The Silicon Graphics Delta C++ Compiler

Andrew J. Palay
Michey N. Mehta
Shankar C. Unni

White Paper (Preliminary)

1.0 Introduction

For object-oriented systems to be ultimately successful, developers must be able to make compatible modifications to class definitions and implementations without requiring consumers of that class to recompile their code. The separation of interface and implementation found in pure object-oriented systems makes it possible to do just that. Unfortunately, when confronted by the need to provide reasonable performance, most object-oriented systems have broken down the barrier between interface and implementation. By exposing class implementation details to consumers, we are left with systems that cannot support modifications to shared libraries without obsoleting consumers of these libraries. Furthermore, costly recompilations are needed whenever a class definition is changed. This is especially true of today's C++ environments.

C++ systems expose the implementation of classes to consumers of a class by resolving class layout and function binding at compile time. The effect of doing compile-time class layout is magnified by exposing the data-members of a class, either directly or through the use of inline member functions which access data members. While this violates the tenets of pure object-oriented programming practice, accessing data members directly is essential in achieving satisfactory performance in many applications.

Delta C++ is a new C++ compilation system that provides a better separation of interface and implementation without imposing a performance penalty, and without limiting the use of the C++ language. Delta C++ provides this separation by resolving class layout and function binding at link-time. The compiler generates code similar to non-Delta compilers but lets the linker do the final relocations in the code to reflect the actual class implementation. The linker performs class layout and relocations by utilizing a new set of symbol tables generated by the compiler.

With Delta C++, a developer can make upwardly compatible changes to a class definition without affecting consumers of that class. A new interface and implementation of a class can be introduced and code compiled using the old interface need only be relinked, either statically or dynamically. This allows us to support upward compatible changes to shared libraries because any applications which use such shared libraries are automatically relinked by the runtime linker. The development environment is also improved by reducing the number of recompilations that need to be performed when a class interface changes; only the implementation of the class needs to be recompiled, and consumers need not be recompiled.

2.0 Separation of Interface and Implementation

Shared libraries have become an essential requirement for the development of large software systems. There are various reasons for this:

1. Shared libraries can reduce the memory requirements for running a suite of applications which use the same library, since only one copy of the library is needed in memory.
2. Shared libraries can reduce the disk requirements of a system, by having only one disk copy of code shared by multiple applications.
3. A shared library provider can make quality and performance improvements in a library and replace an earlier version of the library; this automaticallyprovides benefits to existing clients of the shared library without the need to recompile or relink.
4. A shared library provider can add new functionality to a library; existing clients will continue to work, and new clients can take advantage of the added functionality.

In order to take advantage of shared libraries, it must be possible to release a new version of a shared library, where the developer has modified the implementation (and perhaps the interface) in a compatible fashion, and have applications that use that shared library continue to work. To achieve this, it is essential that there be an absolute separation between interface and implementation. If that separation does not exist then a change to the implementation provided in a shared library will require existing applications to be recompiled.

2.1 Shared Library Renaming

One common solution to the problem of shared library evolution is to bind applications against specific instances of a shared library. When a developer releases a new version of a shared library that is not link-time compatible with the previous version, the library gets released with a new name; the old version of the shared library will continue to exist. Applications that used the old version will continue to work by linking against the old version, while new applications can link against the new version. This solution does allow new clients to take advantage of added library functionality; however, we do not get the memory and disk space savings that shared libraries should deliver, and existing clients cannot take take advantage of any quality and performance improvements in the library.

If application A requires version 1 of the shared library and application B needs version 2 of the shared library, then when both applications are running, there will be two copies of the shared library in memory at the same time. Similarly, because both libraries are needed, they must both be resident on the machine's disk. In fact, using this scheme users must install all versions of a shared library ever released, since they will not know which versions are needed to run the applications they are interested in.

From a library providers perspective, this mechanism poses many difficulties. Not only does a library provider have to continue to ship out all versions of a library, but they may be left with the near impossible task of fixing problems in older versions of the shared library. Assume in the previous example that users of application A find a problem in version 1 of the shared library that is fixed in version 2 of the library. Since the two versions are not link-compatible the library provider cannot force existing applications to use version 2 of the library. Either the application provider must release a new version of their application (a cost they may not want to assume, since the problem is not in their code) or the library provider must provide an new version of the library that is link-compatible with version 1. This may or may not be possible to do, but assuming that it is, going back and fixing older versions of a system is often extremely expensive and difficult.

Another drawback with this mechansim for supporting shared libraries is with applications that are designed to dynamically load shared libraries, specifically ones that are provided by other vendors. Many systems are being built in this fashion, where a basic object-oriented framework is provided ,and multiple vendors provide extension components that are loaded on demand. For example, IRIS Inventor provides a 3D framework, that allows other developers the ability to extend the framework by creating and dynamically loading shared libraries that contain new 3D elements. For these systems to work, the framework provider must be able to release a new versions of the framework and have existing extension libraries continue to work. This can only be done if the new framework library is link-compatible with the old library.

2.2 Reducing Build Times

Separation of interface and implementation also provides benefits during software development. With a clean separation, code that uses an interface need not be recompiled when the underlying implementation has changed or when the interface changes in a compatible fashion. With a clean separation, the cost of introducing a modified interface/implementation should be limited to the cost of recompiling the new implementation and relinking. Without a clean separation, it is usually necessary to recompile all code that uses the interface, and this means unacceptably long build times. The effect of not providing a clean separation is that developers stop modifying code in heavily used portions of the system. Developers are thus discouraged from making changes in the kernel of a system, leading to ad-hoc workarounds in other parts of the system. This leads to systems that are difficult to understand and maintain.

3.0 The Problem with Existing C++ Compilation Systems

The fundamental problem with today's C++ systems is that they resolve class interfaces and implementation at compile time and these implementation decisions are embedded in client code. This is necessary in order to achieve acceptable performance. Even C++ code written in a pure object-oriented fashion (i.e. classes that only provide virtual member functions and have only private data members) compilers still generate code, where simple changes to even the private interface of a class will result in the generation of link-incompatible code. For example, consider the following class declarations:

Code Example 1.

```
class alpha {
   public:
      virtual long f();
      virtual long g();
   private:
      long a;
      long b;
};

class beta : public alpha {
   public:
      virtual long i();
      virtual long j();
      virtual long k();
   private:
      long c;
      long d;
};
```

and the following code:

Code Example 2.

```
long beta::i()
{
   return c + j();
}
```

Under the layout scheme used by most C++ compilers the `return` statement would access the member `c` by accessing the information 12 bytes off of the `this` pointer and would access the function `j` by looking for a pointer to the virtual table that is 8 bytes from the `this` pointer and then looking in the fourth slot of the virtual table for the function pointer. On a MIPS-based processor the generated code for `beta::i` would look like:

Code Example 3.

```
lw    t6,40(sp)      # load this pointer
nop
lw    s0,8(t6)       # load vtable pointer
nop
lh    t7,32(s0)      # load d field for function
lw    t9,36(s0)      # load f field for function
addu  a0,t6,t7       # add d offset to this pointer
jalr  ra,t9          # call virtual member function
nop
lw    t8,40(sp)      # load this pointer
lw    t0,12(t8)      # get value of c field
nop
addu  v0,v0,t0       # Add the two values together
```

Now the provider of `alpha` wants to change the implementation by adding another data member so that `alpha` now looks like:

Code Example 4.

```
class alpha {
   public:
      virtual long f();
      virtual long g();
   private:
      long a;
      long b;
      long x;
};
```

With this change in `alpha`'s definition the layout for `beta` has also changed so that the data member `c` is now located 16 bytes from the `this` pointer and the pointer to the virtual table is now 12 bytes from the `this` pointer. Even with this simple change (the interface to `alpha` has not changed), a call to `beta::j` cannot work without being recompiled.

A possible solution to this problem is to add one level of indirection to the layout of objects and use indirection when accessing data members. Using this scheme a pointer to an instance of a `beta` would be pointing to a three word structure. The first being a pointer to the virtual table, the second being a pointer to the data members provided by `alpha` and the third being a pointer to the data members provided by `beta`. The cost of this layout scheme is an added memory reference whenever accessing data members of a class and an extra word per depth of the class hierarchy for each class instance. An instance of a class that derives from `beta` would have a three word overhead.

Consider a simple change to the interface of `alpha` - adding a new virtual member function resulting in the following definition:

Code Example 5.

```
class alpha {
   public:
      virtual long f();
      virtual long g();
      virtual long m();
   private:
      long a;
      long b;
      long x;
};
```

Now the slot for the member function `j` in the virtual table for `beta` moves from four to five. This would require that we recompile the call to `beta::j`. Again, we could solve this problem by adding a second level of indirection to the virtual table. Thus the virtual table pointer associated with an instance of a beta will actually point to a two word table. The first word would point to a table for the virtual functions introduced by `alpha` and the second would point to the virtual member functions introduced by `beta`. This would introduce one extra memory reference whenever we access a virtual member function.

A similar solution would be to introduce a second virtual table pointeri n each instance of a `beta`. In this case a pointer to an instance of a `beta` would be pointing to a four word structure, the virtual table pointer for the virtual member functions provided by `alpha`, the pointer to `alpha`'s data members, the virtual table pointer for the virtual member functions provided by `beta`, and the pointer to `beta`'s data members. This would remove the extra memory reference when accessing a virtual member function but increases the memory overhead per instance to twice the depth of the class in a class hierarchy.

These schemes start to fail as we start to consider more interesting changes to class interfaces that should be possible without forcing recompilation of code that uses the interface. Another simple change to a pair of class interfaces is to promote a member from derived class to a base class, for example moving the virtual member function `k` from `beta` to `alpha`. From a consumer of either `alpha` or `beta`, neither class interface has changed in an incompatible fashion. Consumers of `alpha` see an expanded interface, and consumers of `beta` see exactly the same interface. What changes is the layout of the virtual function table for `alpha` and `beta` in the normal layout mechanism used by most C++ compilers. In most C++ systems the slot for the virtual function `beta::j` changes to six.

Neither of the above proposed schemes for handling simple changes to virtual member functions can handle this type of change. In either of these schemes the class in which a member is defined is fixed at compile time. Changing this defining class invalidates the generated code.

The ability to promote members must be supported for both data members and member functions. C++ allows data members to be part of a the non-private interface of a class, either explicitly, by declaring them public or protected, or implicitly, by using them in a non-private inline member function.

Changing the overriding behavior for a class member is another type of change that requires the recompilation of consumers of a class. Consider the class definition:

Code Example 6.

```
class gamma : public beta {
   public:
      virtual long p();
      virtual long q();
};
```

If we change the class definition of `beta` so that it overrides the member function `f`, the virtual table associated `gamma` must now refer to `beta::f` as opposed to `alpha::f`. This would not happen unless the code that contains the definition of `gamma`'s virtual table is recompiled.

Similarly if the original definition of the class `alpha` had included a non-virtual member function `nv()` and there was a call to that function from a pointer to a `gamma` (`pg->nv()`) then changing `beta` to override the definition of `nv` will require recompiling the function call in order to get the proper behavior. In the original case a direct reference would be made to the function `alpha::nv` while in the latter a reference to `beta::nv` would be needed.

4.0 Delta C++

The Silicon Graphics Delta C++ compiler solves many of the problems that exist in current C++ compilers. By delaying the actual binding of class definitions until link-time, developers can make substantive changes to class definitions and class hierarchies, while only recompiling the files that implement classes that have been directly changed. For example, in moving from the definition of `alpha` given in Code Example 1 to the definition of `alpha` given in Code Example 4, only the files that implement `alpha` would have to recompiled. The files that implement `beta`, or are clients of `alpha` or `beta` need not be recompiled. Relinking is all that is necessary.

Delta C++ is designed to strike a reasonable balance between the ability to change class definitions and the need to generate efficient code. We are not willing to sacrifice speed in order to allow developers complete freedom to make arbitrary changes to class definitions. One example of this is the handling of inline member functions: while it would be nice to allow developers the freedom to change the definition of an inline member function, to do so would require such functions to be called out-of-line, and this would negate the reason for using inlines in the first place. Developers often use inline member functions as accessor functions for private data members. This allows them to write code that has only a functional interface but compiles down to direct references to an object's data members. In order to allow developers to change the definition of an inline member function we would have to ignore the inline attribute and generate out-of-line copies.

In developing Delta C++ we chose to support class changes that could be done by applying simple link-time relocations without changing the generated code sequences. Developers using Delta C++ can make the following types of changes to class definitions without recompiling consumers of the class that has been changed:

- Member Extension (the addition of new members to a class)
- Member Promotion (promoting a member from a derived class to a base class)
- Override Changing (overriding a member from a base class)
- Class Extension (adding new base classes to a class)
- Member Reordering (reordering the members if a class)

4.1 Member Extension

This is the most basic type of change that is allowed by Delta C++. Member extension is the addition of any new member to a class definition. For example, moving from the definition of alpha in Code Example 1 to Code Example 5 is an example of member extension where both a data member and a virtual member function have been added. From the perspective of a consumer of `alpha` any code that previously referenced a member of `alpha` will still be valid. Without the ability to extend a class definition (which includes just adding private data members), even fixing minor bugs in C++ libraries becomes impossible without breaking link-time compatibility.

There is an important limitation on member extension. Adding an overloaded member function that has the same number of parameters as another of the overloaded member functions is not allowed. For example going from:

Code Example 7.

```
class lambda {
   public:
      virtual long f(int);
};
```

to:

Code Example 8.

```
class lambda {
   public:
      virtual long f(int);
      virtual long f(float);
};
```

is not considered to be a compatible change. The reason behind this is the following code sequence:

Code Example 9.

```
lambda *pl;

pl->f(5.0);
```

when compiled against the definition of lambda given in Code Example 7 will call the function `lambda::f(int)`, coercing the 5.0 to the integer 5, while with the definition of `lambda` given in Code Example 8 it should call the function `lambda::f(float)`. The signature of every function being called is determined at compile time, and classes cannot not be changed in a way such that an existing function call would resolve to a function with a different signature. This limitation also implies that the addition of new conversion operators will usually be an incompatible change.

Another limitation on member extension is that it is an incompatible change to modify a class such that a global entity gets hidden by a newly added member. Consider the classes `base` and `derived` in Code Example 10:

Code Example 10.

```
int var = 0;
class base {
   public:
      ...
};
class derived : public base {
   void func();
};
...
void derived::func(){
   var = 5;
}
```

The reference to `var` in `derived::func` will bind to the global variable `var` in the current release. Consider what would happen if class `base` gets changed in a future release to the version listed in Code Example 11:

Code Example 11.

```
class base {
   public:
      int var; //This hides the global var
      ...
};
```

The reference to `var` in `derived::func` should now bind to the member `var` in class `base`; if the compiler had to generate code that could allow a reference to a global entity to bind to a class member in a subsequent release, the generated code would be inneficient. Therefore, this is not considered a compatible change, and class `derived` will need to be recompiled.

4.2 Member Promotion

Member promotion is moving of a member from a derived class to a base class. For example , given the definition of `alpha` and `beta` in Code Example 1, the following is an example of member promotion:

Code Example 12.

```
class alpha {
   public:
      virtual long f();
      virtual long g();
      virtual long j();
   private:
      long a;
      long b;
};

class beta : public alpha {
   public:
         virtual long i();
         virtual long k();
   private:
      long c;
      long d;
};
```

From a consumer of the class `Beta`, its interface has not changed. Any references to the method `beta::j` are still valid. The offset in the vtable for this method will be different as will the function address stored in the vtable, but these are handled by Delta C++ linker.

There are several limitations in Delta C++ on the ability to promote members. We do not allow the promotion of non-static members into a virtual base class. In order to do this we would have to generate code that is ready to accept that type of promotion. When accessing a member of a virtual base class code must be generated to follow the virtual base class pointer prior to accessing that member. For example given the following class definitions:

Code Example 13.

```
class alpha {
   public:
      long f();
   private:
      long x;
};

class beta {
   public:
      long g();
   private:
      long y;
};

class delta: public alpha, public virtual beta {
   public:
      long h();
   private:
      long z;
};
```

a call to pd->g() results in the following MIPS assembly code:

Code Example 14.

```
lw    a0,32(sp)      # load this pointer
lw    t9,0(gp)       # load function address
lw    a0,8(a0)       # indirect through vbase ptr
jalr  ra,t9          # call beta::g
```

while a call to pd->h() results in the following MIPS assembly code:

Code Example 15.

```
lw    t9,32(sp)      # load function address
lw    a0,32(sp)      # load this pointer
jalr  ra,t9          # call beta::h
```

If we wanted to promote the member function `h` from `delta` to `beta` then we would have to have generated the code sequence given in Code Example 14 and have a virtual base class pointer in delta that points to itself. The performance penalty for doing this would be an additional pointer in every instance of a class, and an additional instruction for data member and non-virtual function references, and an additional memory reference during execution. Since virtual base classes are rarely used, we felt that this was an unacceptable performance penalty to impose on the vast majority of C++ developers.

By a similar argument we also chose not to allow the promotion of non-static member functions into a non-leftmost base class. To allow this we would always have to prepare for this case. This would have added an instruction that added an offset to the `this` pointer when making a non-virtual member function call. In the single inheritance case that instruction would just add a 0 to the this pointer. Since the cost of allowing this type of change is very small, we might discover after more people have used Delta C++, that we were overly aggressive in choosing performance over features.

4.3 Override Changing

Supporting override changing makes it possible for a developer to override a member that was originally found in a base class. Typically only member functions are ever overridden, but Delta C++ also allows the overriding of data members. By allowing changes to the override behavior of a class, a developer can, in a subsequent release, specialize the actions of the derived class. For override changing to work properly, consumers of a class must limit their use of qualified member invocations. For example, using the following class definitions:

Code Example 16.

```
class alpha {
   public:
      long f();
};

class beta : public alpha {
   public:
      long g();
};
```

the following is syntactically valid but potentially semantically invalid:

```
long test(beta *pb) {
   return pb->alpha::f();
}
```

With the current class definitions the call to `pb->alpha::f()` is the same as a call to `pb->f()`. However in a subsequent release the definition of beta might be changed to:

```
class beta {
   public:
      long g();
      long f();
};
```

By overriding the function `f` in `beta`, the developer is free to completely ignore the implementation provided by `alpha`. Now the call `pb->alpha::f()` will skip over the call to `beta::f()` and may be counting on information in the instance that is not valid. For this reason, the use of qualified non-static member references should be limited to be within the scope of a member function, and only made to a direct base class.

When overriding a member in a derived class, the type of the member must not be changed. If the base class has a virtual member function that takes a `long` parameter and returns a `char`, it can only be overridden with a virtual member function that takes a `long` parameter and returns a `char`. This limitation extends to overriding overloaded member functions. If the function `f` is overloaded in the base class, it can only be overridden in a compatible fashion by overriding all versions of the function `f`.

Although Delta C++ does not allow the promotion of a member into a virtual base class, it does allow the overriding of members that are inheritedfrom a virtual base class.

4.4 Class Extension

Class Extension is the ability to add new base classes to an existing class. Delta/C++ allows the programmer to add new base classes provided the following two guidelines are obeyed:

1. If a class has no base classes or a single non-virtual base class, then a developer can split the class into two classes, one of which derives from the other.
2. If a class has more than one non-virtual base class or any virtual base classes then any number of non-virtual base classes can be added, as long as no virtual base classes are added to the class hierarchy above the class being changed. New base classes must be added to the end of the list of already existing base classes.

Completely general class extension would allow the developer to add an arbitrary number of new base classes to an existing class definition. We chose to introduce the restrictions we did, because supporting general class extension would have resulted in a substantial performance penalty during the construction and destruction of class instances. The largest performance penalty was the need to prepare for the initialization of an arbitrary set of virtual base classes. The exact set of virtual base classes would not be known until link time. A performance degradation is also encountered in the setting of virtual table pointers: with general class extension, the exact number of virtual table pointers would not be known until link-time.

Since most use of C++ is limited to single inheritance, we chose performance over functionality. This can be done as long as multiple inheritance or virtual base classes are not added to the hierarchy above the class being changed. For example, the following class definition:

```
class alpha {
   ....
};

class gamma : public alpha {
   ....
};
```

can be changed to:

```
class alpha {
   ....
};

//gamma has been split into beta and gamma
class beta : public alpha {
....
};

class gamma : public beta {
   ....
};
```

4.5 Member Reordering

Member reordering allows the movement of members within a class definition. This would normally be used to repack the layout of a class instance to reduce the size of memory being allocated for each instance.

5.0 Dynamic Classes

When using Delta C++, classes can either be *dynamic*, *internal dynamic* or *non-dynamic*. A class is *dynamic* only if the developer specifically makes the class dynamic by using a `pragma` statement. Dynamic class definitions can be changed in ways specified above. The actual size and layout of a dynamic class is not known until link-time. A class is *internal dynamic* if it either derives from a dynamic class or if one or more of its non-static data members is an instance of a dynamic class. Internal dynamic classes are not laid out until link-time, but from the developer's perspective, the definition of an internal dynamic class cannot be changed. *Non-dynamic* classes are resolved at compile-time and cannot be changed.

While we could have chosen to make all classes dynamic, we have allowed the developer to maintain control over which classes are dynamic and which are not. There are two reasons for this. First we wanted to provide a smooth transition path for developers using non-Delta C++ compilers to the Delta C++ compiler. During this transition period it is necessary for developers to be able to mix code compiled with non-Delta C++ compilers with code compiled with Delta C++. For example, in the initial release of Delta C++ none of the C++ shared libraries normally found on a Silicon Graphics system were compiled with Delta C++. By giving developers a way to control which classes are dynamic, a developer could make their own classes dynamic and still use non-dynamic versions of the classes provided in the C++ shared libraries.

5.1 Performance Considerations

Another reason for giving developers control over which classes are dynamic is the slight performance difference between dynamic and non-dynamic classes. Making classes dynamic does introduce some performance penalties. First, all constructors, destructors and assignment operators of a dynamic class have to be dropped out-of-line. This occurs even if those functions are declared to be inline. If any of these routines were inlined, it would not be possible to change the definition of that class. For very simple classes the added function call will have an impact on the performance of the program.

A second performance penalty comes from the fact that constructors, destructors and assignment operators for dynamic classes will always be generated, unless it would be illegal to do so. This implies that bit-wise copy of dynamic classes is not possible. This has the largest effect when passing or returning instance of objects to functions. In this case a temporary copy of the object is made and initialized using either the copy constructor or the assignment operator.

Constructors, destructors and assignment operators can also be more expensive with dynamic classes. All dynamic classes have constructors and destructors. Thus a class that derives from a dynamic base class will have a call from the derived class constructor to the base class constructor. If the base class was non-dynamic and did not require a default constructor then no code would be generated to initialize the base class. These routines are also more expensive for classes that have no virtual functions, or for classes that have more than one base class. In this case a runtime routine is called to set the virtual table pointers.

Another performance problem comes from a small number of cases where the optimizer cannot be as aggressive with code that implements dynamic classes. For example, given the following class definitions:

Code Example 17.

```
class alpha {
   public:
      static long x;
      ....
};

class beta {
   public:
      static long x;
      ....
};
```

the following code fragment:

```
i = alpha::x;
beta::x = 10;
j = alpha::x;
```

cannot be optimized to look like:

```
j = i = alpha::x;
beta::x = 10;
```

The reason that this optimization cannot be done is that in a subsequent release the class definitions given in Code Example 17 can be changed to:

Code Example 18.

```
class alpha {
   public:
      static long x;
      ....
};

class beta : public alpha {
   public:
      ....
};
```

With these class definitions, the assignment to `beta::x` would change the value of `alpha::x` between the assignments to the variables `i` and `j`.

Another optimization that cannot be done affects the code generated for accessing bit-fields. The position and size of a bit-field member can change from one release to another. Thus we must always generate the most general code when accessing a bit-field. For example in the following class definition:

```
class alpha {
   public:
      int i : 8;
      int j : 23;
      int k : 1;
};
```

it would be possible to optimize access to either the member `i` or `k` for non-dynamic classes. In the case of accessing the member `i`, we can generate a *load byte* operation. For accessing the member k, we can do a *load word* followed by an *and* operation to mask out all but the low-order bit. However, such optimizations cannot be performed for dynamic classes, since these code sequences would not work with a subsequent release where the class definition is changed to looks like:

```
class alpha {
   public:
      int a : 1;
      int i : 8;
      int j : 23;
      int k : 1;
};
```

A final optimization that cannot be performed is encountered when indexing vectors of dynamic classes. Since the size of a dynamic class is not known until link-time, constant folding and special handling of multiplications used when calculating addresses of array elements cannot be done.

The classes a developer chooses to make dynamic depends on a number of issues. If the code is under active development, then it is best that all classes are made dynamic. This will allow the developer to make changes to class definitions and not have to recompile a large amount of code to continue development. For code that is being compiled for release , the choice is a little more complicated. In a typical program or shared library, there are three sets of classes that are used:

1. There are classes imported from another library.
2. There are classes internal to the library or application.
3. There are classes exported for use by other libraries.

Whether the classes that the library or application is importing are dynamic or not depends on the providers of those classes. Classes that are internal to the library need not be dynamic, as there is no way a change to a definition of one of those classes can effect any code not contained in the library. Classes that are exported by this library should be dynamic. For most applications the set of exported classes is empty. This is not the case for applications that allow extension by dynamically loading in libraries. In this case the application will indeed export classes.

Classes can be made dynamic in Delta C++ using one of two pragmas. An individual class can be made dynamic by using `#pragma dynamic_class`. For example the following makes the class alpha dynamic:

```
class alpha {
#pragma dynamic_class
      ....
};
```

A set of classes can be made dynamic with the following pragma:

```
#pragma this_directory_tree_is_dynamic
```

This pragma is normally put in a header file and it implies that any class that contains at least one non-inline member function and whose declaration is found in a source/header file in that same directory or any of its sub-directories will be dynamic. This makes it easy for a shared library provider to make all of the exported classes found in that library dynamic.

6.0 Smart Build

During the development phase, Delta C++ gives developers the ability to make changes to class definitions and only recompile the code that implements the class being changed. Unfortunately, current build systems (like *make*) will force the recompilation of any file that includes the header file containing the changed class definition. This would negate some of the utility of using Delta C++ when developing code. In order to solve this problem, a companion feature called *Smart Build* was added to the Delta C++ compiler.

Smart Build keeps track of the types of changes made to header files, and in particular to class definitions, between compilations. It compares the previous class definition with the new definition and determines if it only has Delta-compatible changes. If it does then the compiler will only recompile a file if it implements the class (i.e. defines any member functions or static data members of the class).

For files that only use the class, it immediately exits after touching the output file, thereby keeping the build system from constantly wanting to recompile that file. If the changes to the class are not Delta-compatible then all files that use the modified header file will be recompiled.

7.0 Conclusions

There have been various attempts to solve the problems associated with releasing new versions of C++ libraries without obsoleting clients of these shared libraries. Current solutions to problem will usually impose various restrictions on the providers of new shared libraries:

- Some systems restrict library providers to making only trivial modifications (for example, the layout and size of classes may not change).
- Some systems restrict library providers from making full use of the C++ language (for example, multiple inheritance may not be used).
- Some systems allow greater latitude in making changes to existing libraries, but runtime performance is degraded (for example, accesses to all data members will be done using indirections).

The Silicon Graphics Delta/C++ compiler provides a comprehensive solution to the problems of shared library evolution. This compiler provides developers with the ability to make a wide range of compatible modifications to a C++ shared library, while using the full power of the C++ language. An important goal in the development of Delta/C++ is that runtime performance should not be compromised, and preliminary measurements indicate that we have achieved this goal.