Issues in the Design of a Reflective Library for Checkpointing C++ Objects

Mangesh Kasbekar
The Pennsylvania State University,
University Park, PA 16802
kasbekar@cse.psu.edu

Shalini Yajnik, Reinhard Klemm
Bell Laboratories, Lucent Technologies
600, Mountain Avenue, Murray Hill, NJ 07974
{shalini,klemm}@research.bell-labs.com

Yennun Huang
AT&T Labs-Research,
Florham Park, NJ 07932
yen@research.att.com

Chita R. Das
The Pennsylvania State University,
University Park, PA 16802
das@cse.psu.edu

Abstract

Object Persistence is an important feature of Object-oriented languages. The C++ language specification does not include or discuss any method of providing persistence for C++ objects. Several schemes have been developed for adding persistence to C++. Some of them require persistent objects to be allocated and treated differently than non-persistent objects, while some others require the programmer to provide vital parts of the persistence mechanism. It is desirable to make the persistence feature transparent, but the nature of C++ makes it difficult. This paper discusses in detail the various interesting language issues to be considered for adding persistence to C++ and how they lead to the design of the reflective object-checkpointing library, MemberAnalyzer.

1 Introduction

It is often required to save data of an application beyond the execution of the application that creates it. Various techniques used for this purpose require the data to be saved to be processed or formatted in some fashion before saving it in stable storage. This representation of data on the stable storage (external representation) is typically much different than the representation of the same in a running program (internal representation). In addition, when this data is required to be reused from the storage, it needs to be converted back from external representation to the internal representation. In modern object-oriented programs, data is represented in form of objects, and making the objects persistent is an efficient way of saving the application data into stable storage. In this scheme, the external and the internal representations are very similar to each other, and the overhead of conversion of representations is small.

Popular persistent object systems can be roughly classified into two categories. Type I systems make persistence a property of individual objects. They control object allocation in memory. The program may contain a persistent memory region, and all objects allocated in it are persistent by default. Object databases and persistent stores [13] fall under this category. Type II systems make persistence a property of the type. In this type, each class possesses methods that are used for saving and restoring the state information of the objects of the class. These functions use a set of primitive save/restore operations provided by the system. In order to make an object persistent, the programmer is required to call these functions of the object explicitly. Java serialization and persistence in Microsoft COM belong to this type.

On another front, the area of reflection in programming languages [2] is gaining popularity. Newer languages (such as Java) make use of reflection for many purposes. Reflection is defined as a property of a system to reason about and alter its own behavior. Reflection, typically a runtime action, can also be carried out at compile time, wherein a suitably equipped compiler can analyze the program being compiled and alter it. Compile time reflection can be viewed as a very sophisticated pre-processing. Due to availability of rich data about the program being compiled (called meta-data), much intelligence can be added into program transformations. In this paper, we use compile time reflection for enhancing the Type II technique described above.

The areas of our interest are fault-tolerance, object migration, replication and load balancing. Fault-tolerant systems use replicated processes for increasing their availability. This work is motivated by the need to support primary-backup replication for distributed objects. In such an environment, it is frequently required to save the state of objects into stable storage (cold backup) or transfer it to a backup object in a replicated process running the same application (warm backup). Type II persistent systems are more suited for this purpose than Type I (e.g. use of the machinery of an object database for transferring object state between processes is clearly an overkill.).

In this paper, we analyze in detail the issues involved in developing a suitable Type II persistence system for C++.

1In this paper, we refer to these functions as Checkpoint() and Restore() respectively
and extend the discussion to implementation of a source-code generator for object persistence. This persistence mechanism is implemented as a library called MemberAnalyzer. The goals of this library are to:

1. Provide an optimized object checkpointing mechanism for C++. The library must satisfy all the necessary requirements for persistence (stated briefly in the next section).

2. Provide a sufficient level of user-transparency to the persistence mechanism: In order to make objects persistent, the programmer should not be required to be aware of the underlying mechanism that saves and restores the state of the objects. The only responsibility of the programmer should be the invocation of the mechanism.

3. Be smart: Data structures in programs are often very complex. Writing manually the functions required to save and restore their states can be very cumbersome and a sub-optimal implementation may save redundant copies of data in checkpoint. The mechanism should be able to handle any legal C++ data structure, so that the case of complex data structures can be handled reliably and efficiently by the library.

4. Be flexible: The library should be flexible enough to allow the user to provide functions to completely override the default mechanism, or provide functionality supplementary to the default.

5. Leave the language unaltered: C++ has several features that are difficult and complex to handle. The use of the library should not change the way any language feature is supposed to work, it should not require programmers to change their individual coding style, and it should not impose any restrictions on the use of any language features.

The rest of the paper is organized as follows. Section 2 covers principles of object persistence and related work in brief. The requirements, overview and usage of the library is detailed in section 3. Section 4 provides an in-depth analysis of the issues related to the chosen programming language, C++. Next, sections 5 and 6 contain the implementation details and an example of source code generation by the library. Section 7 remarks on possible future work and concludes the paper.

2 Principles of Object Persistence and Related Work

2.1 Principles of Object Persistence

The principles of object persistence are language-independent and can be tailored to any specific language [7]. In short, they are as follows:

1. Persistence can be either a property of type or only certain instances of the type. If it is a property of a type, it automatically becomes a property of all objects of that class. On the other hand, it can be purely a property of the instances, so that some objects of the type can be persistent, while some others can be non-persistent objects.

2. Lifetime of objects is determined by reachability from other persistent objects. Saving an object into stable storage must also save all objects that are reachable from it. Thus, an object becomes persistent if it is explicitly saved in stable storage, or if it is reachable from another persistent object.

3. During the normal execution of a program with persistent and non-persistent objects, it should be indistinguishable whether the code is operating on a persistent object or a non-persistent object.

These principles are not explicit about whether or not the programmer has any control over when the object state becomes persistent. Here, we assume that objects are made persistent or restored from a persistent state only when the program invokes the appropriate functions.

2.2 Related Work

Persistence has been used very commonly in object systems. In this section, we list some of the systems that make persistence a property of type, as opposed to a property of individual object. Among these, the systems that use C++ are the Arjuna system [9], CMU Dome system [1], Microsoft Foundation Classes (MFC) [10] and Microsoft Component Object Model (COM) [11]. All of the above require the programmer to provide per-class functions that are used for saving and restoring state of objects of the class.

The Python language [12] and Java serialization [15] follow a slightly different manner. In Java, all classes that have implemented the serializable interface can use the default functions provided for serialization of state without the programmer having to supply any code. The Forest project [7] for adding orthogonal persistence to Java takes a similar approach, in which all types possess the persistence property by default. In Python, the pickle module handles saving and restoring of objects (which is termed as pickling and unpickling of objects) in files. Given an arbitrary object and a file, it serializes the object into the file in a python specific format and vice versa.

The difference between the above two approaches arises from the fact that C++ is a compiled language as opposed to both Java and Python, which are interpreted. The runtime environment of a language plays a major role in transparent object persistence. In Java, for example, whenever a class is loaded by the Java Virtual Machine (JVM) at runtime, the complete type information of the class members is also recorded. This information can be accessed at runtime through the Java Reflection API. If an object of the class requests serialization, the default serialization library uses this information to serialize the members of the object. The availability of this information enables a single generic serialization procedure to serialize objects of arbitrary type.

However, there is no runtime environment for compiled languages that maintains information which can be used for our purpose. In the absence of type information of class members at runtime, the persistence schemes for C++ mentioned above cannot provide a default mechanism similar to
that of Java, and have to rely on the programmer to provide class-specific functions that save and restore state.

MemberAnalyzer enhances the persistence mechanism for C++ by providing default functions for user classes. Its compile time component analyzes the user code and generates the `Checkpoint()` and `Restore()` member functions for each class. The type information thus gets indirectly included in the executable code. The penalty for not having type information available at runtime in C++, is thus paid in terms of code size.

The research on fault-tolerance at Bell Laboratories uses process-level checkpointing of `libft` and `libckp` [5]. These libraries are not designed to handle object level checkpointing. `libft` [16], on the other hand, is used for checkpointing data structures. It supports only C programs, uses the approach of source-code generation for structure serialization, and is very close in its approach to MemberAnalyzer.

The only other effort of this kind known to us is part of the work on reflective fault-tolerant architectures at LAAS [6]. This technique also uses reflection for transferring object state across replicated objects. The main stress of this system is on incremental object state checkpointing using Compile-Run-Time (CRT) reflection technique.

3 Overview of the Checkpointing Mechanism

3.1 Using the Library

In order to enable object checkpointing in a program, the source code of the application must be available. The user must use the OpenC++ compiler [3] (occ) to compile the program, with the dynamic link library for the compiler (MemberAnalyzer.dll for Windows-NT or MemberAnalyzer.so for Solaris 2.5) available for use during compilation. When the compilation is completed, each class in the system possesses a `Checkpoint()` and a `Restore()` method. These two generated functions are exactly complementary to each other in their action. The runtime library (libMemberAnalyzerRT.lib for Windows NT or libMemberAnalyzerRT.a for Solaris 2.5) should be linked with the user application.

3.2 How it works

The runtime library contains a global object called `RuntimeSystem` that manages all the data required about the executing program. In order to checkpoint and restore the state of an object pointed to by `thisPtr`, the user is required to make the following calls:

RuntimeSystem.Checkpoint (thisPtr);
RuntimeSystem.Restore (thisPtr);

The object from which state saving or restoring begins is called the root object of the checkpoint. The generated code, at runtime, uses several functions provided by `RuntimeSystem`. Lists called `CheckpointList` and `RecoveryList` are used at the time of checkpoint and restore of objects. These lists contain pointers to objects to be checkpointed and restored respectively.

A call to `Checkpoint()` as shown above inserts the object to be checkpointed into the `CheckpointList`.

The library then invokes the `Checkpoint()` method of this object, which saves the object state into checkpoint file. As a side effect, this method may append additional object pointers to the `CheckpointList`. After the object has finished the execution of its `Checkpoint()` method, the library repeats the same procedure for the next object in the list. The checkpoint ends when all objects in `CheckpointList` are processed.

A call to `Restore()` as shown above performs the same steps as a `Checkpoint()` call, while using `RecoveryList` instead of `CheckpointList`. It is required to instantiate the root object before trying to restore its state from the checkpoint. However, objects reachable from it are not required to be existent. If they do not exist, MemberAnalyzer creates them before restoring their state.

Although the checkpoint and restore procedures are exactly complementary to each other, an additional task is performed by the restore procedure. Since an object pointer may be restored before the object it is pointing to, a correct value cannot be assigned to it until the latter is restored. Such pointers are called `PendingPointers` and they are appended to a `PendingPointerList`. When the `RecoveryList` is processed completely, all the `PendingPointers` are updated.

In section 5, we describe the actual process by which the functions are generated. But before that, section 4 discusses in detail how MemberAnalyzer treats various aspects of C++.

4 Linguistic Issues in Object Checkpointing

In this section, we discuss how the nature of C++ influences the design of MemberAnalyzer. This section is based on the material in [14] and [4]. MemberAnalyzer processes only class definitions. Since the state of an object is nothing but the state of only its data members, it generates code for only the data members of class, and not the function members. The term `processing` used below refers to checkpointing and restoring of an object or a member that takes an appropriate action based on the type of the member.

4.1 Data Types

Here, we describe the action taken by MemberAnalyzer for processing a member of a given data type.

1. Basic Data Types: Types such as `int`, `char`, etc. are atomic and of constant size. Processing members of basic type involves saving their values into the checkpoint and reading them back later. Enumerated types are typically represented by integers by most compilers, and MemberAnalyzer treats them as integers.

2. Composite Types: Members of type `struct` or `class` receive identical treatment. As we know, each class (and also `struct`) has `Checkpoint()` and `Restore()` member functions (generated). These type of members are processed by simply invoking those methods.

3. `union`: Since C and C++ do not use tagged unions, it is not possible to tell the type of the value contained in any union at compile time as well as runtime. In the
absence of this information, a way to process union
members is to save and restore them as an untyped
chunk of memory. This is particularly unsafe, if the
union has a pointer member, since it can potentially
cause correctness problems after it is restored. In or-
doer to avoid untyped data in checkpoints, the current
version of MemberAnalyzer does not handle unions.

4. References: This is a feature of C++, which allows
aliasing of variable names. class and struct types
cannot have members which are references unless they
are initialized in the initializer list in the constructors.
Which means, if during construction of an object O_n,
an alias is being created to another object O_m, then the
latter must have been instantiated before the former.
This has two implications. First, each reference mem-
ber adds an object (O_m) in the set of objects reachable
from O_n, which is unknown at compile time. Sec-
ond, it requires a certain order in creation of objects
at the time of restoring from checkpoint, which cannot
be guaranteed. Therefore, the reference members are
not supported in this version of MemberAnalyzer.

5. Pointers and arrays: The treatment of pointers and ar-
rays is discussed in detail later in this section.

6. const members: Members with the qualifier const
before them in the class definition are constants for the
given class. Compilers do not allocate them as data
members. It is therefore not required to save and re-
store these members.

7. static members: Static members of a class are com-
mon across all instantiations of the class, and hence
can be seen as members of all objects of the same class.
Persistent object systems are divided into two with
regards to their treatment of static variables. Some
schemes include the static members into object check-
points, while others leave them out. MemberAna-
lyzer’s approach is to save the static data mem-
bers of the class into the checkpoint. This is necessary
for transferring state of one process to another. Mul-
tiple objects of the same class may get added to the
same checkpoint, if they are reachable from the start-
ing object. The subsequent objects of the same class
exclude the static data from their checkpoint. It re-
duces the amount of redundant data in the checkpoint.
Since the Checkpoint() and Restore() meth-
ods of any class are exactly complementary to each
other, the Restore() method of only the first object
of any given class restores the static data.

4.2 Inheritance

A class can have one or more parent classes. The pro-
cessing of a class must include processing of its own data
members as well as processing of the data of its parent class.
Given that each class has the generated Checkpoint() and
Restore() functions, this can be easily achieved if the
Checkpoint() method of the child class is made
to call the Checkpoint() method of all its parents.
Since Restore() must be exactly complementary to
Checkpoint(), it must call Restore() of the parent
classes. C++ supports different types of inheritance. Some
issues are required to be handled differently for different
types of inheritance.

private and protected inheritance allows scope
reduction in a class. The generated Checkpoint() and
Restore() functions are public members of class, so
that all child classes have access to these methods of their
parents. Thus, any scope reduction does not restrict a
child class from invoking its parent’s Checkpoint() and
Restore() methods.

If inheritance is virtual, all virtual base classes in the in-
heritance hierarchy have a unique copy in the instantiation.
Virtual bases are reachable from the child through multi-
ple paths. Since each child class calls the Checkpoint() and
Restore() functions of the parent class, those func-
tions of the virtual parents would get called multiple times
while checkpointing a single object. By adding some guard-
ing code in the beginning of the Checkpoint() and
Restore() functions, it is ensured that members of each
virtual parent are saved (and restored) only once.

If any of the classes in the inheritance hierarchy contain
virtual functions or if any of the parent classes is virtual,
then the objects of the class contain hidden members added
by the compiler. They are, virtual function table pointers
(vfptr) and virtual base pointers (vbptr). vfptrs are pointers
to code segment locations, and are fixed addresses for all
objects of a given class. vbptrs are not fixed addresses. Each
vbptr points to the part of the object corresponding to the
virtual base class, for which the vbptr is added. It is not
required to process vfptrs or vbptrs. As will be seen later,
the object restoring procedure automatically takes care of
these hidden members.

4.3 Templates

Although templates are a compile time feature of C++,
it is difficult to process them using compile time reflection.
Stroustrup [14] explains “A C++ parameterized type will
be referred to as class template. A class template specifies
how individual classes can be constructed, much the same
way as a class specifies how an individual object will be
constructed.”

In other words, a class template is not a class, and its
analysis cannot provide complete type information. Al-
though all methods of a template are written using the type
parameter, the same cannot be done for Checkpoint() and
Restore(). Because the treatments given to different
types are so varied, the generation of these functions is
extremely sensitive to the type of members. To avoid
any kind of type detection at runtime, the generation of the
Checkpoint() and Restore() functions is deferred
till complete type information becomes available. This
information can be obtained only at the places where the tem-
plate is instantiated to form a class.

For example, analysis of a class definition of the form
template <class T> class A {...}; is incomple-
te because information about the real type of type-
parameter T is not available. The missing information can
be collected later in the program body, where the template is
instantiated as A<B>, where B is another type. In addition,
there may be multiple instantiations of the same template
with different types substituted in place of T. According to
the definition of class templates above, all these instantia-
tions are separate types, and there must be a pair of these
Arrays can be statically defined or dynamically allocated. They can contain elements of basic type or of composite type. Class members which are arrays of basic or composite type are completely known at compile time in terms of their type and size. Processing them is equivalent to processing multiple members of the given type in sequence. If an array is of basic type, all its elements are copied to / restored from a checkpoint. If it is of type `class` or `struct`, the processing is done in a loop, which calls the `Checkpoint()` and `Restore()` functions of the array elements.

Issues related to dynamically allocated arrays are considered next.

4.5 Pointers

During the execution of a program, variables of type `pointer` can legally contain address of an object, address of an array of objects of its own type, object or array of some other type, the `NULL` location or even completely arbitrary memory locations. Even though the type of the pointer is completely known at compile-time, nothing can be reliably said about what it can potentially point to at runtime. In addition, pointers to type `void` can provide no information about their target at compile-time. The type information is not sufficient to generate code to save and restore the data reachable from a given pointer.

In addition, due to the nature of arrays and their close relation with pointers in C and C++, the principle of persistence by reachability has an important effect on processing of pointers in C++. Given a pointer to an object in an array, it is possible to reach the previous and next objects in the array by simple pointer arithmetic. It makes the whole array reachable from the given pointer member.

Both the above observations imply that it is necessary to perform at runtime some analysis of the data pointed to by any pointer member. The generated code for saving and restoring pointers first carries out some condition checks to figure out the nature of data the pointer is pointing to, and takes appropriate actions to checkpoint that data. The generated `Restore()` code re-creates the same data structure in the restore phase and updates the pointer.

The runtime condition checks use some compiler-dependent low-level information to discover if an object is part of an array. None of this information is saved in any object checkpoint, so as to keep it free from non-portable data. Depending on runtime conditions, the generated code takes decision to either include the pointer in the checkpoint or it may exclude if available information is insufficient. The decision to include the pointer in a checkpoint is taken, only if at runtime the pointer points to either a single object, or if it points to the first element of an array (single or multi-dimensional).

The pointer support for heap pointers is discussed next for various cases that arise out of the type (pointer to basic type or class type) and level of indirection.

4.5.1 Pointer to basic type with single level of indirection

The data pointed to can be treated as a one-dimensional array of the basic type. The information required to checkpoint this array is the beginning and size of the array. This information can be obtained from the malloc library only if the pointer points to the first element of the array. In the absence of this information, the pointer is treated no differently than a pointer containing an arbitrary value. The code generated to checkpoint the pointer first makes a query to the malloc library by invoking `msize()`, that returns size of the memory block. If successful, the content of the array is saved into the checkpoint. Otherwise, an invalid entry is written into the checkpoint.

4.5.2 Pointer to class type with single level of indirection

Whether or not an object pointer actually points to a valid object (as opposed to an arbitrary location) can be easily tested with help from the `RuntimeSystem`. If the pointer is not found in the lists maintained by the `RuntimeSystem`, it is treated as a garbage value. Although the basic procedure to handle this type of pointer remains the same as previous case, an additional complication is added because of the type rules of C++ about equivalence of pointers to a given type and its subtype. If class `PersistentClass` is a parent of class `A`, then `PersistentClass*` as well as `A*` are legal pointer types to any object of type `A`. Moreover, two pointers pointing to the same object, but of different types are offset from each other by a compiler- and class-specific value called `thunk`. Thus, even if a pointer points to the first element of an array, it may not necessarily be the pointer to the block of memory allocated for the array. To handle this successfully, the pointer value is adjusted using `thunk`\(^2\), an `msize()` query is made to discover the size of the memory block allocated for the array. Once this information is obtained, saving each object in the checkpoint is trivial and is done by simply appending the elements of array to the `CheckpointList`. On the other hand, if the query fails to return a size for the block, (knowing the pointer is a valid object pointer), an entry is appended to the `CheckpointList`, optimistically hoping that the other parts of the array will also get added to the list due to processing of other members.

\(^2\) And also the object array header mentioned in 4.7.
4.5.3 Pointers with multiple levels of indirection

Any such pointer is seen as a start of a multi-dimensional data structure as shown in figure 1(b). This structure contains levels of pointers pointing to other pointers with one less level of indirection, until the last level, which is made up of one dimensional arrays of data objects. The treatment of the last level of this structure is the same as in the two cases described above (appropriate to their type). The treatment of all the middle levels of pointers is identical, and involves saving appropriate descriptors in the checkpoint that carry information about the size and level of indirection. Multiple pointers may point to the same location in the complex data structure. The solution is optimized to avoid duplicates.

We do not describe any of the recovery procedure in detail, because in most cases it is exactly complementary to the checkpointing procedure.

C++ is extremely flexible in creation and placement of objects. Objects can be dynamically allocated on the heap, as well as instantiated on the stack like a local variable and in the data segment like global variables. All the three ways of instantiating objects are well-used in practice. If the pointer is not a heap pointer, the current version of MemberAnalyzer does not handle it. These pointers could point into the stack or the data segment, and have the same issues as described earlier. But, no additional information can be obtained due to the unavailability of suitable memory functions from Windows NT or Solaris and hence the structure of the pointed data cannot be determined at runtime.

The pointer support of MemberAnalyzer is the most complex feature. On one hand, the design attempts to keep the flexibility and power of pointers intact and not prevent programmers from developing any complex data structure. On the other hand, absence of any runtime mechanisms for discovery of the true nature of data pointed to by the pointer prevents this version of MemberAnalyzer from supporting completely arbitrary use of pointers. However, note that the programmer is not prevented from using pointers freely. All the restrictions above apply only while saving the state of an object.

4.6 Type Safety

RuntimeSystem maintains a list of all objects in the program. At checkpoint and rollback time, it invokes their Checkpoint{} and Restore{} functions. It creates new objects at the time of restoring them. Objects of several different types exist in any program, and their handling by the library must be type safe. If the objects are not addressed by a pointer of their own type (or a parent class pointer), its virtual functions cannot be resolved. This prevents the runtime library from keeping track of the objects in form of void pointers.

Pointers to parent class are legal pointers to child class objects, and therefore compile time type information could be misleading. Such situations are created by the type system of C++, and can be satisfactorily handled by using the type system itself. A class called PersistentClass is made a parent class of every class in the system. It allows the library to view every object as a PersistentClass object. The static and dynamic typecast, virtual inheritance and virtual method features of C++ can be used to handle the object and invoke appropriate methods. In the absence of this feature, the library would have to maintain the handle to each object in form of a void pointer, which is unsafe.

RuntimeSystem instantiates objects during a restore phase. Viewing each object as an object of PersistentClass is inadequate at this time. In order to keep track of the type of the object, MemberAnalyzer assigns a unique type-id to each class during compilation. Checkpoint of each object contains the type-id, which is utilized for the above purpose.

4.7 Constructors

Constructors are member functions of classes that are executed when an object is created. Typically, they are used for initialization of the class members. However, there is no restriction on what can be contained in a constructor. Execution of a constructor may have an effect on entities outside the object being created. If the constructor is executed at the time of recreating the object during recovery, such an external effect can potentially violate correctness of resulting state. In order to avoid this, user-defined constructors should not be used when re-instantiating objects in recovery phase. MemberAnalyzer adds a dummy overloaded constructor to each class. The re-instantiation procedure uses this constructor at the time of recovery and bypasses execution of any of the user defined constructors.

Although re-instantiations of single objects can be made using the dummy constructors, object-arrays cannot be re-instantiated using the dummy constructors. C++ uses only the default constructor for initializing objects in an array. The difficulty in checkpointing is compounded by the unavailability of the array size at runtime. We have used an undocumented feature of the compilers for this purpose. The C++ compilers (cl on Windows NT and g++ on Solaris) follow the object layout pattern suggested in [4] (as shown in figure 1(a)), which suggests a header for each object array, that contains the size of the array.

When checkpointing dynamically allocated object arrays, the number of array elements is saved in the checkpoint before checkpointing any of the members of the array. At the time of restoring the array, the RuntimeSystem first allocates a memory chunk for the array based on the number found in the checkpoint, updates the header and then instantiates every object in the array in-place by using the placement new operator and the dummy constructor. The arrays thus created are fully compatible with the delete[] operator. All objects of the array get restored to their state in the checkpoint in the due course of the recovery sequence. The re-instantiation procedure is generated...

Figure 1. Structure of object arrays in C++
for each class, and is called ManufactureObject\_n(), where \( n \) is the type-id of the class.

MemberAnalyzer never destroys any objects, and therefore no special treatment is needed for the destructors.

### 4.8 Memory Management

As explained before, the generated code queries the malloc library to determine the size of dynamically allocated memory chunks. The malloc library of Windows NT provides a function called _msize(), which is used for this purpose, but typical Unix systems do not have it. Our prototype on Solaris includes a modified BSD-malloc library that contains this function.

### 4.9 Consistency of Compilation and Linking

Conventionally, classes are defined in header files and these header files are included in several other files containing program code. Each program file is compiled separately from each other to produce object files that are finally linked together. Class definitions in all the object files are consistent with each other, because of the original inclusion of the same header file in all program files.

The compile time component of MemberAnalyzer alters class definitions. In order to maintain consistency, everything must be compiled by MemberAnalyzer, so that the alterations are consistent everywhere. This is seldom possible because libraries are available only in binary form. It essentially means that library classes cannot be made persistent. Therefore, they must be treated differently or excluded from all alterations.

### 4.10 Exclusion from Code Generation

Often, it is not desirable to include all members of an object in its saved state. Such a case may arise when the data contained by some members is temporary in nature, or it loses its significance after transfer to another object. Including these members in a checkpoint only increases the checkpoint size unnecessarily. In order to reduce the size, MemberAnalyzer provides a simple solution to leave them unprocessed. During the analysis of a class definition, each member is processed by default, unless it is prepended by a keyword MA\_Transient (e.g. MA\_Transient int x ;). Members declared as MA\_Transient by the programmer are considered as insignificant in the state information, and their processing is skipped.

Earlier, we described why library classes, whose code is available only in binary form, should be treated differently. Even if the library classes are available in source form, it may be necessary to exclude some from processing. In a test of MemberAnalyzer with a large program that includes some multimedia header files, in excess of 600 class definitions were processed and code was generated for them, while the program used only one class from the header file. The code generated for unused classes is dead code and only gives rise to code-bloat. It is therefore necessary to provide another escape for excluding class definitions from being processed. The system header files cannot be altered to add MemberAnalyzer-specific keywords in them, and therefore, no keyword based solution as above can be applied for class-exclusion. For each class definition encountered, the default action of MemberAnalyzer is to exclude it from processing. Each class that is required to be processed must be specified with another keyword MA\_Persistent (e.g. MA\_Persistent class A {{...}}). In addition, the user may explicitly specify the names of the classes to be processed in a supplemental file processClasses.db. All classes specified by these two means are processed by MemberAnalyzer.

### 4.11 Special Treatments

So far, we assumed that saving and restoring values of the data members is sufficient for correctly saving and restoring the state of the object. This may not be true, when external state is taken into account. For example, an integer member of a class may in reality be a file descriptor corresponding to an open network socket. Saving and restoring the value of the socket descriptor is certainly useless. In this case, the member needs special treatment. MemberAnalyzer, which generates code based on type-information, cannot provide any special treatments. In such cases, the programmer must write this special code. MemberAnalyzer provides two ways for the programmer to provide this.

The programmer can choose to completely write the Checkpoint() and Restore() functions. MemberAnalyzer can detect the presence of these functions and skip code generation. Often, very few members of the class require special treatment. In order to save the programmer the burden of writing the complete functions, the facility of supplementary functions is provided. If a class contains member functions called CheckpointSupplement() and RestoreSupplement(), MemberAnalyzer generates the checkpointing and restoring code as usual, and in the end of those functions adds a call to the supplementary functions. In case of the example above, the programmer may mark the integer member corresponding to the socket as MA\_Transient and provide the supplementary functions that save and restore the state of the network socket in the CheckpointSupplement() and RestoreSupplement() functions.

### 5 Implementation Details

The library is based on OpenC++ 2.5 [3], which is a source-to-source translator for C++. This section describes how MemberAnalyzer uses the functionality. Due to lack of space, the details of source-to-source transformations are omitted. They can be found in [3].

#### 5.1 Design of MemberAnalyzer Compile-time Component

The meta-level program (i.e. the compile-time component of MemberAnalyzer) is derived from the OpenC++ meta-objects and changes the default functionality in order to carry out the program transformation as desired. Of special interest here, are the TypeInfo, Class and Metaobject. The TypeInfo metaobjects represent types that appear in the program, the Class metaobjects represent the class definitions and also control the source-to-source transformation, and the Member metaobjects rep-
resent class members. The functions of the Meta-Object Protocol (MOP) that deal with the translation of class definition and analysis of Member metaobjects are of interest to us here, and are listed below.

1. TranslateClass(): This function translates class definitions. The default implementation is provided by class Class and it does nothing. MemberAnalyzer provides this function, which is responsible for the type analysis of class members and generation of code.

2. Signature(): Returns a TypeInfo metaobject corresponding to the member being analyzed. This metaobject provides information about the type of the member.

3. Functions such as IsBuiltInType(), IsConst(), IsArray() are used on the TypeInfo metaobject for determining the exact type of the member. After the type of the member is determined, appropriate code can be generated for it.

MemberAnalyzer is composed of three meta-classes that carry out the source-to-source translations. The class MemberAnalyzer is the main metaclass. It inherits from Class and provides its own implementation for TranslateClass(). This function analyzes each member of the class definition being translated, determines the type and uses the functionality provided by CheckpointTreeGenerator to put together the parse trees for Checkpoint() and Restore() functions. After all members of the class are processed, the generated functions are appended to the current file.

The class CheckpointTreeGenerator knows the correlation between a type and the method to check point and restore any member of the type. It provides several functions that generate parse trees for checking and restoring members of types such as basic types, class type, pointer type etc. These functions are invoked by translateClass() appropriately after deducing the type of the member being processed. CheckpointTreeGenerator provides functions for handling members of class type and also pointer and array types. For more primitive data types, it uses the functionality provided by its base class, TypeSpecific. This class also contains functions for initializing the parse trees of the Checkpoint() and Restore() functions. The initial code for all classes is the same and it checks certain conditions to ensure that the same object is not being checkpointed twice in the same checkpoint and static variables are not being saved multiple times in the same checkpoint.

The class TypeSpecific handles members of basic types. It contains two functions for generating parse trees for checkpointing and restoring for each basic type supported by C++. This class also handles arrays of basic type.

5.2 Design of MemberAnalyzer Runtime Component

The runtime component of MemberAnalyzer (i.e. the object RuntimeSystem) is fairly simple. During execution of a program, the runtime component is invoked only when an object is created, destroyed, checkpointed or restored. The RuntimeSystem maintains a list that keeps information about objects existing at any given time in the execution of the program. The constructor and destructor of PersistentClass contain code that enter and delete an entry in the list corresponding to the object being created or destroyed. Since PersistentClass is a parent of all classes of interest in the system, each object creation and destruction gives rise to a call to the runtime component.

The other two lists in RuntimeSystem, CheckpointList and RecoveryList, are used only during the checkpoint and restore time. Any object appended to these lists gets automatically scheduled for a checkpoint or a restore operation. Both these lists perform duplicate entry detection and thus avoid any object from getting scheduled multiple times in a single checkpoint or restore operation.

6 An Example of Source Code Generation

In this section, we present a small example of the source code transformations done by MemberAnalyzer. The program shown in Figure 2 consists of four classes - DataHolder, Foo, Bar and NetworkedFooBar. NetworkedFooBar is inherited from Foo and Bar, and contains a pointer to class DataHolder.

After the translation, all classes are amended by adding members to the class definitions and the generated code is appended to the file. We show the changes and generated code only for NetworkedFooBar.

```cpp
class DataHolder { ... };
class Foo { ... };
class Bar { ... };
class NetworkedFooBar :
  public Foo, public Bar {
    public :
      int sockid ;
      int "a ;
      DataHolder d ;
    void NetworkAction (char*) {
    }

  main () {
    NetworkedFooBar *x ;
    x = new NetworkedFooBar ;
    x->NetworkAction ("1") ;
    RuntimeSystem.Checkpoint (x) ;
    x->NetworkAction ("2") ;
    RuntimeSystem.Restore ( x ) ;
    return 0 ;
  }
}
```

**Figure 2. Original Source Code**

The code sample in Figure 3 shows the amended class definition. The following changes can be observed.

Along with the original base classes, PersistentClass is now found in the base class list. A type-id is assigned to the class, and is a constant for the class expressed as an enum. Two new constructors have been added, a dummy constructor for re-creating
the objects during restore phase, and a default constructor. The class definition now contains the prototypes of Checkpoint() and Restore() functions.

```cpp
class NetworkedFooBar : public Foo, public Bar, virtual public PersistentClass {
public:
  int sockid;
  int * a;
  DataHolder d;
  void NetworkAction ( char * );

public:
  enum {GENERATED_MEMBER_TypeId = 4};
private:
  friend PersistentClass*
  ManufactureObject( unsigned int);
  friend PersistentClass*
  ManufactureObjectArray( unsigned int, int);

public:
  NetworkedFooBar (DummyClass *GENERATED_ARG_0, unsigned int oid);
  NetworkedFooBar (unsigned int, int);}

private:

public:
  virtual void Checkpoint (int, int=0);
  virtual void Restore (int);

private:
  NetworkedFooBar();
};
```

**Figure 3. Amended Class Definition**

The code sample of Figure 4 shows the generated Checkpoint() function. The function consists of the following: The first few lines contain code to avoid multiple copies of the same class from being saved into the same checkpoint and handle treatment of static members. The members `sockid`, `a`, and `d` are saved in the checkpoint. After all the members are saved, control is transferred to the base classes by invoking their Checkpoint() functions. An exactly complementary function Restore() is shown in Figure 5.

In a test of MemberAnalyzer with an application approximately 8K lines in size having 21 user classes, it was seen that generation of code increased the size of the executable from 500 Kbytes to approximately 598 kilobytes.

## 7 Summary and Concluding Remarks

This paper presented the issues in the design of MemberAnalyzer, a reflective library for transparent object check-pointing of C++ objects. The library uses compile time reflection and the technique of source code generation for adding the default mechanism of saving and restoring object state to C++, similar to the default object serialization mechanism of Java.

The design of the library attempts to fulfill all goals listed in the beginning of this paper, and the implementation succeeds to a large extent in achieving them. An object persistence mechanism has been developed for C++. The programmer is not required to be aware of the underlying mechanism in order to use it. The design attempts to provide support for arbitrarily complex data structures, and allows for using most features of C++. It also allows for overriding the default functionality. The library has some limitations in handling reference and union members in classes. Although the use of pointers is not restricted, the checkpointing procedure is very sensitive to the data pointed to by the pointers. If sufficient information cannot be gathered at run-time, it may result in exclusion of the pointer from the checkpoint.

Being a source-processing library, MemberAnalyzer is not much dependent on the platform, and has been used
void NetworkedFooBar::Restore (
    int checkpoint_number ){
    int GENERATED_VARIABLE_chunksize = 0 ;
    int GENERATED_VARIABLE_restoreStatic = 0 ;
    swizzle GENERATED_VARIABLE_us = 0,0 ;
    static int local_checkpoint_number = -1 ;
    static NetworkedFooBar* last_object = (NetworkedFooBar*)0 ;
    if ( last_object == this &&
        local_checkpoint_number == checkpoint_number) return ;
    last_object = this ;
    if (local_checkpoint_number <
        global_checkpoint_number) GENERATED_VARIABLE_restoreStatic = 1 ;
    local_checkpoint_number = global_checkpoint_number;
    CheckpointRead (& sockid ,4);
    if ( a ) GENERATED_VARIABLE_chunksize = msize(a);
    if ( GENERATED_VARIABLE_chunksize > 0 ) free(a );
    CheckpointRead *(( GENERATED_VARIABLE_chunksize,sizeof(int) ));
    else CheckpointRead (a,sizeof(int*) );
    d .recover(checkpoint_number ) ;
    Foo::recover(global_checkpoint_number);
    Bar::recover(global_checkpoint_number);
    }

Figure 5. Generated Restore() code

on Windows-NT and Solaris 2.5. It has been tested with medium sized applications (less than 10K lines of code). An altered version was created for use with research on checkpointing of multithreaded programs[8] at the Pennsylvania State University. The source code generator can also be used in visual environments that perform significant amount of code generation themselves.

Future work can take many directions. Machine independent object checkpoints are very useful. Libraries such as XDR and RPC-NDR can be used to make the checkpoints machine independent. More investigation is needed in making the object checkpoints independent of the language. Such a facility would be both interesting and useful.

Acknowledgements

Thanks to Shigeru Chiba for answering numerous questions about OpenC++.

References


