A Dossier Driven Persistent Objects Facility

Robert Mecklenburg, Charles Clark, Gary Lindstrom and Benny Yih

University of Utah Center for Software Science
Department of Computer Science
Salt Lake City, UT 84112

E-mail: {mecklen,clark,gary,yih}@cs.utah.edu

Abstract

We describe the design and implementation of a persistent object storage facility based on a dossier driven approach. Objects are characterized by dossiers which describe both their language defined and "extra-linguistic" properties. These dossiers are generated by a C++ preprocessor in concert with an augmented, but completely C++ compatible, class description language. The design places very few burdens on the application programmer and can be used without altering the data member layout of application objects or inheriting from special classes. The storage format is kept simple to allow the use of a variety of data storage backends. In addition, these dossiers can be used to implement (or augment) a run-time typing facility compatible with the proposed ANSI C++ standard. Finally, by providing a generic object to byte stream conversion the persistent object facility can also be used in conjunction with an interprocess communication facility to provide object-level communication between processes.[1]

1 Motivation

The basic problem of a persistent object store (POS) is simply stated:

> Given a reference to the root node of a graph of objects generate a data stream which can be used to reconstitute the original object graph at a later time.

Many approaches have been pursued to solve this basic problem (see Section 12 for a summary). The utility of these approaches is governed by the constraints they impose on application code in such dimensions as (i) language or compiler extensions, (ii) mandatory inheritance from library base classes, (iii) system transformation of application source code, (iv) expansion of object size, (v) mandatory presence of virtual function tables, and (vi) programmer declaration of supporting functions and observance of programming style restrictions.

We describe a new approach which poses no constraints in (i) – (v), and minor client obligations in (vi). Our approach is based on preprocessor-generated dossier objects[14], which drive fully polymorphic (i.e., applicable to all types) load and store functions. In addition to supporting object persistence, our approach provides a fully general means for transporting object graphs in address space independent form (i.e., "pickled", with "unswizzled" pointers). Our design has been motivated by the stringent demands of a large (750,000 line) C++ CAD/CAM/visualization application[2].

[1] This research was sponsored in part by the Advanced Research Projects Agency (DOD), monitored by the Department of the Navy, Office of the Chief of Naval Research, under Grant number N00014–91–J–4046. The opinions and conclusions contained in this document are those of the authors and should not be interpreted as representing official views or policies, either expressed or implied, of ARPA, or the U.S. Government.

2 What Is An Object?

We begin by defining our unit of persistence, which we term an *object*. While some approaches take this to be C++ class instances, this basis is too narrow for applications such as our CAD client, which make extensive use of graphs of vectors and structures, with semantically significant sharing relationships. Hence we define an *object* to be a contiguous region of memory whose type is known either through

- static type information,
- dynamic type information (e.g., virtual function table), or
- information provided by the application programmer.

An object is identified in an application by a pointer or reference to its first address along with some notion of its bounds (derived from type information). We explicitly disallow pointers to the interior of objects. An *object graph* consists of a collection of objects formed into an arbitrary graph by pointers embedded in the objects. An object is identified in the persistent store by a unique *object identifier* (OID). An application requests objects by OID and can access the OID of an object given its virtual address in the application.

One consequence of this definition is that data members of objects cannot be read or written independently of the containing object. This might occur when a data member is passed as an argument to a function, then saved. Given that we know the type and size of the data member this restriction might be lifted, however, we have not yet had occasion to do so.

3 Client Constraints

To be as convenient as possible a POS must minimize the impact of its use on application source code and the software development process while at the same time maximizing functionality. Among the features of a POS, we feel the following to be important: minimal impact on object layout and class declarations, allow the use of standard language tools, provide object access from a variety of hardware platforms, provide object access after class mutation. We discuss each of these requirements in turn.

The POS should not require "large" changes to class definitions. In particular, any system which requires altering the class layout by adding data members, virtual functions (where none existed before) or additional base classes is unacceptable. Such a system would impose storage overhead and incompatibilities intolerable to many applications. However, adding virtual functions to a class with an existing virtual function table is an acceptable change which would allow more convenient use of the storage facility. If this modification were allowed (but optional) it would provide a convenient interface for application specific classes while still allowing library classes (for which there is no source code) to persist.

One of the biggest problems caused by many POSs is the requirement for non-standard language tools (e.g., special compilers) to enable objects to persist. These tools either parse an extended language syntax (translating into standard C++), generate augmented class implementations, or some mixture of the two. Our group, having worked on large software projects using these approaches and finding them burdensome, chose to require the class definition be written in standard C++. This means that there is only one class definition (with no additional semantic information in other files) and that applications can be compiled and run (albeit without persistence) with or without the persistent objects facility. This significantly simplifies porting, piece-wise development and testing of applications.

Once a POS is integrated into an application or organization its use quickly becomes fundamental to the project and the persistent objects themselves become a valuable resource. As such, it is often unacceptable to abandon the database when new hardware or software is acquired or when class definitions change. Furthermore, as the size of the database grows evolving the data *en masse* becomes a significant burden. We feel a more reasonable approach is to integrate platform heterogeneity and type evolution cleanly into the persistent store allowing for lazy transformation of objects to the reader's requirements.

We discuss other, less major, constraints on the POS as they arise.

4 An Object Description Language

We now address the need for a language in which to describe objects. An object which is an instance of a primitive C++ type may be described simply by its standard type name. One may reasonably expect that an object which is an instance of a class may be described by the C++ declaration of that class. Indeed, to a first approximation, that is correct. Unfortunately, there are several "extra-linguistic" patterns of use which are not sufficiently described by standard C++ syntax, particularly with respect to dynamically sized objects (e.g., strings and other vectors). The problem is to identify important idioms required by applications and to provide an annotation mechanism which does not invalidate the use of standard language tools. In addition to these annotations, the POS may require classes to provide various semantic handles to allow storage and retrieval.

The most important idiom in C++ which is not adequately described by class declarations is the use of pointers to access dynamically sized regions of memory. Strictly interpreted, the declaration:

```
char *cp;
```

identifies a pointer to an unknown number of characters. By convention the number of characters is determined by a *sentinel value*, in this case the null character. The sentinel value technique for dynamically sized data can be used with any data type, but is most typically used with pointers and integral types where the zero bit pattern is the most common sentinel. A competing style for identifying the size of dynamically sized memory regions relies on a pair of data values:

```
int   n;     // size of cp
char *cp;
```

where the dynamic size is stored explicitly in a separate data member.

Static data members of a class pose a different sort of problem for a POS. Indeed, one may question whether static data members should persist at all. Often these data members are used to resolve issues inherent in run-time data management. For instance, an application might maintain an *extent* list of all allocated instances. Such a list acquires a completely different meaning in a persistent store owing to the shared, distributed, and concurrent nature of the store. Our approach is to allow the application programmer to indicate whether static data members should persist. However, we chose not to manage concurrent access. Aside from ensuring consistent concurrent writes for single data members we do not assume any further capabilities of the underlying POS such as notifying readers of updates to shared data. Similar to static data members there may be non-static data members which the programmer does not want saved. For example, an object might contain a pointer to a buffered file structure which has no meaning (or a different meaning) when stored in a POS. These nodes can be annotated as *orphaned* objects; their value will not be stored and their pointers will not be traversed.

Unions present an interesting problem due to the ambiguous nature of the type information available. In particular, if a union contains both a pointer and a non-pointer, should the pointer be traversed? The current approach is to require unions to be enclosed in a class containing at least the union and an integer which is used as the union discriminator. We feel this is a reasonable compromise between a completely arbitrary decision (on our part) and completely user defined behavior (which we have no way of specifying).

Finally, pointers to member functions are not yet supported. Since the implementation of pointers to member functions does not require a virtual memory address, it may be easy to save and restore them. Unfortunately, the possible variance in their implementation makes any one storage technique non-portable. It may be possible to encode the pointer to member function implementation along with the pointer itself in the persistent object, but we have not investigated this technique.

4.1 Syntactic Considerations

How can these annotations be applied to a class definition if standard compilers are used and no additional files are consulted? There are three basic approaches: parameterized classes, embedded annotations in comments, or augmented identifier names. These would be used to identify the three basic annotations discussed above:

- dynamically sized arrays terminated by a sentinel,
- dynamically sized arrays whose size is defined by an associated integer, and
- objects which are to be omitted from the persistent store (orphaned).

Templates could be used to identify data members with these attributes by defining a template class for each of the annotations. For instance, a simplified template for a null terminated array might be:

```
template <class T>
class null_terminated {
    T *p;
public:
    operator T*() { return p; }
};
```

This would then be used in an application by replacing a simple pointer declaration with:

```
class X {
    null_terminated<char>   cp;
    ...
```

Unfortunately, this approach introduces all the problems associated with a smart pointer class[9]. In particular, our template class does not interact well with the `const` keyword, nor is it guaranteed to have equivalent performance characteristics. Applying this class to existing programs would entail significant source code (and possibly algorithmic) changes. Although we feel this is a syntactically elegant solution to the annotation problem, it is only useful in a restricted domain (e.g., writing new applications).

The second approach to annotating classes places comments adjacent to data members containing keywords identifying various attributes. Similarly, the third approach uses the data member name itself (or its type name) to contain the attribute. An example of the later is:

```
typedef char char__null; // Null terminated string.
char__null *path;

typedef int int__sized; // Integer sized string.
int__sized n;
char *     name;
```

```
// reconstructor_t - Type used to identify the reconstructor.
enum reconstructor_t { RECONSTRUCT };

// char__null - Annotated type for a null terminated array of char.
typedef char char__null;

// int__sized - Annotated type for an integer sized array.
typedef int int__sized;

// dictionary_c - A simple association table.
class dictionary_c {
 public:
    dictionary_c( reconstructor_t );
    ~dictionary_c();
    ...
    dossier_c *         __get_dossier() const;
    void                __load_store_hook( int when );

 private:
    char__null *        name_;  // The dictionary name.
    int__sized          len_;   // The size of the table.
    dict_elem_c *       table_; // The association table.
};
```

Figure 1: A typical class with annotations.

We implemented this last technique for several reasons. We consider the annotations themselves to be an essential part of the type information which a programmer must usually omit due to limitations in C++ declaration syntax. By augmenting the type declaration we are making this type information explicit at the appropriate time and place. Also, it does not interfere with a standard commenting style for class declarations. Finally, it allows us to experiment with a novel annotation technique which we have not seen used before. Annotating the type of the data member (rather than the member itself) leaves the application programmer free to select meaningful member names unencumbered by the annotations. The currently supported annotations are:

`__null`	dynamically sized, zero terminated
`__sized`	dynamically sized, this member is the size, following member is the pointer
`__orph`	an orphaned object, don't save

Figure 1 shows a simple dictionary class augmented with several annotations.

4.2 Application Object Services

To recreate the original semantics of a persistent object the POS must be able to request certain services of the object. Most importantly, that of object allocation and creation. Likewise, the object may require that the persistent store relinquish program control to the object at special times, often just prior to storage and just after loading.

Often the implementations of objects have highly specific meanings associated with the application or environment which do not persist well. Examples of such problems include storing hash tables and file handles. As with other members the writer of the object must annotate the stored instance with information allowing the reader to reconstitute a similar object with semantics equivalent to the original object. For a hash table, the reader may have a different hash function or table size and therefore must rehash the members of the table. For a file handle, the reader must find and open the file and set the current position. An annotation on a declaration cannot transmit this information (and indeed, may not have the information to transmit). To allow for this type of application specific behavior the programmer can define load and store *hooks* which are called by the POS during object I/O. The load/store hook has a special name and type signature recognized by the dossier generator:

```
void __load_store_hook( int when );
```

This member function is added to the class declaration of any class requiring special handling during I/O. The function can be called under three circumstances (indicated by the `when` parameter): after loading an object, before storing an object, and after storing an object. Figure 2 shows a typical load/store hook for those classes requiring one.

When an object is restored from the POS several application and implementation specific initializations must be performed. The most obvious of these is setting the virtual function table pointer. This can be done in a variety of ways: from using the `new` placement syntax and having the application programmer invoke the constructor to copying the pointer from an initialized sample instance. The later approach does not allow for the application to gain control during object allocation and is therefore unacceptable. Using the `new` placement syntax has the problem of compatibility with other software packages (including the application's classes). A compromise requires the application class to define a special constructor which we call the *reconstructor*. This approach allows classes to overload `new` and `delete` and to gain control during object construction. The reconstructor is identified by its type signature:

```
<class_name>( reconstructor_t );
```

In fact, the reconstructor can be omitted if there exists a default constructor which performs the same function. That is, the default constructor does not have any unwanted side effects and does not assume that the initial values of the object will be seen by the client application code.

Figure 2 shows the typical implementation of a reconstructor. Reconstructors usually have no body since their only duties are to invoke the class (or application) specific memory allocator and to set the virtual function table pointer(s). The actual data members will be overwritten with values from the loaded persistent object.

Finally, to allow convenient use of the POS with polymorphic objects we encourage the application programmer to declare a virtual function for accessing the dossier of a class:

```
virtual dossier_c *__get_dossier() const;
```

This allows the application and POS interface to access the dossier of conforming objects simply. For objects which do not support the `__get_dossier` member function, the application must provide the dossier handle explicitly. This is done by calling a dossier lookup function which accepts the string name of the requested class and returns a pointer to the dossier. These interfaces allow simple and convenient access for classes under application programmer control, while still allowing other classes to persist. After the dossier for the root object is obtained, dossiers for other objects in the graph can be accessed through the root object dossier.

Once an application's class declarations (e.g., `.h` files) have been adapted to express these extra-linguistic features, they become the application's class description. These files are read and analyzed by a preprocessor based on the C++ grammar written by James Roskind[23]. The preprocessor emits auxiliary C++ files which construct instances of class dossiers embodying the class descriptions, including associated annotations. These emitted files are compiled and linked, along with a support library, into an application to implement the client side of the POS. Note that client source files are only read, not transformed, in this process. The application causes an object to persist through an explicit `store` function call. Similarly, objects are loaded from the persistent store by calling a `load` function with the appropriate OID.

```
dictionary_c::dictionary_c( reconstructor_t )
{
}

void dictionary_c::__load_store_hook( int when )
{
    switch ( when )
    {
    case 0: // After loading.
        // Resort the table using current criteria.
        sort_table();
        break;
    case 1: // Before storing.
        break;
    case 2: // After storing.
        break;
    }
}
```

Figure 2: A typical reconstructor and load/store hook.

5 Capture of Compiler and Platform Characteristics

To build a complete description of objects, including data member layout, the dossier generator must mirror the algorithms of the current compiler and would therefore not be particularly portable. We avoid this problem by separating the dossier into machine/compiler independent and dependent portions. The compiler independent portion is constructed by the dossier generator while the dependent portion is computed at run-time from auto-configuring code written into the dossier initializer. The compiler and machine dependent structures gather three types of information: size and format of data types, location of data members in objects, and handles on member functions. We discuss each briefly.

To allow dossier code to read and write objects on differing platforms (both hardware and software) the polymorphic I/O code must know the size of each data type and its format when written to a persistent store. Size information is easily acquired through the use of the `sizeof` compiler directive. Also, byte order and floating point format must be determined. In the worst case, these characteristics must be explicitly specified for each platform making the dossier source code non-portable. In the normal case, however, byte order can be determined through simple calculations and floating point format can be acquired through host configuration files.

The location of data members and base classes for an object are determined using a technique similar to the ANSI C `offsetof` macro. For each (non-static) data member, its location is determined by taking its address and subtracting the object's base address. This requires that the dossier initializer be either a friend or member function of the class. Base class offsets are calculated similarly by casting a "pointer to derived class" to a "pointer to base class". For example, if class `D` derives from class `B`, the expression:

```
((B *)((D *)8)) - 8
```

returns the offset of a `B` instance within a `D` instance. (The use of a non-zero base address subverts optimizations in various compilers.) This expression is portable across all platforms (that we are aware of)[10].

Finally, the polymorphic I/O operations must invoke class reconstructors and load/store hooks to perform their functions. Since the address of a constructor cannot be computed, we wrap the reconstructor in a simple C++ function and store its address in the dossier. For uniformity we use the same technique to store the load/store hook in the dossier.

6 The Storage Algorithm

The basic storage algorithm is a simple graph traversal driven by the graph's root object and the dossiers. We begin by retrieving the OID of the object to be saved. If the object does not have an OID, allocate one. Next place the object and its OID into the queue of objects waiting to be processed. The rest of the algorithm proceeds as follows:

Algorithm 1

```
dequeue the next node to process
if the node is unsaved
    run the pre-store hook
    mark the object as saved
    enqueue all embedded pointers (allocate OIDs, if necessary)
    store the dossier, if necessary
    store the object and dossier OIDs, and machine id
    store the object
    store the OID of the target of every embedded pointer
    run the post-store hook
```

Dossiers are just objects so they are stored, along with the objects they describe, using the same algorithm. Of course, only one copy of the same dossier is stored and that dossier is referenced by all instances of that class through its OID. Since a dossier is an object, to be read and written it must have a descriptor, or *meta-dossier*. This meta-dossier is a permanent component in the support library and is never written to or read from a POS or communication channel. The meta-dossier is generated by running the dossier generator over its own data structures.

The storage format is designed to be "retargetable" to different object storage engines and is therefore a mix of low-level formats and high-level information. The storage engines currently in use are a transactional DBM and a simple Unix file interface (an Exodus interface is planned). Writing is performed in the simplest possible way, by copying the machine representation of each data member value to the POS. It is the responsibility of the reader to decipher the writer's format. Since objects are often read and written on a single platform this proves reasonably efficient for local communication and temporary storage.

Retrieving object graphs is similar. The retrieval is initiated by the application with the OID of the root node of an object graph. This node is entered into a queue of nodes yet to be read and proceeds as follows:

Algorithm 2

```
dequeue the next node to process
if the node is not yet read
    load the dossier of the object
    load the binary image of the object
    invoke the reconstructor to allocate memory for the object
    record the new object's address and OID
    copy the values of data members from the binary image to the new object
    for each pointer member set the new address, if available
        if not available, place pointer member on patch queue
    run the post-load hook
else
    return the address of the object
traverse patch queue, setting remaining pointer members
```

The object is loaded as a set of binary values from the original object. The dossier is used to pick through this bag of bits to identify data members and their values. The new values for pointers are accessed by the OID of the target object. Due to cyclic graph structures some objects will not have been read yet, so pointers to these objects must be queued until the desired object has been read.

7 Heterogeneity

Heterogeneity is handled by providing a machine description object which contains information concerning hardware and compiler specific data. In Algorithm 1 a machine identifier is stored along with the OIDs of the object and its dossier. This machine identifier references a structure describing the hardware characteristics (e.g., byte order, floating point format) and software characteristics (e.g., member layout) of the writer. When the data for an object is copied from the binary image of the writer to the run-time memory allocated for the reader machine dependent translations are performed.

Although the translations from one hardware platform to another must be hand-crafted, the actual process of converting values from one format to the other is controlled through the dossiers. To avoid writing n^2 conversion routines a standard intermediate format can be used to reduce the number of conversion routines to $2n$.

8 Object Evolution

Invariably, the classes for objects stored in the POS will change due to changes in the user's requirements and added functionality. It is important that old data continue to be accessible to current applications. There are three basic approaches to evolving an object instance from one class declaration to another:

1. provide accessor functions,
2. copy using a "static" algorithm,
3. copy using a "dynamic" algorithm.

The first technique requires that an application be enhanced with accessors that know the old and new type and offset of the desired data member. This accessor is invoked on the old object and returns a value as if from a new object. This is unsuitable for many applications due to its highly hand-crafted nature. The second technique uses the dossier of the old and new objects to copy data member values one by one from the old to the new object using some fixed algorithm. Types that have changed may be converted if the conversion is sufficiently simple (e.g., int to float) and discarded otherwise (assuming that the old value has no translation). New data members may be initialized to some default value (e.g., zero). Experience with one large project indicates that this is a useful evolution technique for many simple object transformations[16]. Nevertheless, it is insufficient as the only (or even primary) type evolution mechanism. The final technique allows the application programmer to provide a function to translate an object from one version of a class to another.

Dossiers can be annotated with version information and can record translation functions capable of converting from one version of an object to another. These translation functions would be written by application programmers when class definitions are modified. The dossier driven type evolution system can then chain conversion functions to evolve from one version of an object to the next until the desired version has been computed. A mixture of the second and third techniques described above is being implemented for our POS.

9 Other Applications of Dossiers

Once a dossier generator is available several other applications become immediately apparent. Two of these applications are run-time type information and remote procedure call generation. There are essentially three options for using the proposed run-time type information feature[32] with dossiers. First, as Stroustrup suggests, the RTTI system can be queried to determine a type name which is then used as a key to access auxiliary information:

```
dossier_c *dp = lookup_dossier( typeid(*p).name() );
```

This has the obvious advantage that it uses only standard language features and is thus portable across all implementations.

Second, we could derive the `dossier_c` class from `Type_info` itself and cause `dynamic_cast<T>` and `typeid()` to return `dossier_c` instances. This would allow both the persistent object support library and applications to use extended type information directly through language supported mechanisms. Unfortunately, a preprocessor/support library approach to RTTI cannot be implemented portably owing to the variance in RTTI implementations. If the `dynamic_cast<T>` and `typeid` language features are implemented with support functions, then it would be possible to replace them with new versions returning references or pointers to dossiers. The dossier constructors could be enhanced to maintain any state in the base `Type_info` object required by the RTTI implementation. If, however, either of the RTTI constructs are implemented as inline code we see no mechanism, short of modifying the compiler, for substituting dossiers for `Type_info` objects.

The third technique would use a hybrid of the first two. The `Type_info` class could be extended with new (non-virtual) member functions (either through inheritance or direct modification) to support the functionality of dossiers. These member functions could use the type information in the `Type_info` object to access the dossier through a lookup table and return the appropriate values. Thus, to the user, it would appear that the `Type_info` object contained extended type information when, in fact, it did not. This approach has the advantage of simplicity and portability.

A dossier generator can also be used to build a remote procedure call (RPC) facility. One approach would be to enhance the generator itself to write RPC stubs which would be linked into the application. This would require parsing general function declarations (member and non-member) and possibly adding additional annotations for in, out, and in/out parameters. Our generator already performs this parsing. This implementation would render a powerful and convenient implementation of standard RPC. Another technique would be to implement a polymorphic RPC dispatcher capable of dynamically marshalling and unmarshalling arbitrary argument lists. This would allow advertising and accessing services dynamically and may be the basis for a CORBA-like object broker.

10 Current Status

The dossier generator, *goofie* (a General Object-Oriented Framework for Interface Expression), is largely complete. Goofie can generate dossiers for a large subset of C++ including all annotations described above. The omissions are due mainly to the highly decomposed nature of the Roskind grammar (i.e., rare or obscure grammar productions have not been fleshed out). An initial version of the polymorphic load and store code is complete (for a single platform) and is able to read and write objects and dossiers. The interface to the persistent store has been defined and two distinct stores have been implemented. The first uses a version of DBM supporting transaction semantics. The other converts objects to a serial byte stream for use across interprocess communication channels. We plan to add an interface to the EXODUS storage manager[4] shortly.

Although the design described here is quite general there are a number of limitations in the current system. Most important, we do not support pointers to the interior of objects (although the load store hooks allow crude handling of some cases). We also do not support unions or pointers to member functions in the current system. Only two styles of dynamically sized data members are supported although many others can be envisioned. We are dissatisfied with the treatment of static data members mainly due to the uncertain semantics of persistent, shared members.

In terms of portability and simplicity of the solution there are several short comings. Of these, the most important is the requirement that the application programmer alter class definitions to include a reconstructor (optional), load/store hooks (optional), and the dossier accessor function (optional) or friend declaration. We see no solution to this problem given the initial problem constraints. Another problem is the possibility that the byte order and floating point format must be explicitly indicated in the dossier making it non-portable.

11 Future Work

The most important features currently unavailable in our system are heterogeneity and class evolution. To provide a universal and stable POS these are fundamental requirements. The design of these features is largely complete and an initial implementation should be completed soon. We hope to support both the simple static evolution algorithm used in [16] and the dynamic one described in Section 8. We are also investigating the ability to lazily load individual nodes of the object graph. Given our current implementation constraints this will probably require complete object encapsulation. In addition, dynamically loading class definitions in the form of dossiers and member functions is possible through the use of our object/meta-object server[19].

A portable, comprehensive dossier facility has applications in a variety of areas. Two applications related to our research are inter-language object transmission[17] and dynamic reconfiguration of software systems[5].

12 Related Work

Persistence for C++ systems has been the focus of vigorous and diverse research and development activity. Several commercial products, notably object-oriented database systems (e.g., [15]), provide persistence as a C++ extension. In addition, there are several experimental systems such as Arjuna which provide comprehensive support for persistent C++ objects.

Tables 1 and 2 summarize representative systems in terms of six distinguishing dimensions (see column headings). These correspond to important decisions which must be resolved by any persistent C++ system designer. We consider each in turn, offering a few clarifying comments. Further details are available from the references cited in each case.

Object description language: Several systems exploit C++ language extensions to describe persistent objects (Avalon, O++, OBST, SOS). Typically, these involve new key words or syntactic extensions. Arjuna and EC++ support a subset of full C++. For systems relying on persistent virtual memory (C**, E, and the Texas system), the C++ class definitions suffice for object description, though ObjectStore uses a database schema declaration facility for class evolution control. Similarly, the NIH class library, being ASCII file oriented, requires no object description language.

Dossier objects: Run-time information describing persistent objects is utilized by O++, ObjectStore, C**, and the Texas system. This information is captured in dossier objects in all but the Texas system, which uses a tabular representation. The remaining systems do not exploit dossier information.

Preprocessor use: Like the Utah approach, several systems use preprocessors to collect object description information. These include ObjectStore (optionally), OBST, Arjuna, Avalon, C**, EC++, and the Texas system. Three systems (E, SOS, and O++) rely on modified compilers.

Invocation of object storage and retrieval: A wide variety of techniques are relied upon for causing persistent objects to be saved and restored. The C++ option of overloaded `new` (i.e., placement syntax) is exploited by O++, ObjectStore, SOS and the Texas system. Reliance on a special base class conferring persistence is utilized by O++, SOS, Arjuna, Avalon, EC++, and the

NIH class library. OBST, C**, Avalon, and E support involves keywords, object registration or parallel class. Like the Utah approach, the NIH class library provides explicit object read and write operations.

Implementation of storage and retrieval services: A wide variety of approaches are employed for implementing object dereferencing, copying, sharing, and inter-process transmission. Seamless pointer swizzling by page faulting is a principal advantage of persistent virtual memory based systems (ObjectStore, C** and E). Other systems rely on distributed processing, with special RPC-based services such as object identifier creation, binding and dereferencing. SOS uses a special persistent object pointer class, with faulting semantics. Systems providing transaction semantics include ObjectStore, OBST, and Avalon.

Transitive closure of object storage and retrieval: Finally, systems differ on whether object save operations include saving all referenced objects, i.e. saving object graphs, rather than individual objects. The point is moot for persistent virtual memory systems such as ObjectStore and C**. Other systems use special pointers, or named roots, to control save transitivity. Inline code controlling read/write depth is utilized by Avalon, EC++ and the NIH class library.

13 Conclusions

Using dossiers as the foundation for a persistent object store we have built a flexible, portable storage facility capable of supporting class evolution and platform heterogeneity. The requirements of the facility are such that any compiler compliant with the proposed ANSI C++ standard can be used to build applications with persistent objects. Our dossier generator, *goofie*, requires minimal alteration of application class descriptions and can be used where library source code is not available. In particular, the burden on the application programmer can be summarized as:

- pointers to dynamically sized memory must be annotated;
- a reconstructor must be added to the class or the default constructor must not have unwanted side effects;
- load/store hooks must be written for objects whose data values are application dependent; and,
- a virtual `__get_dossier` function should be added to a class, or the non-virtual `__get_dossier` function must be made a friend, or the class's data members must be publicly readable.

In many interesting classes the actual source code change is the addition of a friend declaration to allow access by the `__get_dossier` function.

The ability to apply this persistent store to large, existing software systems is an important aspect of our design and implementation. The dossiers generated for application objects can also be accessed from the proposed run-time type information system and can be used by the programmer to build application specific polymorphic functions. The ability to manipulate objects polymorphically allows us to serialize arbitrary object graphs and restore them providing the basis for inter-process object transmission and RPC stub generation. A prototype of the dossier generator, polymorphic I/O code, and object store are complete and work is continuing to enhance their functionality.

14 Acknowledgements

We gratefully acknowledge the contributions of the members of the Mach Shared Objects project. In particular, we would like to thank Mark Swanson, Jay Lepreau, and Doug Orr whose insight and assistance made this work possible. We would also like to thank the members of the Alpha_1 project who gave us their cooperation, support and creativity, especially Beth Cobb, Tim Mueller, Russ Fish, and Mark Bloomenthal.

System	Description Language	Dossiers	Preprocessor	Invocation	Implementation	Graph Traversal
Arjuna [8, 29]	Restricted C++	no	yes	special base class	rpc	no
Avalon [11]	Augmented C++	no	yes	special base class, `stable` keyword	rpc w/ transactions	inline code in r/w
C** [3, 18]	none	yes, not user visible	yes	object register method	vm and pointer swizzling	yes
E [20, 22, 21]	none	no	modified g++	parallel class hierarchy	vm and pointer swizzling	n/a
EC++ [25, 31]	Restricted C++	no	yes	named object, special base class	rpc	inline code in r/w
NIHCL [13]	none	no	no	special base class, r/w functions	ASCII files	inline code in r/w

Table 1: Summary of persistent objects systems and their approach.

System	Description Language	Dossiers	Preprocessor	Invocation	Implementation	Graph Traversal
O++ [12, 7, 1]	Augmented C++	yes	compiler	overloaded new, special base class	tagged byte stream	yes
ObjectStore [15]	DB schema	yes	yes	overloaded new	vm and pointer swizzling	no
OBST [6, 33, 24]	Augmented C++	no	yes	create in container object	copied on container commit	names as roots
SOS [27, 26, 28]	Augmented C++	no	modified g++	special base class, overloaded new	object fault on special pointer class	special pointer class
Texas [30, 34]	a.out	yes, packed in 2 tables	tdesc	overloaded new	vm and pointer swizzling	n/a?
Utah Dossiers	Augmented C++	yes, persistent objects	Roskind grammar-based (goofie)	r/w functions	distributed rpc w/ tagged byte stream	dossier driven

Table 2: Summary of persistent objects systems and their approach (Continued).

15 Availability

The software described in this paper is available through anonymous ftp from `ftp.cs.utah.edu`. The distribution is a Unix compressed tar file, `pub/goofie.tar.Z`. This paper is included in the distribution. The software and paper are also available from the World Wide Web under the URL `http://www.cs.utah.edu/projects/mso/goofie/goofie.html`.

References

[1] Rakesh Agrawal, Shaul Dar, and Narain H. Gehani. The O++ database programming language: Implementation and experience. In *Proceedings of the IEEE 9th International Conference on Data Engineering*. IEEE Computer Press, 1993.

[2] Alpha_1 Project. Integrated computer aided design and manufacturing: An overview of Alpha_1. Technical report, University of Utah, Dept. of Computer Science, March 5, 1992.

[3] Vinny Cahill, Chris Horn, Andre Kramer, Maurice Martin, and Gradimir Starovic. C** and Eiffel**: Languages for distribution and persistence. In *Proceedings of the 1990 OSF Microkernel Applications Workshop*, Grenoble, France, 1990.

[4] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Storage management for objects in EXODUS. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 341–369. Addison-Wesley, 1989.

[5] John B. Carter, Bryan Ford, Mike Hibler, Ravindra Kuramkote, Jeffrey Law, Jay Lepreau, Douglas B. Orr, Leigh Stoller, and Mark Swanson. FLEX: A tool for building efficient and flexible systems. In *Proc. Fourth Workshop on Workstation Operating Systems*, October 1993.

[6] Eduardo Casais, Michael Ranft, Bernhard Schiefer, Dietmar Theobald, and Walter Zimmer. OBST — An overview. Technical report, Forschungszentrum Informatik (FZI), D-76131 Karlsruhe, Germany, 1993.

[7] S. Dar, N. H. Gehani, and H. V. Jagadish. CQL++: A SQL for a C++ based object-oriented DBMS. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Advances in Database Technology — EDBT '92: Proceedings of the 3rd International Conference on Extending Database Technology*, Vienna, Austria, March, 1992, 1992. Springer-Verlag.

[8] G.N. Dixon, G.D. Parrington, S.K. Shrivastava, and S.M. Wheater. The treatment of persistent objects in Arjuna. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 169–189, University of Nottingham, July 10–14, 1989. Cambridge University Press.

[9] Daniel R. Edelson. Smart pointers: They're smart, but they're not pointers. In *USENIX C++ Conference Proceedings*, pages 1–20, Portland, Oregon, August 1992. The USENIX Association.

[10] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[11] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility*. Data Management Systems. Morgan Kaufmann Publishers, Menlo Park, CA, 1991.

[12] N. H. Gehani. OdeFS: A file system interface to an object-oriented database. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey 07974, 1989.

[13] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley & Sons, 1990.

[14] John A. Interrante and Mark A. Linton. Runtime access to type information in C++. In *USENIX Proceedings C++ Conference*, pages 233–240. USENIX Association, 1990.

[15] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[16] Robert W. Mecklenburg. The specification for a binary file format for Alpha_1 models. Alpha_1 technical report 88-6, University of Utah, 1988.

[17] Robert W. Mecklenburg. *Towards a Language Independent Object System.* PhD thesis, University of Utah, Salt Lake City, Utah, June 1991.

[18] Michael Mock, Reinhold Kroeger, and Vinny Cahill. Implementing atomic objects with the RelaX transaction facility. *Computing Systems*, 5(3):259–304, Summer 1992.

[19] Douglas B. Orr and Robert W. Mecklenburg. OMOS — An object server for program execution. In *Proc. International Workshop on Object Oriented Operating Systems*, pages 200–209, Paris, September 1992. IEEE Computer Society. Also available as technical report UUCS-92-033.

[20] Joel E. Richardson and Michael J. Carey. Persistence in the E language: Issues and implementation. *Software—Practice and Experience*, 19(12):1115–1150, December 1989.

[21] Joel E. Richardson and Michael J. Carey. Implementing persistence in E. In John Rosenberg and David Koch, editors, *Persistent Object Systems: Proceedings of the Third International Workshop*, Workshops in Computing, pages 175–199. Springer-Verlag, Newcastle, Australia, January 10–13, 1989, 1990.

[22] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. The design of the E programming language. Technical Report 814, Computer Science Department, University of Wisconsin, Madison, WI, February 1989.

[23] Jim Roskind. A yacc-able C++ 2.1 grammar, and the resulting ambiguities. July 1991.

[24] Bernhard Schiefer, Dietmar Theobald, and Jürgen Uhl. User's guide: OBST release 3.3. Technical report, Forschungszentrum Informatik (FZI), D-76131 Karlsruhe, Germany, July 1993.

[25] Manuel Sequeira and José Alves Marques. Can C++ be used for programming distributed and persistent objects? In *Proceedings 1991 International Workshop on Object Orientation in Operating Systems*, pages 173–176, Palo Alto, CA, October 17–18, 1991. IEEE Computer Society Press.

[26] Marc Shapiro. Prototyping a distributed object-oriented operating system on Unix. In *Proceedings of the First USENIX/SERC Workshop on Experiences with Distributed and Multiprocesor Systems*, pages 311–331, Fort Lauderdale, FL, October 5–6, 1989. Usenix Association.

[27] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating systems—Assessment and perspectives. *Computing Systems*, 2(4):287–337, Fall 1989.

[28] Marc Shapiro and Laurence Mosseri. A simple object storage system. In John Rosenberg and David Koch, editors, *Persistent Object Systems: Proceedings of the Third International Workshop*, Workshops in Computing, pages 272–276. Springer-Verlag, Newcastle, Australia, January 10–13, 1989, 1990.

[29] Santosh K. Shrivastava et al. *The Arjuna System Programmer's Guide*. Arjuna Research Group, Computing Laboratory, University of Newcastle upon Tyne, UK, February 1992. Public Release 1.0.

[30] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An efficient, portable persistent store. In *Proceedings of The Fifth International Workshop on Persistent Object Systems (POS-V)*, San Miniato, Italy, September, 1992, 1992.

[31] Pedro Sousa, Manuel Sequeira, André Zúquete, Paulo Ferreira, Cristina Lopes, José Pereira, Paulo Guedes, and José Alves Marques. Distribution and persistence in the IK platform: Overview an d evaluation. *Computing Systems*, 6(4):391–424, Fall 1993.

[32] Bjarne Stroustrup and Dmitry Lenkov. Run-time type identification for C++ (revised). In *USENIX C++ Conference Proceedings*, pages 313–339, Portland, Oregon, August 1992. The USENIX Association.

[33] Jürgen Uhl, Dietmar Theobald, Bernhard Schiefer, Michael Ranft, Walter Zimmer, and Jochen Alt. The object management system of STONE: OBST release 3.3. Technical report, Forschungszentrum Informatik (FZI), D-76131 Karlsruhe, Germany, July 1993.

[34] Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 364–377, Dourdan, France, September 24–25, 1992. IEEE Computer Society Press.