure 1. By instrumenting the application, we can capture the higher-level application semantics independently of a specific platform. Within this approach, the instrumented application can be executed under an actual persistent object system to produce a PTF trace file that can then be fed into an analyzer as shown in Figure 1. This approach also lead us to the development of a library and associated tool, AMPS (Application Modeling for Persistent Systems), that consists of a set of C++ classes and a TCL interface to ease the creation of instrumented applications. Using AMPS, an application is modeled using a combination of a schema specification and the AMPS library. The application model then runs without an actual persistent object system to generate a PTF trace file. A detailed discussion of the AMPS tool appears in a separate publication [6].

The POSSE Trace Format (PTF) is our 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. 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.

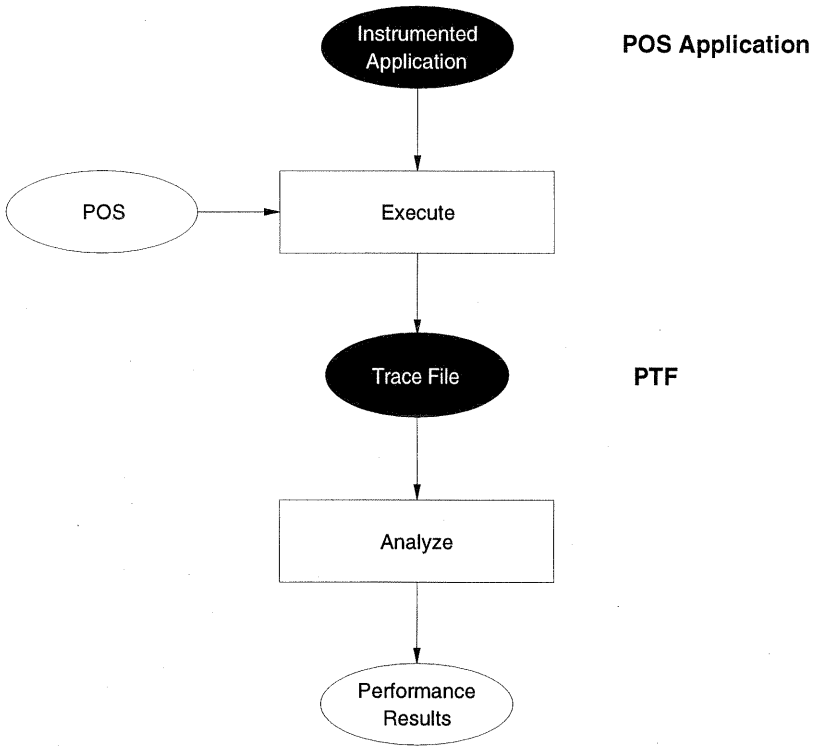



Figure 1: The Role of PTF in POS Trace Generation and Analysis.

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

- Transient objects.
- Objects that are transient because they are newly created and 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. 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 format 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 similar in spirit to the design of the Java Virtual Machine [7], 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 stucture of the persistent store as described in the schema specification of the OO7 benchmark [2] as well as the behavior of the tasks that make up single-user workloads of the OO7 benchmark. Although we have only used the trace files in simulation studies of storage management in persistent object systems, the PTF traces can be used for a variety of purposes, such as application visualizations, debugging, and statistical summaries of application behavior.

This report describes PTF. We begin with an overview of the PTF design. The remainder of the report describes the syntax and semantics of each trace event. We then describe the binary and ASCII representations of the trace format. We conclude with a summary and a brief discussion of future work.

## 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 2 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.

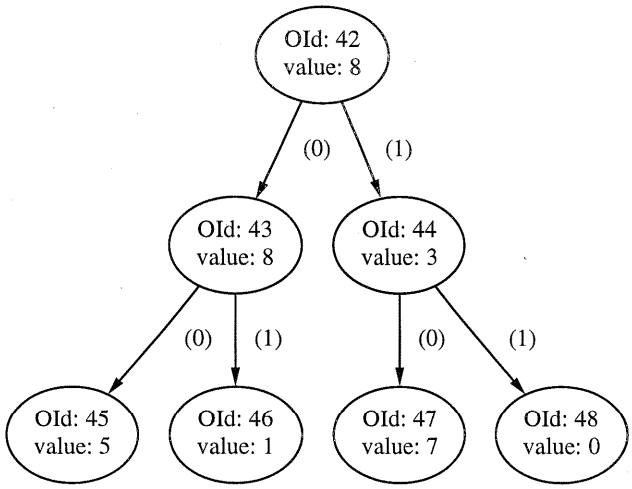



Figure 2: A Simple Persistent Store Organized as a Binary Tree.

PTF contains events that reflect operations to create, delete, access, and modify persistent objects. Table 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 types 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 [1]. The event `set root` indicates that the object contains the root set to be used in the reachability analysis. This object will be referred to as the super root and is maintained by the POS. At the application level, any number of objects can be designated as roots. 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

| Category | Event Name | Abbreviation | Arguments |
|---|---|---|---|
| Format Object | format object | fo | super class format OId, number of pointers, number of data attributes, number of array data attributes, length of object name, 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 | ts | |
| | end no collection | te | |

Table 1: PTF Trace Events.

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 transactions.

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 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.

# 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 4.

## 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>
```

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>
```

```
Trace begin
fo 41 0 2 1 0 11 11 BinTreeNode Format object for the BinTreeNode object
ts                              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 the super 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 42 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
te                              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: Annotated PTF Trace Generated from a Binary Tree Application.

```
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.

## 3.2 Events that Manipulate Objects of the Persistent Store

### 3.2.1 Format Object

Event: `<format_object>`

Parameters:

| | | |
|---|---|---|
| (int) FormatId | format object identifier |
| (int) SuperFormatId | format identifier for super object |
| (int) NumberOfPointers | number of pointer values |
| (int) NumberOfDataMembers | number of data members |
| (int) NumberOfArrayMembers | number of array data members |
| (int) LengthOfClassName | length in bytes of name |
| (int) DataMemberFormatId | format ID of data members |
| (int) ArrayDataMemberFormatId | format ID of array data 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.

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

| 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 2: Object Identifiers of Primitive Format Objects

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 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 4. Figure 4 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.

### 3.2.2 Create Object

| | | | |
|---|---|---|---|
| Event: | <create_object> | | |
| Parameters: | (int) | FormatId | format object ID |
| | (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.

### 3.2.3 Create Array Object

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

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.

**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 | Old (DO) | 0 | 0 | 2 | 1 | 12 |
|---|---|---|---|---|---|---|

| Old (int) | Old (int) | Old (array of char) | 10 | DesignObject |
|---|---|---|---|---|

**PTF Object Format**

| fo | Old (CP) | Old (DO) | 5 | 0 | 0 | 13 | CompositePart |
|---|---|---|---|---|---|---|---|

**Object Format of the Persistent Store**

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

Figure 4: Three Levels of Object Layout Descriptions.

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.

### 3.2.4 Delete Object

| | | | |
|---|---|---|---|
| Event: | `<delete_object>` | | |
| Parameters: | (int) | FormatId | format object ID |
| | (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.

### 3.2.5 Edge Write

| | | | |
|---|---|---|---|
| Event: | `<edge_write>` | | |
| Parameters: | (int) | FormatId | format object ID |
| | (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.

### 3.2.6 Edge Read

| | | | |
|---|---|---|---|
| Event: | `<edge_read>` | | |
| Parameters: | (int) | FormatId | format object ID |
| | (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.

### 3.2.7 Data Write

| | | | |
|---|---|---|---|
| Event: | `<data_write>` | | |
| Parameters: | (int) | FormatId | format object ID |
| | (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.

### 3.2.8 Data Read

| | | | |
|---|---|---|---|
| Event: | `<data_read>` | | |
| Parameters: | (int) | FormatId | format object ID |
| | (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.

### 3.2.9 Array Data Write

| | | | |
|---|---|---|---|
| Event: | `<array_data_write>` | | |
| Parameters: | (int) | FormatId | format object ID |
| | (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.

### 3.2.10 Array Data Read

| | | | |
|---|---|---|---|
| Event: | `<array_data_read>` | | |
| Parameters: | (int) | FormatId | format object ID |
| | (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 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.

#### 3.2.11 Set Root

| | | | |
|---|---|---|---|
| Event: | `<set_root>` | | |
| Parameters: | (int) | FormatId | format object ID |
| 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.

#### 3.2.12 Get Root

Event: `<get_root>`

This event captures a reference to the super root object.

### 3.3 Garbage Collection Directives

#### 3.3.1 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.

#### 3.3.2 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.

# 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.

## 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.

## 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 2.

# 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 [8]) 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.

Our PTF design is most closely related to the work of Scheuerl et al., who developed the MaStA I/O trace format [9] 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.

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. It also does not support transaction events. We are now in the process of defining appropriate support for transactions for the next version of PTF. Furthermore, some objects and data structures are part of the POS implementation (e.g., indices, 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 indices, we do not support the manipulation of indices. We envision adding some high-level trace events (such as `create_index` and `update_index`) to capture operations on indices in a later version.

In conclusion, our immediate future work consists of three major focuses. The first focus is to extend PTF to support the generation of trace events that capture concurrency within multi-user workloads. In the first version we were able to do meaningful experimentation in the area of garbage collection without data values; however we realize that some experimentation may require data values. Thus, a second focus is to standardize the format of data that may be included with trace events. The third focus is to extend PTF to capture the manipulations on indices.

# References


[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] T.O. Humphries, A.W. Klauser, A.L. Wolf, and B.G. Zorn. An Infrastructure for Generating and Sharing Experimental Workloads for Persistent Object Systems. *Software-Practice and Experience*. To appear.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.