SOFTWARE—PRACTICE AND EXPERIENCE
*Softw. Pract. Exper.* 2006; **36**:139–156
Published online 15 September 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.686

# Fast dynamic casting

Michael Gibbs[1,*,†] and Bjarne Stroustrup[2]

[1]*Lockheed Martin Aerospace, MZ 6602, P.O. Box 748, Fort Worth, TX 76101, U.S.A.*
[2]*Department of Computer Science, TAMU 3112, Texas A&M University, College Station, TX 77843, U.S.A.*

**SUMMARY**

**We describe a scheme for implementing dynamic casts suitable for systems where the performance and predictability of performance is essential. A dynamic cast from a base class to a derived class in an object-oriented language can be performed quickly by having the linker assign an integer type ID to each class. A simple integer arithmetic operation verifies whether the cast is legal at run time. The type ID scheme presented uses the modulo function to check that one class derives from another. A 64-bit type ID is sufficient to handle class hierarchies of large size at least nine levels of derivation deep. We also discuss the pointer adjustments required for a C++ dynamic_cast. All examples will be drawn from the C++ language. Copyright © 2005 John Wiley & Sons, Ltd.**

KEY WORDS: dynamic casting; C++; modulo; embedded systems; hard real-time

## INTRODUCTION

When writing C++, we often convert a pointer to a derived class (subclass) to a pointer to one of its base classes (superclasses). This is the basis for the object-oriented programming style where we use the interface provided by a base class to access the implementation in the derived class. Occasionally we want to cast such a pointer to a base class back to a pointer to its derived class. C++ supports this functionality with the **dynamic_cast** type conversion operation. Consider

```
Source* ptr = new Actual();
Target* t = dynamic_cast<Target*>(ptr);
```

We create an object of type **Actual**, assign its pointer to **ptr** (where **Source** must be a base of **Actual**), and attempt to cast pointer **ptr** to a pointer to **Target**. If the **Actual** object contains a single public base of class **Target**, the offset to the beginning of the **Target** object is computed and a pointer to the **Target** object returned. If the class **Actual** does not inherit from **Target**, the cast fails and returns **0**. If the **Actual** object contains more than a single base of type **Target** (due to multiple non-virtual derivation),

*Correspondence to: Michael Gibbs, Lockheed Martin Aerospace, MZ 6602, P.O. Box 748, Fort Worth, TX 76101, U.S.A.
†E-mail: james.m.gibbs@lmco.com

*Received 22 March 2004*
*Revised 17 December 2004*
*Accepted 11 March 2005*

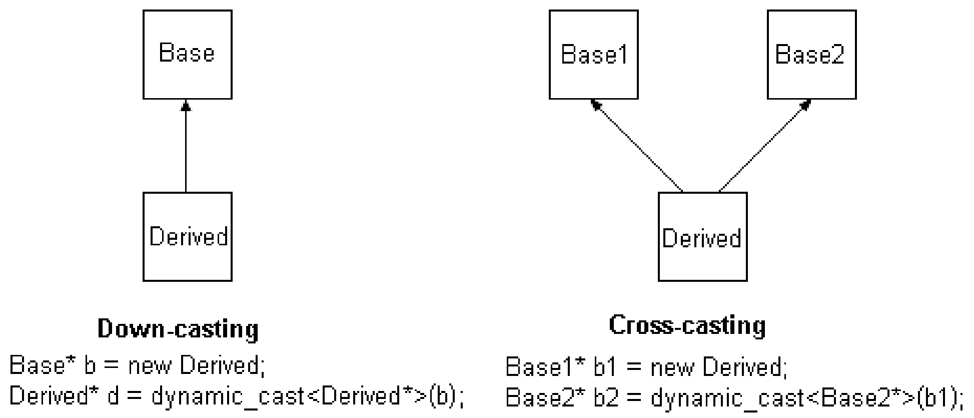

Figure 1. Down-casting and cross-casting in C++.

the cast is ambiguous and returns **0**. The exact specification can be found in section 5.2.7 of the ISO C++ standard [1]. A more tutorial explanation can be found in section 15.4 of [2]. A brief discussion of performance and predictability of **dynamic_cast** can be found in sections 3.2.8 and 4.2.2.1 of [3]. Figure 1 shows the common cases of down-casting and cross-casting.

Dynamic casting is useful in many applications, but current implementations of this functionality are slow compared with other C++ operations. Current implementations of **dynamic_cast** require extra type information to be kept for each class with a virtual function. When **dynamic_cast** is called, the algorithm will traverse a data structure representing all of **Actual**'s base classes, searching for a match for the **Target** type. If the type is not found or is found more than once, **dynamic_cast<Target*>(ptr)** returns **0**.

## A FAST, PREDICTABLE ALTERNATIVE

Unfortunately, conventional implementations of **dynamic_cast** render it unsuitable for real-time applications where speed, small memory footprint, and predictable performance are essential. We present an alternative implementation strategy designed to meet those constraints. Our solution is primarily aimed at embedded systems where whole program analysis is typically feasible, but (with some additional effort) it can be extended to systems with dynamic linking. Please see the section entitled 'Dynamically loaded classes' for details.

An alternative to walking the inheritance tree is to assign an integer type ID at link time to each class, most likely kept in the virtual table. This ID needs to be a static constant associated with the class. The linker will assign these integers to the classes such that the following property holds: there is a function **F** which takes two integer arguments and returns an integer such that **F(derived_type, base_type) == 0** if and only if the class represented by the ID **derived_type** is a derived class of the class represented by the ID **base_type**.

The trick lies in finding a function **F** that satisfies this criterion and in determining the size of the integer necessary to accommodate realistic software projects. We would prefer 32- or 64-bit integers so that the function **F** could be performed efficiently. If the evaluation of the function takes longer than walking the derivation tree, then nothing has been gained except a few bytes of program space.

Our suggestion for an **F** function is integer modulo. To construct the type ID, a different prime number is associated with each class. The type ID for that class will be that prime number times the prime number for each of its base classes. Thus, the type ID for a class will be evenly divisible by the type ID of any of its base classes, and only by its base classes. This is guaranteed by the uniqueness of the factorization of integers into prime factors. Given an arbitrarily large integer type ID, any hierarchy can be represented this way. We wish to determine how large a hierarchy can be represented using a machine's built-in integer types.

This method has the advantage that integer math is very fast for numbers that fit in a machine register and should be much faster than the usual tree-walking routine. Importantly, the check that one class derives from another is performed in a known, constant time. If the full type of the object being cast contains more than one copy of the target class, the cast will be ambiguous. In this case, the ID for the full type will be divisible by the square of the prime for the target class. Performing a second divisibility check determines if the cast can be done unambiguously. If the cast can be performed unambiguously, the address of the returned object must be computed, as described below.

There are a number of heuristic methods for keeping the size of the type ID to a minimum number of bits. To prevent the type IDs from being any larger than necessary, the classes are sorted according to priority. The priority for a class is the maximum number of ancestors that any of its descendants has. The priority reflects the number of prime factors that are multiplied together to form a class' type ID. Classes with the highest priority are assigned the smallest prime numbers.

If we had to assign each class a unique prime number, the type IDs would quickly get very large. However, this is not strictly necessary. While all classes at the root level (those having no base classes) must be assigned globally unique prime numbers, independent hierarchies can use the same primes for non-root classes without conflict. Two classes with a common descendant then cannot have the same prime and none of their children may have the same prime. This also implies that no two children of a class may have the same prime. In all other cases, the primes can be duplicated across a level of the hierarchy. For example, in a tree structure two classes on the same level of the tree never have a common descendant, so they may have identical sub-trees beneath them without a conflict.

## CONFLICTING CLASSES

Figure 2 shows an error that an overly simple assignment of prime multipliers could cause. The proposed scheme avoids this problem. Note that the type ID for class **D**, 10, divides the type ID for class **E**, 210, even though class **E** does not derive from class **D**. This has happened because we assigned the same prime multiplier, 2, to classes **C** and **D**, thinking they were independent. They are not independent because classes **A** and **B** share a common descendant, class **E**. In our terminology, classes which share a common descendant are said to be in the same *group*. Class **A** and class **B** are in the same group. This motivates the rule for assigning primes: if any parent of class **F** is in the same group as any parent of class **G**, then class **F** and class **G** must be assigned different prime multipliers. This rule is probably too strict, but we do not see any obvious way to loosen it.

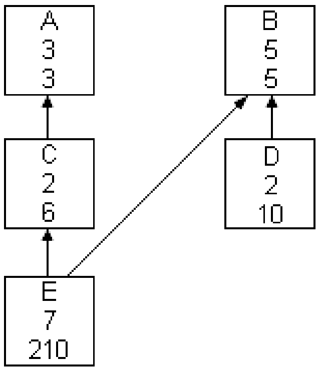

Figure 2. Justification for the conflict rule.

The classes can be divided into levels and each level assigned a distinct set of prime numbers. The level of a class must be greater than the level of any of its base classes and a class with no base classes (i.e. a root class) has a level of 0. One way to assign level numbers is to make the level the maximum number of upward links that need to be traversed to reach a root node. For tree-like inheritance trees, the level number is very intuitive.

## EXAMPLES

Figure 3 shows a simple hierarchy of Vehicle classes. (This hierarchy is for illustrative purposes only. This is not intended to be an actual categorization of all possible vehicle types.) In this example, all classes descend from **Vehicle**, so the **Vehicle** class may be assigned a type ID of 1. The type ID for **Jet** is 663 which can be uniquely factored into primes as $3 \times 13 \times 17$. Therefore, **Air** (type ID 3) and **FixedWing** (type ID $39 = 3 \times 13$) are base classes for **Jet** since 663 is evenly divisible by 3 and by 39. As another example, suppose we have a **Vehicle** pointer that actually points to an instance of **Motorcycle**. When we attempt to perform a **dynamic_cast** to a pointer to **Wheeled**, the cast succeeds because the type ID for the **Motorcycle** is 1771 which is evenly divisible by the type ID for **Wheeled**, 77. If we attempt to perform a **dynamic_cast** to **Ship**, the call would return a 0 because 1771 is not evenly divisible by 55.

```
Vehicle*   veh = new Motorcycle();
Wheeled*   whl = dynamic_cast<Wheeled*>(veh); // Succeeds
Ship*      ship = dynamic_cast<Ship*>(veh); // Fails
```

Also note how the prime factors are re-used in different parts of this hierarchy. For example, **RotaryWing**, **Ship**, and **Wheeled** all are represented by the prime factor, 11. Their type IDs are distinguished by having different base classes. Since **Air** is represented by 3 and **Water** by 5, as long as we never use the factors 3 and 5 anywhere below level 1 in the tree, we can never confuse an air vehicle with a water vehicle.

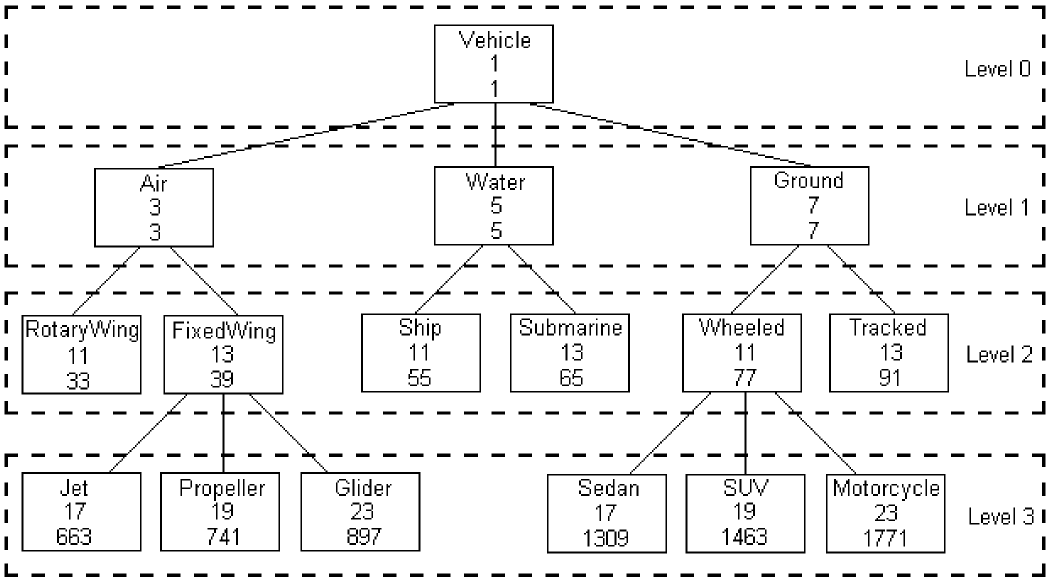

Figure 3. Vehicle hierarchy with prime assignments and type IDs.

Now let us explore some larger structures to determine how large a hierarchy of classes can be represented in a 32-bit or a 64-bit type ID. Here are some examples of type ID assignment for tree structures.

A binary tree 7 levels deep (255 classes) has a maximum type ID of $2 \times 5 \times 11 \times 17 \times 23 \times 31 \times 41 \times 47 = 2\,569\,288\,370$, which will fit in a 32-bit word. The value 1 is not assigned to the root node to account for the possibility that there may be other classes that do not derive from the root node.

Level 0 (1 node) is assigned the prime 2.

Level 1 (2 nodes) is assigned 3 and 5.

Level 2 (4 nodes) is assigned 7 and 11.

Level 3 (8 nodes) is assigned 13 and 17.

Level 4 (16 nodes) is assigned 19 and 23.

Level 5 (32 nodes) is assigned 29 and 31.

Level 6 (64 nodes) is assigned 37 and 41.

Level 7 (128 nodes) is assigned 43 and 47.

Figure 4 is a diagram of levels 0 through 4 of the binary tree, showing the assigned primes above the type IDs for each class. A binary tree 12 levels deep (8191 classes) has a maximum type ID of $5.956 \times 10^{18}$, which will fit in a 64-bit integer. The maximum type ID for a binary tree of 21 levels will fit in a 128-bit integer and 37 levels require a 256-bit integer (see Figure 5 for chart).

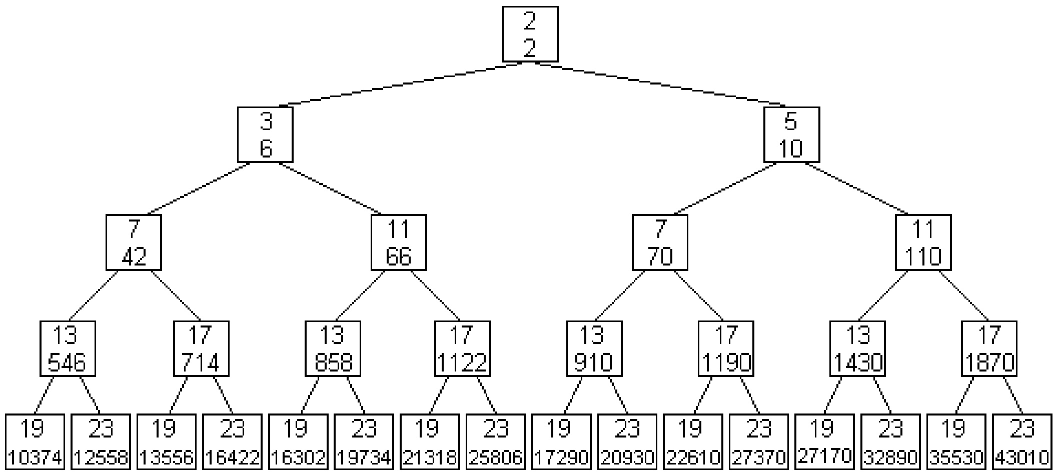

Figure 4. Type IDs in a binary tree.

A trinary tree six levels deep (1093 classes) has a maximum type ID of $2 \times 7 \times 17 \times 29 \times 41 \times 53 \times 67 = 1\,004\,869\,082$, which will fit in a 32-bit integer. A trinary tree 11 levels deep (265 720 classes) has a maximum type ID of $1.452 \times 10^{19}$, which will fit in a 64-bit integer. The maximum type ID for a trinary tree of 19 levels will fit in a 128-bit integer and 34 levels require a 256-bit integer.

The worst-case hierarchy is one in which a class has a large number of immediate base classes. The curve marked 'Unary' in Figure 5 shows the number of bits required for a given number of base classes. In particular, a class with nine base classes requires a 32-bit integer, 15 base classes require a 64-bit integer, 26 base classes require a 128-bit integer, and 43 base classes require a 256-bit integer.

A tree nine levels deep with branching ratios (100, 100, 100, 10, 6, 5, 4, 3, 2), which has over 12 billion classes, has a maximum type ID of $1 \times 2213 \times 1451 \times 733 \times 113 \times 71 \times 43 \times 23 \times 11 \times 3 = 6.163 \times 10^{17} = 2^{59.096}$, which will fit in a 64-bit word (see Figure 6). This example is inspired by the Visual Component Library (VCL) hierarchy of Borland C++ Builder 6, which has similar branching ratios, but is far from a full tree. From these examples, the primary limiting factor seems to be the depth of the tree, though there is a tradeoff between tree width and tree depth. For the nine-level example (Figure 6), if the trees underneath a 100 branching factor are not all full (as is highly likely), then a significant reduction in maximum type ID may be made by assigning the smallest primes to those branches that have trees underneath them and the larger primes to the branches that terminate early.

A strategy where classes bid on the prime values according to some pre-computed priority is effective in assigning small primes to those classes that have deep trees underneath them.

The type IDs get large when a class has a large number of classes from which it derives. This can either be due to the depth of the tree or from multiple inheritance. The proposed scheme will generate type IDs larger than 64 bits if there are classes with large numbers of base classes (usually more than 10), though a class can have up to 15 base classes if there is only one class with that many base classes.

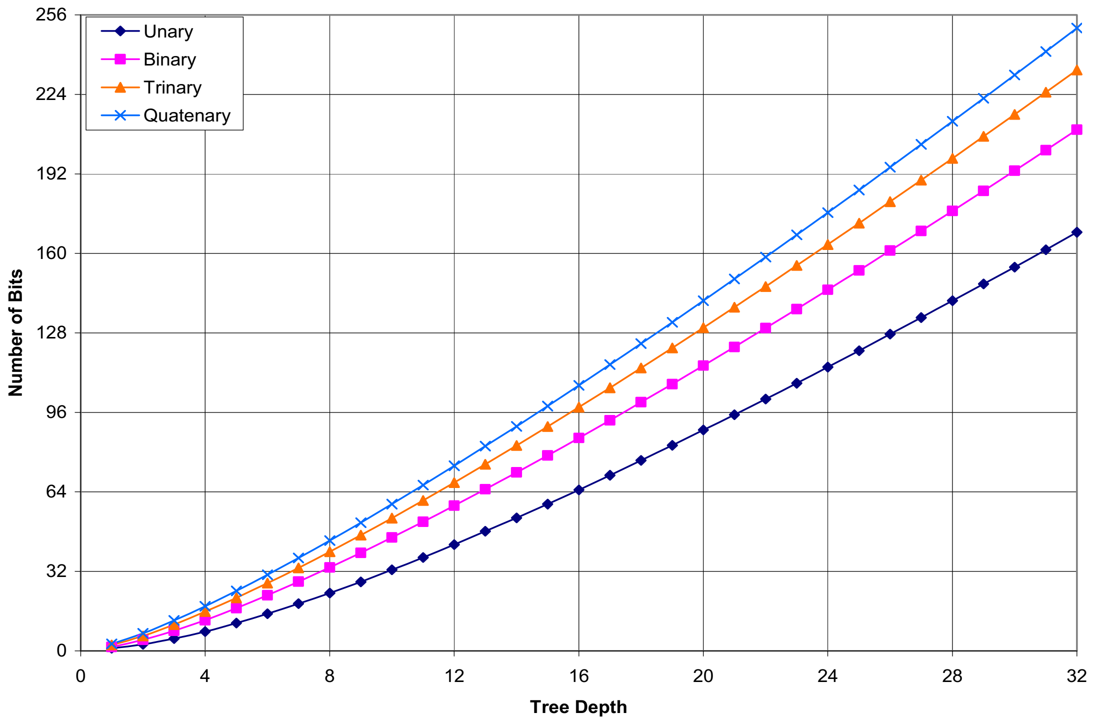

Figure 5. Depth of tree hierarchy versus number of bits in type ID.

## VIRTUAL INHERITANCE AND AMBIGUITIES

Virtual inheritance (also known as repeated inheritance in some languages) complicates the assignment of IDs to classes, but the scheme still works with appropriate modification. Every class has three possible types of ancestor classes: ambiguous, virtual, and unambiguous non-virtual. A base class is ambiguous if more than one copy of it exists in the derived class. A base class is virtual if it is declared virtual in the inheritance tree. A virtual class is included only once in the derived class. Classes which appear exactly once in the base class and are not declared virtual are unambiguous non-virtual. In the previous discussion we have considered only cases with unambiguous non-virtual base classes. Here is an example of a more complex class:

```
class A { . . . };
class V { . . . };
class C : public A, public virtual V { . . . };
class D : public C { . . . };
class E : public C { . . . };
class F : public D, public E { . . . };
```

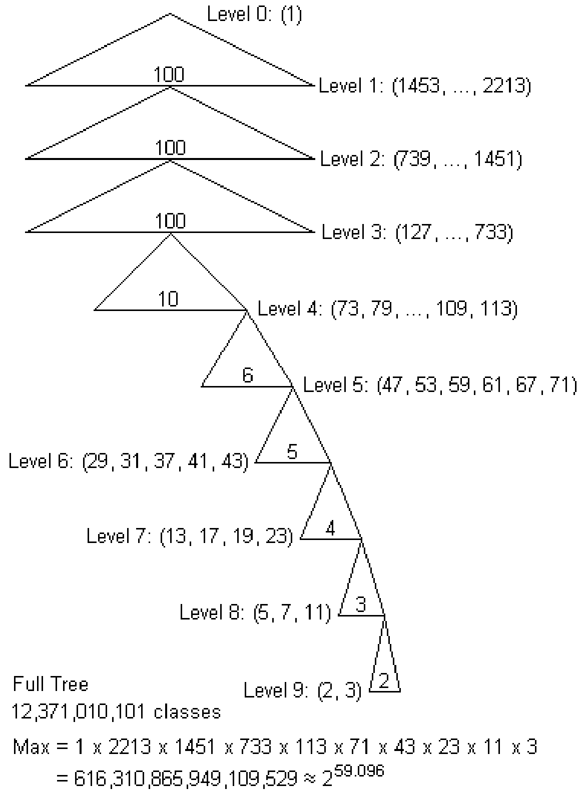
Level 0: (1)
100
Level 1: (1453, ..., 2213)
100
Level 2: (739, ..., 1451)
100
Level 3: (127, ..., 733)
10
Level 4: (73, 79, ..., 109, 113)
6
Level 5: (47, 53, 59, 61, 67, 71)
5
Level 6: (29, 31, 37, 41, 43)
4
Level 7: (13, 17, 19, 23)
3
Level 8: (5, 7, 11)
2
Level 9: (2, 3)
Full Tree
12,371,010,101 classes
Max = 1 x 2213 x 1451 x 733 x 113 x 71 x 43 x 23 x 11 x 3
= 616,310,865,949,109,529 ≈ 2^59.096

Figure 6. A full nine-level tree of classes with branch ratios (100, 100, 100, 10, 6, 5, 4, 3, 2).

In this more complicated case, for each class we must keep track of the type of each of its ancestor classes. If each class **C** has been assigned a prime $P_C$, the ID for class **D** is the product of $P_D$, the primes of each of its virtual and its unambiguous ancestors and the square of the primes of its ambiguous ancestors. The type IDs for the above example are shown in Figure 7. The **Value** field listed for each class will be explained in the 'Computing the offset' section.

Given an object of type **F** and a pointer to one of the **A** objects in the **F**, we can cast the **A** pointer to a pointer to the **C** of which the **A** is a part. We cannot cast a pointer to **V** to a pointer to **C** because we would not know which **C** we should choose. In addition, we can cast from either **A** to **D**, **E**, or **F**.

The ID for each class is divisible by the IDs of all classes from which it inherits. When attempting to cast an **F** to a **C**, we first verify that $ID_F \bmod ID_C = 180\,180 \bmod 78 = 0$. This means that **F** contains at least one **C**. Then we check $ID_F \bmod P_C^2 = 180\,180 \bmod 9 = 0$ which means that the cast is ambiguous. Thus a cast from **F** to **C** should return 0.

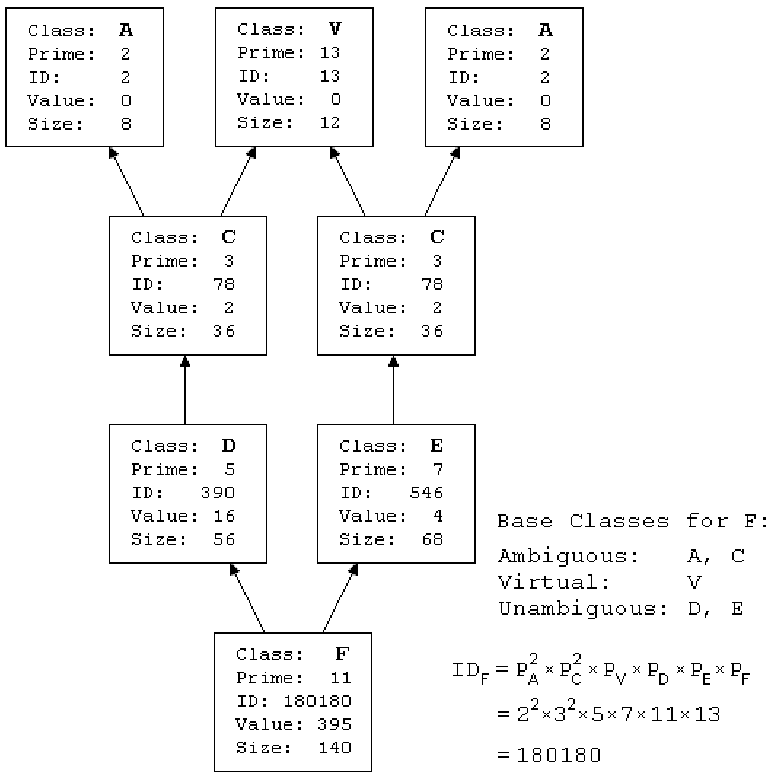

Figure 7. ID computation with virtual inheritance.

## COMPUTING THE OFFSET

In addition to determining if the actual class derives from the target class, we must also compute the return value of the dynamic cast, which is a pointer to an object of the target type. Our initial code example is repeated for reference:

**Source* ptr = new Actual();**
**Target* t = dynamic_cast<Target*>(ptr);**

There are three steps in performing a dynamic cast.

1. Finding the type and address of the **Actual** object given the **Source** pointer (**ptr**).
2. Determining if the **Actual** object derives unambiguously from the **Target** class. If it does not, null is immediately returned.
3. Computing the offset of the **Target** class within the **Actual** class to return a pointer to a **Target** object.

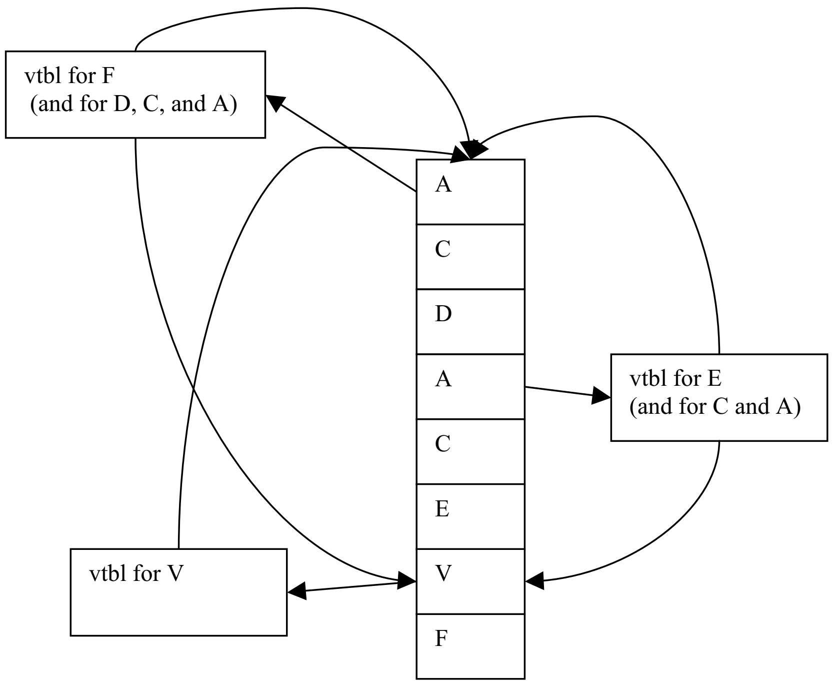

Figure 8. Layout of class **F** showing virtual tables.

Each step exists in every ISO Standard C++ implementation to support virtual function calls and **dynamic_cast**. We will outline a traditional scheme and then describe how to optimize all or parts of steps 2 and 3.

Consider the complex object in Figure 7. There are several ways to lay out such an object, and several different ones are in use. What we describe here is a scheme that is a bit simpler and faster than most because it does not try to match the C++ Application Binary Interfaces (ABIs). Figure 8 shows a plausible layout for an **F** object and its basic structures supporting virtual function calls.

Each **A** and **V** have a separate **vtbl** ('virtual function table'). **F** and **D** and **D::C** share **F::D::C::A**'s **vtbl**. Similarly, **E** and **E::C** share **E::C::A**'s **vtbl**. Such sharing is an almost universal optimization. Each **vtbl** has a pointer to the start of **F** (their joint 'most derived class'). Given a pointer to any base class object, we can find a **vtbl**, and through that **vtbl** find the start of the **F** object.

Step 1 of the implementation of **dynamic_cast** (finding the complete object—as it was created—here **F**) is easy and fast in any implementation. Here, we simply follow a pointer stored in each **vtbl**.

In current implementations, steps 2 and 3 are performed by traversing (base, offset) tables accessed through the **vtbl** until we find an entry where both base and offset match the pointer with which we started (the operand to **dynamic_cast**). Because of the possibility of an ambiguity, we have to search further than the first match.

The proposed scheme will perform step 2 (checking if the dynamic cast will succeed) with a check that the ID of the actual class is evenly divisible by the ID of the target class. If it is not, null is returned. If it is divisible, it checks if the actual ID is divisible by the square of the target prime. If it is divisible, the cast is ambiguous and null is returned. If it passes both these checks the cast can be performed unambiguously. Clearly, this will typically be faster than any search through a list of base classes. In particular, the modulo test will not need to access memory repeatedly. This optimizes the performance of an unsuccessful cast, making that operation constant time.

It is not uncommon for **dynamic_cast** to be used to compare an object against several possible classes. In such uses, the cast will fail repeatedly. The proposed scheme catches this failure with a single modulo computation, resulting in a significant decrease in the effort to locate the correct class type; we only search (base, offset) tables when we know we will find a base.

However, as long as we do any form of search, the total cost of **dynamic_cast** cannot be constant. Without further improvement our implementation has the cost of a failed **dynamic_cast** constant and successful **dynamic_cast** has a worst-case cost linear with the number of classes in the most derived object (using the same techniques as existing implementations). We can do better.

The complexity of finding the offset depends on the complexity of the class. We can distinguish three cases:

1. single inheritance only;
2. multiple inheritance without any base being replicated;
3. multiple inheritance with one or more bases replicated.

A compiler can trivially distinguish between these three cases and generate different—and optimal—code for each.

The first case (single inheritance) is interesting because no offset modification is needed, so that the problem is solved with close-to-zero added cost in time or space. All that is needed is an indicator in the **vtbl** that no offset calculation is needed—all sub-objects share a single pointer to the ultimate base. Basically, the data structure needed is what we showed in Figure 8 and a **dynamic_cast** is done in constant time, and very fast.

The second case (no base appearing twice) has the property that we can provide a simple (base, offset) map. The simplest implementation would just make that map a vector and search it linearly. Since there is no need for an ambiguity check, the cost will be a maximum of $N + 1$ comparisons, where $N$ is the number of bases in the object, and the average cost $(N + 1)/2$. The $+1$ is to count the most derived object itself. Better performance may be achieved by sorting the vector and using a binary search.

Once the base is found, a simple addition to the pointer to the complete object gives the desired pointer result. The (base, object) map would be attached to or be part of the **vtbl** for the complete object. Note that the offsets can all be relative to the start of the complete object and applied to the pointer to the complete object independently of which base class was pointed to by the **dynamic_cast** argument.

The third case (one or more bases appearing twice) is more complicated. For example, consider trying to obtain **dynamic_cast<C*>(pa)**. If **pa** points to the first **A**, we should get a pointer to the first **C** and if **pa** points to the second **A** we should get a pointer to the second **C**. To do that, we could use a single table with complicated offsets that we can search with the offset of **pa** in the complete object. A simpler approach is to use a table per **vtbl** (Figure 9). In this case, we search the relative offset table

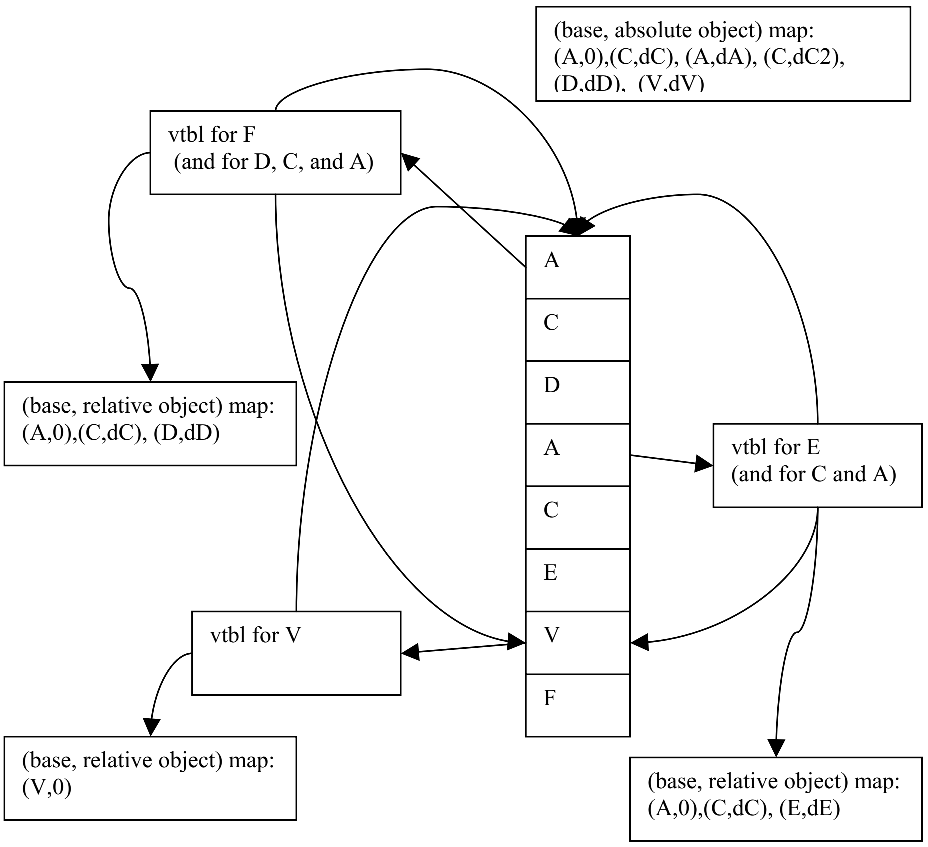
(base, absolute object) map:
(A,0),(C,dC), (A,dA), (C,dC2),
(D,dD), (V,dV)
vtbl for F
(and for D, C, and A)
A
C
D
A
C
E
V
F
(base, relative object) map:
(A,0),(C,dC), (D,dD)
vtbl for E
(and for C and A)
vtbl for V
(base, relative object) map:
(V,0)
(base, relative object) map:
(A,0),(C,dC), (E,dE)

Figure 9. Layout of class **F** showing virtual tables and object map.

first to see if the target class is derived from our base by single non-virtual derivation. If so, we use the relative offset found; otherwise, we search the absolute offset table of the bases that are unambiguously accessible from the most derived class.

Case 1 is unbeatably cheap and constant time. Case 2 is relatively inexpensive and has an easy-to-calculate upper bound that may be as good as constant time. Case 3 is pretty ugly; it does have an upper bound, but that bound is not trivial for humans to estimate and current implementations show that a **dynamic_cast** implemented roughly this way has a cost comparable with several function calls (about 10 for examples like that of Figure 7).

Even so, the modulo technique provides a significant improvement over traditional techniques: a common use of **dynamic_cast** is to compare an object against several possible class types. In this usage, the cast will fail in most cases. The proposed scheme catches this failure with a single modulo computation, resulting in a significant decrease in the effort to locate the correct class type.

Also, the knowledge that the cast will succeed if we get to step 3 (determine the offset) simplifies that computation.

When we are not satisfied by a worst case estimate of the cost, cannot afford the elaborate mechanism of case 3, or need an absolute constant-time guarantee, we can optimize by using the modulo function again. We use the Chinese remainder algorithm to compute our offsets. We put all the offsets into an indexed table and assign each class another integer value **V** such that $V_D \bmod P_B$ gives the index of the offset of **B** within **D** in the offset table. We will illustrate this technique using the hierarchy in Figure 7.

For example purposes, we will assume that **A** is 8 bytes and **V** is 12 bytes. **C** has 16 bytes over and above what it inherits from **A** and **V**, making its total size 36 bytes. **D** has 20 bytes over and above what it inherits from **A**, **C**, and **V**, making its total size 56 bytes. **E** has 32 bytes over and above what it inherits from **A**, **C**, and **V**, making its total size 68 bytes. **F** has 40 bytes over and above what it inherits from **A**, **C**, **D**, **E**, and **V**, making its total size 140 bytes.

Offset of **A** in **C**: 0
Offset of **V** in **C**: 24
Offset of **A** in **D**: 0
Offset of **C** in **D**: 8
Offset of **V** in **D**: 44
Offset of **A** in **E**: 0
Offset of **C** in **E**: 8
Offset of **V** in **E**: 56
Offset of **D** in **F**: 0
Offset of **E** in **F**: 44
Offset of **V** in **F**: 140

Create a table of unique offset values. The offset zero occurs frequently and should be put at index zero:

| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Offset | 0 | 8 | 24 | 44 | 56 | 140 |

Assign primes and values to each class at link time using the Chinese remainder algorithm (these values appear in Figure 7):

| Class | **A** | **C** | **D** | **E** | **F** | **V** |
|---|---|---|---|---|---|---|
| Prime | 2 | 3 | 5 | 7 | 11 | 13 |
| Value | 0 | 2 | 16 | 4 | 395 | 0 |

For example, for **F** we need $V_F \bmod 5 = 0$ since **D** has offset 0 in **F** and we need $V_F \bmod 7 = 3$ since **E** has offset 44 in **F**. Note that offset 44 appears at index 3 in the offset table. We also need $V_F \bmod 13 = 5$ since **V** has offset 140 in **F**. We can see that $V_F = 395$ satisfies these constraints.

Performing a dynamic cast from an object of type **F** to an **E**, first we verify that the cast is legal (it is), then we compute the offset. $V_F \bmod P_E = 395 \bmod 7 = 3$. The offset found at index 3 in the offset table is 44, so **E** is to be found 44 bytes from the beginning of **F**.

The Chinese remainder theorem guarantees that we can find a number that satisfies all the necessary modulo constraints provided all the base classes of a class are given unique prime values. We already have the constraint that no two classes with a common descendant may be assigned the same prime, so we could re-use the primes from the validity check in the offset computation as well. The prime chosen for a class must be larger than any offset index it produces for a descendant class, which provides another constraint on the prime that may be assigned to a class.

We now have a system where each class has three additional integers attached to its virtual table and dynamic casting may be performed in a constant, known time. This modulo method for finding offsets is not as fast as the conventional method for the single inheritance case, but should equal the conventional method for non-repeated bases (at least for classes with many bases) and clearly beat the conventional method in the general case with repeated classes. The optimal combination of techniques will depend on the mix of classes used and the relative cost of machine operations (especially memory access). However, our desire for fast, constant time (or firm upper bound) performance is clearly met.

## DYNAMICALLY LOADED CLASSES

The main aim of this paper is to demonstrate a way of implementing **dynamic_cast** that is suitable for embedded systems programming so we will not develop a dynamic linking extension here. However, such an extension appears to be feasible; whether it is practical in real-world systems is another question (which we do not propose to answer here).

The scheme for assigning type IDs assumes the entire hierarchy of classes is available at the time the type IDs are assigned. Creating type IDs for dynamically loaded classes might be done as follows.

A code fragment (CF), that is, a shared library (dynamically linked library in Windows) or a main program, contains all the information it needs to resolve any dynamic cast involving classes it defines. The hierarchy above any new class brought in by the CF is known when the code is compiled. To support dynamic casting, each CF could assign type IDs independently of all others. In order to perform a dynamic cast, the most derived type of the object being cast is determined, and the CF in which it was defined. Within that CF, the type ID of the class to which we are casting is determined. If the class is not known to that CF, then the cast fails and returns null. Otherwise, the modulo check is performed as usual.

This scheme implies that a class will have one type ID for each CF that uses it. It is necessary that each class knows in which code fragment it was defined. The type information for a class will effectively consist of a map from CF ID to type IDs. This could be implemented in several ways.

## PRIORITIZING THE CLASSES

We have shown that a 64-bit type ID is sufficient for several large systems; we would like to concentrate on heuristics to keep the type IDs as small as possible, so that as large a hierarchy as possible can be represented with an ID of this size.

The following terminology will be used in the discussion below:

parent—a direct base class;
child—a direct derived class;
ancestor—a direct or indirect base class;
descendant—a direct or indirect derived class;
sibling—two classes are siblings if they have at least one common parent;
root—a class having no parents;
leaf—a class having no children;
level—the level of a class is the maximum number of links one must follow up the hierarchy from it to reach a root node; root nodes are at level zero;
group—a set of classes on the same level having a common descendant.

Much smaller maximum type IDs will be obtained if care is taken to give smaller prime values to classes that have more levels underneath them. This can be done by assigning a priority to each class, giving classes with higher priorities smaller prime values. The priority value should reflect the expected size of the largest type ID of any of a class's descendants. The largest type ID values always occur at the leaves of the inheritance tree, so once the expected maximum value for each leaf has been computed, the values can be propagated upward through the tree to set the priority values. Here are four possible schemes for computing priority.

1. The priority of each leaf class is the number of ancestor classes it has. This is the primary method used in our test program.

2, 3, 4. Assign a range of primes to each level, with wider levels receiving larger starting values. Each node is given a value proportional to the logarithm of the (minimum, mean, or maximum) prime in its level. The priority for a leaf class is the sum of these values for itself and all its ancestor classes.

In all cases, the priority of a non-leaf class is the maximum of the priorities of all its derived classes.

Here is how to use the priorities to assign prime values: sort all classes by priority, highest priority first. Assign the highest priority class the smallest prime in the set of primes for that level, the next highest priority class the second smallest prime, and so on. We can assign the same prime to two or more classes on a level if none of their parents have a common descendant.

## OPTIMIZATIONS

There are several optimizations which will make the numbers smaller, hence able to represent larger class diagrams.

- A class which is the only child of each of its parents can be assigned the prime value 2 if it has only one parent or the value 1 if it has more than one parent. This assumes that the prime 2 is reserved exclusively for this purpose.
- If there is a single root class in the system, from which all other classes derive, it can be assigned a 'prime' value of 1, and thus a type ID of 1.
- Disjoint hierarchies can be broken into levels and assigned type IDs independently of one another, provided that all root classes have unique prime type IDs.

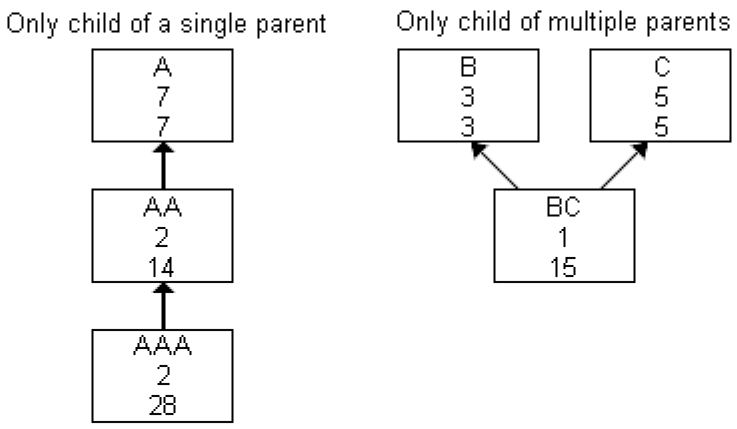

Figure 10. Simple optimization.

- For unbalanced trees, it may be beneficial to give levels non-consecutive sets of primes. For example, if a level has a width of six, but at most three of the siblings in any family have deep trees underneath them, then the level could take three small primes and three large primes. This can be implemented by having levels bid on the primes according to the priority of their classes.

The first two of these optimizations are easy to implement. For cases where there is a single derived class of another class, an 'only child', we can extend the hierarchy quite deep using these techniques. These two techniques are incompatible with the Chinese remainder offset scheme and can break if another class is added to the hierarchy. Breaking the hierarchy into disjoint pieces should be unnecessary if the last optimization (bidding) is used.

Figure 10 illustrates these first two optimizations. A derived class must always have a type ID larger than any of its base classes so that the algorithm never identifies the base class as deriving from the derived class. However, we require only that the derived class is divisible by all its base classes and distinguishable from all its siblings.

The bidding strategy is quite helpful for unbalanced trees and can be implemented as follows. Each level needs a set of prime numbers assigned to it. First, every class is assigned a priority by method 1 as described above. The classes are sorted by priority, highest priority first. For classes with equal priority, those with smaller level number come first. The classes are examined one by one, in order of priority. The class is compared to all other classes on its level that have already been assigned primes, checking for conflicts. The smallest available (non-conflicting) prime in the level is assigned to the class, adding a new prime to the level if necessary. The primes are thus dealt out to the levels starting with 3, 5, 7, 11, etc. The value 2 is reserved for only-child classes.

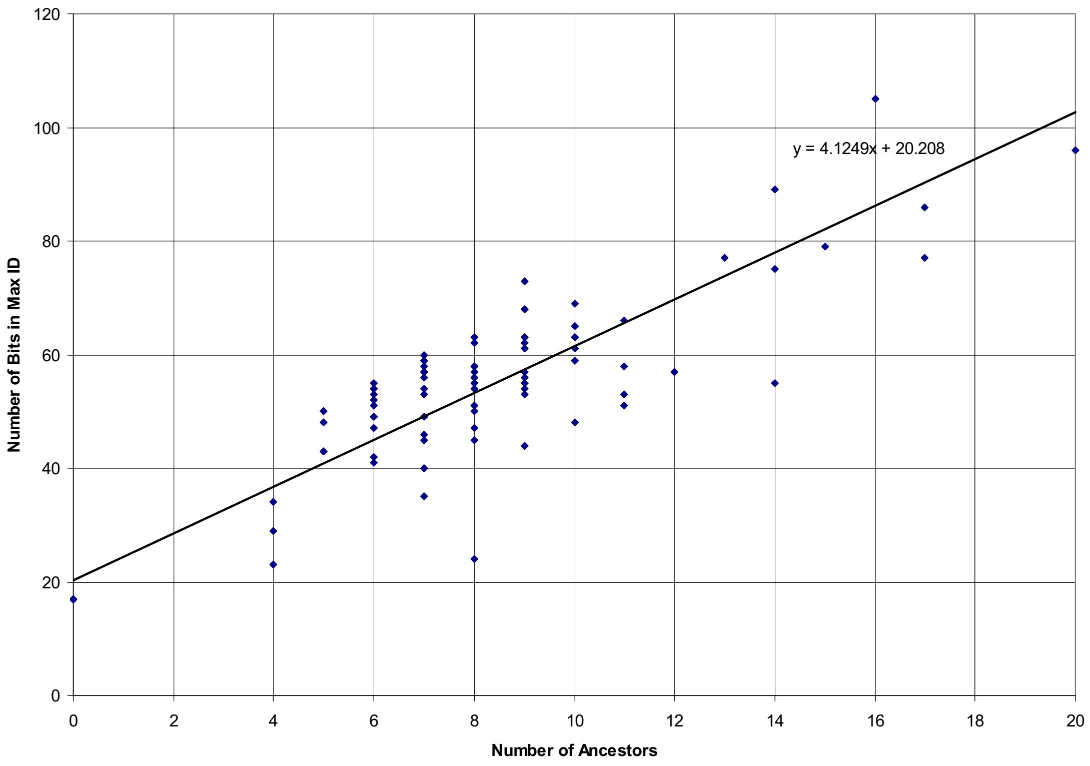

Figure 11. For hierarchies with 8192 classes, the number of ancestors of the class with the largest ID versus the number of bits in the largest ID.

## SIMULATION

We wrote a simulation that randomly generates a class hierarchy with specified statistics, then assigns type IDs and reports the largest type ID used. We used the bidding strategy with the optimizations discussed above. It ran in under a second for 1024 classes and in about 18 s for 8192 classes on a 533 MHz Pentium IV PC. We suspect that most of this time is due to the clustering algorithm and could be optimized substantially.

We ran the simulation for various numbers of classes and various statistics. The results are shown in Figure 11. On average, for a hierarchy with 8192 classes, the type ID of a class with 11 ancestors will just fit in a 64-bit integer (see chart in Figure 5). A class with 26 ancestors is expected to require a 128-bit ID. The more convoluted class hierarchies required more bits to represent them, as expected.

## NOTES AND FUTURE DIRECTIONS

Only classes that have their type ID explicitly accessed or are of a class that is the source or target of a dynamic cast need to have a type ID computed. This may be a small fraction of the total number of classes in the system.

The size of the integer needed could be dynamically changed by the linker according to the size of the problem, or a variable-sized integer class could be used for the type IDs.

It may be possible to devise other strategies for assigning type ID integers, such as a hashing function that returns 0 when the two integer type IDs satisfy a particular pattern. This may allow the system to cover greater numbers of classes than is possible with the modulo function. A hashing system would have some probability of generating a false positive, believing that one class was derived from another when it was not. The linker would have to verify that this did not happen for the particular set of classes involved and start over with a different hashing function if the first did not work.

## SUMMARY

We have demonstrated that it is possible for a linker to generate integer type IDs for classes such that it may be verified by a simple integer modulo computation that one class derives from another in an object-oriented language. When combined with a suitable way of adjusting offsets, this method provides for a fast, constant-time dynamic casting algorithm. A 64-bit type ID is capable of representing quite large class hierarchies containing thousands of classes and at least nine levels deep.

### ACKNOWLEDGEMENTS

We would like to thank Dr Gabriel Dos Reis, who made several constructive comments on earlier drafts of this paper.

### REFERENCES

1. Koenig A (ed.). *The C++ Standard* (2nd edn). IOC/IEC 14882:2003. Wiley: New York, 2004.
2. Stroustrup B. *The C++ Programming Language*. Addison-Wesley: Reading, MA, 2000.
3. Goldthwaite L (ed.). *Technical Report on C++ Performance*. ISO/IEC PDTR 18015, 2003.