SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 26(4), 453–465 (APRIL 1996)

On Efficiency and Optimization of C++ Programs*

PEI-CHI WU AND FENG-JIAN WANG
Department of Computer Science and Information Engineering, National Chiao Tung University, 1001 Ta-Hsueh Road, Hsinchu, Taiwan
(email; {pcwu, fjwang}@csie.nctu.edu.tw)

SUMMARY

The efficiency of object-oriented programs has become a point of great interest. One necessary factor for program efficiency is the optimization techniques involved. This paper presents the performance of several variations of a given C++ program and compares them with a version that uses no object-oriented features. Our result indicates that some object-oriented features in C++ are not well optimized in current C++ compilers. We thus discuss some code optimization techniques that can improve the efficiency based on the given C++ program.

KEY WORDS object-oriented programming; code optimization; C++ efficiency; C++ compiler optimizations

INTRODUCTION

The C++ language[1] is one of the most efficient object-oriented programming languages. A recent comparison[2] using a simple 'database' program showed that the C++ version is faster than those in the other four modern languages: Oberon-2, Modula-3, Sather, and Self. The designers of C++ have carefully considered the efficiency issue. For example, dynamic dispatch is not by default for the invocation of a member function in C++. In addition, C++ is strongly typed: it is easier to provide efficient (constant time) dynamic dispatch when the type information is available. The C[3] programs are thought to be efficient and compact. Do C++ programs inherit this property? Can we write C++ programs without concern for the underlying implementation of object-oriented constructs, even when the efficiency of programs is concerned?

Answering the above questions depends on the applications and compiler technologies of C++. It may be very difficult. This paper presents a case study to show several efficiency and optimization problems specific to C++. It measures the efficiency of some variations of a given C++ example program (*quick sort*) and compares them with a version using no object-oriented features. These variations include a class without inheritance, a class whose base class is empty, and a class whose base class is abstract. The experiment uses four compilers on PC and workstation platforms. Our results indicate that many current C++ compilers still do not produce well optimized codes for

* This research was supported in part by National Science Council, Taiwan, R.O.C., under Contract No. NSC 83-0408-E009-029.

CCC 0038–0644/96/040453–13

Received 13 June 1994
Revised 18 September 1995

given C++ programs. The class and inheritance features are straightforwardly supported and few optimization techniques are applied. We then discuss the following code optimization techniques: allocating a small object in registers, eliminating the space overhead in pure abstract classes, statically binding object values and arrays of object values, and removing offset adjustment in the dynamic dispatch of multiple inheritance. These techniques are easy to apply but are virtually ignored in the literature.

BACKGROUND

Object-oriented programming tries to bridge the gap between the real world and the information world. Class and inheritance are the important constructs in modeling real world objects. Some object-oriented languages, e.g., Smalltalk[4] and Self,[5] provide flexible dynamic binding and also bring in the efficiency problem. One major difficulty for code optimizations in object-oriented languages is that the type information is not available to compilers. For example, adding two integers in Smalltalk is represented by sending a '+' message with an integer parameter to an integer 'receiver'. In a Smalltalk interpreter, the corresponding procedure (*Integer + Integer → Integer*) is called with the object pointers (memory addresses) of the two integers as parameters, even if the operation to be invoked is a primitive method. On the other hand, most *optimizing* compilers translate this operation into one machine instruction and the operands are usually allocated in registers.

Some research work has been devoted to optimizing object-oriented programs. Typed Smalltalk[6] is a compiler for Smalltalk programs annotated with type declarations. It compiled a set of small examples and obtained a speed-up of 5 to 10 over a Smalltalk interpreter. The Self compiler[5,7] developed several optimization techniques. The first is *customization*,[5] which compiles several copies of a procedure, and makes the receiver type in each copy bound statically. The Self compiler allows functions inline, so many message passing and run-time type checking operations can be removed. Another technique is *polymorphic inline caches*.[7] It is a direct extension of the *inline cache*[8] used in Smalltalk. A polymorphic inline cache is a sequence of *if–then–else* statements to match the receiver type and then jump to the corresponding routine for a sending message. The code of a simple method can also be inlined in the cache. On the other hand, because determining exact type information is important in doing optimizations, some other work has been devoted to collecting type information from program profiles[9,10] or data flow analyses for object-oriented programs.[11,12]

The efficiency of a program (disregarding algorithmic issues) depends on (1) how much fine-tuning is applied by the programmers, and (2) how many and what optimizations are applied (by the compilers). The efficiency of a C program mainly depends on the former since most language features of C are close to the instructions of underlying machines. However, a recent work[13] on behavioral differences between C and C++ programs shows that C++ programs pose several challenges for compiler designers and computer architects. C++ not only inherits most of C's features but introduces some features that cannot be directly supported by underlying machines. The features such as *virtual functions* take several machine instructions and have more overhead than direct procedure calls. Reducing such overheads requires compiler's supports, so the efficiency of a C++ program depends more on optimization techniques than does that of a C program.

Some work has been devoted to optimizing C++ (and its compiler). Lea[14] proposed

an extension to C++ for applying customization. The keyword 'template' (originally used for 'generic' declarations in C++) was also used to declare a customized program. Calder and Grunwald[9] measured the effects of various optimization techniques on several large C++ programs. They minimized the cost of virtual functions by using the *branch-prediction* mechanism in modern computers, especially those of deeply pipelined architectures. In addition, there are two simple techniques, *unique name* and *if-conversion*, used in combination with branch prediction. The if-conversion technique is similar to inline cache. However, an if-statement may be slower than an indirect call in computers that have a branch penalty less than a deeply pipelined architecture does. The unique name technique replaces indirect function calls with direct calls when the linker detects that the virtual function has only one copy. The problem is that this optimization can only be done by using a 'smart' linker.

AN EXPERIMENT

Besides the features inherited from C, C++ introduces the following object-oriented constructs: *class, inheritance, template*, etc. Here we consider only class and inheritance, since templates do not cause (execution time) efficiency problems. The experiment is designed to compare the efficiency of the programs that use the class and inheritance constructs in C++.

The experiment tests a quick sort function template **qsort** (Figure 1). The first parameter of the **qsort** function is the array of data to be sorted, where the data type **T** is determined at instantiation. The experiment measures the sorting time for arrays of the following data types: **int, Int, vInt**, and **vvInt**. Type **int** is the built-in

```
template<class T>
void qsort(T a[], int m, int n)
{
  register int i=m;
  register int j=n+1;
  register T k = a[m];
  while (i < j) {
    i++;
    while (a[i] < k)  i++;
    j--;
    while (a[j] > k)  j--;
    if (i < j)  {
      /* interchange a[i], a[j] */
      register T tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;
    }
  }
  /* interchange a[m], a[j] */
  a[m] = a[j];
  a[j] = k;
  if (m < j-1) qsort(a, m, j-1); /* m..j-1 */
  if (j+1 < n) qsort(a, j+1, n); /* j+1..n-1 */
}
```

Figure 1. The function template **qsort** *(quick sort)*

```
class Int
{
  int v;
public:
  Int() { }
  Int(int i) { v=i; }
  operator int() { return v; }
  inline int operator<(const Int& b) const { return v < b.v;
  inline int operator>(const Int& b) const { return v > b.v;
};
```

Figure 2. Class **Int** *(a class with no inheritance)*

```
class Object { };
class vInt : public Object
{
public:
  int v;
  vInt() { }
  vInt(int i) { v=i; }
  operator int() { return v; }
  inline int operator<(const vInt& b) const { return v < b.v; }
  inline int operator>(const vInt& b) const { return v > b.v; }
};
```

Figure 3. Class **vInt** *(inheriting an empty class)*

integer type of C/C++. Class **Int** (Figure 2) is a user-defined class without inheritance. Class **vInt** (Figure 3) inherits an empty class **Object**. In some C++ libraries, e.g., NIHCL,[15] all classes inherit a base **Object** class. Class **vvInt** (Figure 4) inherits an abstract class called **Comparison**, which declares the comparison operators '⟨' and '⟩' used in a sorting routine. All these classes encapsulate a data member of **int** type and

```
template<class NumberT>
class Comparison
{
public:
  virtual int operator<(const NumberT&) const =0;
  virtual int operator>(const NumberT&) const =0;
};

class vvInt : public Comparison<vvInt>
{
public:
  int v;
  vvInt() { }
  vvInt(int i) { v=i; }
  operator int() { return v; }
  inline int operator<(const vvInt& b) const { return v < b.v; }
  inline int operator>(const vvInt& b) const { return v > b.v; }
};
```

Figure 4. Class **vvInt** *(inheriting an abstract class)*

```
template<class T>
void test(T& max, int size)
{
  int i;
  long t0, t1;
  T *data = new T[size+1];
  srandom(19930909);
  for(i=0; i<size; i++)
    data[i]= T(random());
  data[size]= max;
  t0=clock();
  qsort(data, 0, size-1); /* index 0..size-1 */
  t1=clock();
  printf("time (in u-sec): %d\n", t1-t0);
  delete data;
}
```

Figure 5. Function template **test**

provide comparison operations. They are designed using C++'s object-oriented programming features, including class, inheritance, abstract class, and template.

Figure 5 shows the function template **test**, which contains the test loop. The first parameter of **test (T& max)** is necessary because the type parameter (class **T**) must be used in the parameter list (Reference 1, p. 346). The array of data is generated by a random number generator. Figure 6 shows the main program. Each variation is executed with the same data.

These tests were performed on four C++ compilers: AT&T cfront (CC),[16] GNU C++ (G++),[17] Borland C++ (BCC),[18] and Microsoft Visual C++ (MSC).[19] They were executed on a workstation (SPARC station ELC) and a PC (i486, real mode). Since

```
#include <stdio.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/time.h>
extern "C" {
  long clock ();
  long random();
  void srandom(int seed);
}
main(int argc, char *args[]) {
  int size;
  if (argc != 2) {
     printf("usage: %s size\n", args[0]);
     return 0;
  }
  sscanf(args[1], "%d", &size);
  test(int(INT_MAX), size);
  test(Int(INT_MAX), size);
  test(vInt(INT_MAX), size);
  test(vvInt(INT_MAX), size);
}
```

Figure 6. Main program and included header

CC and MSC compilers do not support templates, we replaced the templates with simple macros of C preprocessor. The tests on G++ and CC compilers used an array of 200,000 elements, and the tests on BCC and MSC were reduced to 10,000 elements due to space limitations. Table I shows the execution time measured in seconds. The programs were run five times to calculate the average execution time. The optimization flag for CC and G++ was '-O'. We did not use '-O2', because applying '-O2' in GCC sometimes gets a less efficient code than '-O'. In the test on CC, a C++ program was translated into a C program (by the '+i' option) and then compiled by GNU CC (GCC)[20] with the option '-O'. To simplify the measurement, the clock rate of the PC was set low. The optimization flags were set both to generate i386 instructions and to optimize the speed. Since the generated random numbers in BCC and MSC are not the same, a random number generator was also provided during the test on a PC.

A DISCUSSION OF EFFICIENCY

The `qsort` program used in our test may not be general, neither can the arguments be applied in other programs. Thus, the conclusion drawn from the above performance result may not be applied to other C++ programs. However, our result does show the potential efficiency problem of C++ programs: *More object-orientation, more execution pains.*

Programming with class but no inheritance (i.e. *object-based*) can result in programs that run nearly as fast as equivalent C programs (e.g. class `Int`, 83 per cent speed in G++ and 93 per cent in CC). The use of inheritance takes 1/4 to 1/3 the speed (e.g. class `vvInt`, 26 per cent speed in G++ and 32 per cent in CC). Our result is comparable to some experiences in Reference 21. Replacing built-in data types such as integers and floats by user-defined classes may cause efficiency problems due to the lack of advanced optimization techniques.

In the following paragraphs, we discuss the result in more detail, mainly as concerns CC and G++. The time difference between vesions of `int` and `Int` is due to '`register T k`' in the `qsort` function template (Figure 1). G++ does not allocate a (small) object in registers. CC removes the optimization hint '`register`' from the translated C file (Figure 7), although GCC can allocate a small C structure (e.g. one word) in registers. GCC automatically allocates the variable `tmp` in a register but not for '`register Tk`'. When a hint '`register`' is put in the translated C file, GCC can also allocate variable `k` in a register.

The time difference between the `Int` and `vInt` versions comes from the inheritance of an empty class. A traditional C compiler treats an empty structure as a structure of at least one byte (and aligned on a word boundary in the SPARC architecture).

Table I. The execution times under four compilers of four versions of `qsort`

Program	G++ -O	CC + GCC -O	BCC -O2 -3	MSC -O -G3
`int`	1.81 s	1.79 s	0.59 s	0.39 s
`Int`	2.17 s	1.93 s	0.51 s	0.39 s
`vInt`	3.04 s	3.04 s	0.72 s	0.39 s
`vvInt`	6.85 s	5.69 s	1.65 s	1.27 s

```
char qsortInt__FP3IntiT2 (__0a , __0m , __0n )
struct Int __0a [];
int __0m ;
int __0n ;
{ register int __1i ;
  register int __1j ;
  struct Int __1k ;
  __1i = __0m ;
  ...
    if (__1i < __1j ){
      struct Int __3tmp ;
      __3tmp = (__0a [__1i ]);
      (__0a [__1i ])= __0a [__1j ];
      (__0a [__1j ])= __3tmp ;
    }
  ...
}
```

*Figure 7. The '**register**' hint for __ **1k** has been removed by CC in the translated C file for* **Int**

```
struct Object {    /* sizeof Object == 1 */
  char __W1__5Object ;
};
struct vInt {      /* sizeof vInt == 8 */
  char __W1__5Object ;
  int v__4vInt ;
};
```

Figure 8. The translated C structures for the classes **Object** *and* **vInt** *by CC*

Inheriting an empty class then introduces an additional data member of one byte (**char _ W1 _ 5Object**) in the **vInt** class (Figure 8). All the swap operations in **vInt** operate on two words instead of one. All these objects are allocated in memory. In addition, the temporary object **tmp** is short-lived, and allocating **tmp** in memory causes many memory accesses (Figure 9).

```
sll %o4,2,%o2
ld [%i0+%o2],%o3
sll %l0,2,%o1
ld [%i0+%o1],%o0
st %o0,[%i0+%o2]
st %o3,[%i0+%o1]

// tmp is allocated in %o3
```

```
sll %o3,3,%o2
ld [%i0+%o2],%o0
st %o0,[%fp-32]
add %i0,%o2,%o2
ld [%o2+4],%o0
st %o0,[%fp-28]
sll %l0,3,%o1
ld [%i0+%o1],%o0
st %o0,[%o2]
add %i0,%o1,%o1
ld [%o1+4],%o0
st %o0,[%o2+4]
ld [%fp-32],%o0
st %o0,[%o1]
ld [%fp-28],%o0
st %o0,[%o1+4]
// tmp is allocated in
//   [%fp-32] and [%fp-28]
```

Figure 9. The comparison on the swap operations for **Int** *(left) and* **vInt** *(right) of CC*

The time difference between the **vInt** and **vvInt** versions comes from the dynamic dispatch in class **vvInt**. The class **vvInt** has one additional data field (one word) for storing a pointer to its virtual function table. All comparison operations in **vvInt** are dynamically dispatched (indirect procedure calls). These dispatches double the execution time since the comparison operations are critical (inside the inner loop, Figure 10) in **qsort**.

The comparisons between G++ and CC are also interesting: the combination of CC and GCC produces a more efficient **qsort** program than G++. CC is a translator, and we used CC with GCC in the experiment. G++ and GCC share the same back-end and have the same capability in code optimization. G++ is a native C++ compiler and may do more work in optimizing C++ programs. Our result indicates that G++ does not. It shows that much of the translation work for object-oriented features in G++ is done straightforwardly: translating C++ constructs to some C constructs and then calling the code generation routines for C. Our result reflects that a translator is not substantially different from a native compiler in terms of the output object code, but a translator may run much slower than a native compiler.

The above discussion has focused on the results from CC and G++ tests. The results from BCC and MSC are slightly different. First, BCC and MSC are both able to allocate a register for '**register T k**' in the while-loops for comparisons, but fail to allocate '**register T tmp**' in a register for the versions of **int** and **Int**. The execution times for the **int** and **Int** versions are thus almost equal on BCC and MSC. Secondly, for the layout of **vInt**, MSC does not introduce the additional field, so the speeds for the **vInt** and **Int** versions are the same. BCC does not align **vInt** with a word boundary, so an additional byte copy, instead of a word copy, is needed for swapping two **vInt** objects. Thirdly, for **vvInt**, both BCC and MSC are able to generate dynamic dispatch code of single inheritance.

SOME C++ CODE OPTIMIZATION TECHNIQUES

Current C++ compilers have used many optimization techniques, e.g. function inline, statically binding member functions, and virtual function tables for dynamic dispatch.

```
L30:
    sll %o4,2,%o0
    ld [%i0+%o0],%o0
    cmp %o0,%o2
    bl,a L30
    add %o4,1,%o4

/*
%i0 is address of a[].
%o4 is i.
%o0 is address of a[i].
%o2 is value of k.
*/
```

```
    add %l0,1,%l0
L69:
    sll %l0,3,%o1     ! %l0 is i.
    add %i0,%o1,%o1   ! %i0 is a[].
    ld [%o1],%o2      ! %o1 is a[i].
*   ldsh [%o2+8],%o0  ! %o2 is the
*   add %o1,%o0,%o0   ! pointer to vtable.
    ld [%o2+12],%o2
    call %o2,0        ! call operator<
    add %fp,-24,%o1
    cmp %o0,0         ! the result in %o0
    bne,a L69
    add %l0,1,%l0
/* [%o2+8] is the "delta" field in vtable.
[%o2+12] is the function pointer to
operator <. */
```

Figure 10. The inner loop '`while (a[i]<k) i++;`*' for* **vInt** *(left) and* **vvInt** *of CC*

Our result indicates that there is still a need for more optimization techniques. Here we enumerate some code optimization techniques for further improvement of efficiency. All these techniques can be applied in the 'separate' compiling tradition of C/C++.

Allocating a small object in registers

Most modern computers have many registers. How these resources are used may affect the program efficiency. Many basic data structures, e.g. a node of a linked list, are very small. A temporary object of such data structures can be allocated in registers. However, it is not so easy in practice. First, the semantics of C++ constructors are defined as 'a constructor turns raw memory into an object . . .' (Reference 1, p. 262). Is an address of raw memory necessary when constructing an object? As addressed in the previous section, GCC allocates a small object (a C structure) in a register. Compilers may have more choices when using a constructor. Secondly, objects are usually passed by *references*, i.e. memory addresses. It is impossible to give a 'memory address' for a set of registers. Directly adding a '`register`' hint to object variables does not work. Figure 11 shows the warning messages when adding '`register`' to the variables `k` and `tmp` in the class `vvInt` on the ouput C file. The hint is ignored since the addresses of `k` and `tmp` are needed to call the comparison function.

For a function with parameters of object references, two separate versions of the function code are needed for passing objects, one by registers and the other by memory addresses. For example, C++ compilers may need to provide two copies of a default constructor and a copy constructor, which are automatically generated. This optimization may thus increase code size. However, allocating an object in registers works fine for an inline function. It does not increase the code size, because the inline function is already expanded in callers. In addition, an inline function may be more efficient if the objects involved are allocated in registers.

Eliminating space overhead in pure abstract classes

Abstract classes are useful in object modeling. A *pure* abstract class is an abstract class with no real implementation. The inheritance of a pure abstract class is not for *implementation* but for *interface*. A pure abstract class may be mapped to an empty structure of C. Unfortunately, the size of a structure in C must be greater than zero, so an empty structure contains at least one field of `char`. Inheriting a pure abstract class may thus inherit a useless field and cause overhead in object copying. This space overhead can be easily removed by compilers, if pure abstract classes are handled directly by using specific code generation routines.

Removing offset adjustment in dynamic dispatch of multiple inheritance

Although dynamic dispatch sometimes causes inefficiency, dynamic dispatch is useful and flexible in programming. The type of an object may not be statically determined,

```
qsort..c: In function `qsortvvInt__FP5vvIntiT2':
qsort..c:295: warning: address of register variable `__0k' requested
qsort..c:323: warning: address of register variable `__2tmp' requested
```

Figure 11. The messages in adding '`register`' to the variables `k` and `tmp` for the class `vvInt`

so it is impossible to completely eliminate function calls of dynamic dispatch. Such a kind of inefficiency can be overcome with more optimization techniques.

The designers of the C++ language carefully considered the implementation of virtual functions. The solution is a virtual function table (*vtable*) (Reference 1, Section 10.8.1c). For single inheritance, support of dynamic dispatching requires an indirect procedure call. For multiple inheritance, in addition to an indirect call, two operations, a load of offset, and an add operation are needed in most implementations.

Because the program `qsort` uses only single inheritance, we can simply remove the code for offset adjustment. Table II shows the speed-up by removing offset adjustment in dynamic dispatch of multiple inheritance in G++ and CC. The asterisked instructions in Figure 10 are removed, and the register '`%o1`' is changed to '`%o0`'. The execution time reduction is nearly the same for G++ and CC. The speed-up is 6.9 per cent for G++ and 10.5 per cent for CC. The code generated by CC is more compact than that generated by G++, so the execution time reduced with CC thus looks more significant. Note that BCC and MSC have already applied this optimization.

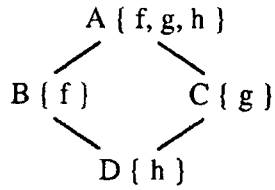
There are three means of applying this optimization. Stroustrup (Reference 22, p. 265) presented an alternative implementation for dynamic dispatch of multiple inheritance. In his approach, a small piece of code is used to adjust the `this` pointer and jump to the corresponding member function. There is no need to store the offset in a virtual function table, no code duplication, and no execution overhead for dynamic dispatch of single inheritance. This implementation technique looks good, but is less portable.

Borland C++ version 3.1[23] (*BCC3* in short, the older version of BCC) restricts the use of multiple inheritance and provides more efficient dynamic dispatching. BCC3 disallows an inheritance with interactions of sibling classes (e.g., the inheritance considered in Reference 1, p. 234). Some code rewriting is needed when the inheritance violates the restriction. Figure 12 shows an example of interactions between sibling classes. Class `C` defines the function `g()` by calling the function `f()`, which is defined in class `B`. The example works in G++, CC, MSC, and BCC (the newer version), but not in BCC3.

Another approach is to duplicate the code inherited from base classes when the interactions of sibling classes happen in an inheritance hierarchy. Considering the above example, Figure 13 shows an equivalent class hierarchy without interactions. The function `f()` is defined once again in class `D`. There is an additional copy of `f` specific for objects of `D`, so the offset adjustment for converting an object of `D` to that of `B` is not needed when calling `B::f`. The equivalent class hierarchy can be automatically generated by compilers. The code duplicated (e.g., function `f()`) may be small, if the inheritance hierarchy is very simple.

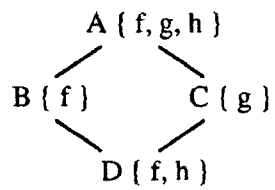
Table II. The speed-up in replacing multiple inheritance by single inheritance

Program	G++	CC
Multiple inheritance (1)	6.85 s	5.69 s
Single inheritance (2)	6.41 s	5.15 s
Speed-up (1)/(2) − 100%	6.9%	10.5%




```
class A {
public:
  virtual void f();
  virtual void g();
  virtual void h();
};
class B : public virtual A {
public:
  void f() {}
};
class C : public virtual A {
public:
  void g() { f(); }
};
class D : public virtual B, public virtual C
{
public:
  void h() {}
};
```

Figure 12. The class hierarchy and an example showing interaction between sibling classes `B` *and* `C`




```
class A { ... };
class B : public virtual A { ... };
class C : public virtual A { ... };
class D : public virtual B, public virtual C
{
public:
  void h() {}
  void f() {}
};
```

Figure 13. Equivalent class hierarchy without interactions in sibling classes

Binding object values and arrays of object values statically

C++ supports polymorphism on object *references* but not on object *values*. Object values and array of object values have *exact* types. They are in wide use, e.g. '`T a[]`' in `qsort`. Let `T` be a class. A declaration of '`T t;`' declares an object (value) `t` of type `T`. A declaration '`T a[];`' declares `a` as an array of object values of class `T`. In contrast, the declaration '`T &b;`' (or '`T *C;`') declares `b (c)` as a reference (pointer) to any derived class of `T`.

For an object value or an array of object values, no dynamic dispatch is needed when executing their member functions. Although 'objects of a derived class can be assigned to objects of a public base class' (Reference 1, p. 297), the copy operations do not change the structure (the type) of either object (Reference 1, p. 298). Binding such objects statically is very simple and can greatly improve the efficiency. This is true since no dynamic dispatch is needed and function inline is applicable. In addition, the vtable pointers of an array of object values are the same. This may be the invariant in a loop that processes an array of object values. Table III shows the speed-up that results from using static binding for the array of integers `a []` in `qsort` (compared with the version of single inheritance). Note that the single inheritance versions of BCC and

Table III. The speed-up in using static binding for an array of objects

Program	G++	CC	BCC	MSC
Single inheritance (1)	6.41 s	5.15 s	1.65 s	1.27 s
Static binding (2)	5.75 s	4.54 s	1.44 s	1.13 s
Speed-up (1)/(2) − 100%	11.5%	13.4%	14.6%	12.4%

Table IV. The availability of the optimization techniques in the four compilers

Optimization	G++	CC	BCC	MSC
Allocating a small object in registers	–	–	–	–
Eliminating pure abstract classes	–	–	–	✓
Replacing multiple inheritance by single inheritance	–	–	✓	✓
Binding arrays of object values statically	–	–	–	–

MSC are the versions of `vvInt`. The speed-ups in the four compilers are all more than 10 per cent.

The means of treating arrays in C may forbid this optimization. For example, '`char []`' in C is defined to be equivalent to '`char*`'. However, the declaration '`T d []`;' is not equivalent to '`T *e`;', since `d` has an exact type `T`, and the type of `e` is a pointer to any derived class of `T`. The type information is lost when translating an array to a pointer. It may not be easily recovered by optimizations in the back-end of a compiler.

Table IV summarizes the availability of these optimization techniques in the four compilers tested. None of these compilers fully support the allocation of an object in registers, and none of them apply static binding of arrays of object values. The table indicates the needs for more optimizations.

CONCLUSIONS

Our experiment shows some potential inefficiency of object-oriented programs using C++ language. One major factor is the lack of code optimizations for the object-oriented features of the C++ language. We have presented several C++ code optimization techniques. Although the improvements have only been measured in the `qsort` example, these techniques may work well for small programs that intensively use object-oriented features. The speed-up may be less significant when they are applied to real C++ applications, which usually involve bulks of C codes.

Most C++ compilers are extended from existing C compilers. Many of their development efforts have been devoted to *correctly* implementing new features, especially *multiple* inheritance and templates. In the future, more research will be needed to improve these features too.

REFERENCES

1. M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.
2. R. Henderson and B. Zorn, 'A comparison of object-oriented programming in four modern languages', *Software—Practice and Experience*, **24**, 1077–1095 (1994).

3. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd edn., Prentice Hall, 1988.
4. A. Goldberg and D. Robson, *Smalltalk-80: The Language*, Addison-Wesley, 1989.
5. C. Chambers and D. Unger, 'Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language', *Proc. ACM SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI'89)*, also as *ACM SIGPLAN Notices*, **24**, (7), 146–160 (1989).
6. R. E. Johnson, J. O. Graver and L. W. Zurawski, 'TS: an optimizing compiler for Smalltalk', *Proc. OOPSLA'88*, pp. 18–26.
7. U. Hölzle, C. Chambers and D. Unger, 'Optimizing dynamically-typed object-oriented languages with polymorphic inline caches', *ECOOP'91 European Conference on Object-Oriented Programming*, LNCS, Vol. 512, Springer-Verlag, July 1991, pp. 21–38.
8. L. P. Deutsch and A. M. Schiffman, 'Efficient implementation of the Smalltalk-80 system', *Proc. 11th ACM Symp. on the Principles of Programming Languages*, pp. 297–302 (1984).
9. B. Calder and D. Grunwald, 'Reducing indirect function call overhead in C++ programs', *Proc. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'94)*, pp. 397–408 (1994).
10. U. Hölzle and D. Unger, 'Optimizing dynamically-dispatched calls with run-time type feedback', *Proc. ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI'94)*, also as *ACM SIGPLAN Notices*, **29**, (6), 326–336 (1994).
11. M. Südholt and C. Steigner, 'On interprocedural data flow analysis for object oriented languages', *Proc. 4th Int. Conf. Compiler Construction (CC'92)*, LNCS 641, Springer-Verlag, 1992, pp. 156–162.
12. J. Vitek, N. Horspool and J. S. Uhl, 'Compile-time analysis of object-oriented programs', *Proc. 4th Int. Conf. Compiler Construction (CC'92)*, LNCS 641, Springer-Verlag, 1992, pp. 236–250.
13. B. Calder, D. Grunwald and B. Zorn, 'Quantifying behavioral differences between C and C++ programs', *Technical Report CU-CS'698-94*, Department of Computer Science, University of Colorado at Boulder, U.S.A., January 1994.
14. D. Lea, 'Customization in C++', *Proc. 1990 USENIX C++ Conference*, San Francisco, California, April 9–11, 1990, pp. 301–314.
15. K. E. Gorlen, *NIH Class Library*, Revision 3.0, May 1990.
16. AT&T, *C++ Translator*, Version 2.1.03, 1990.
17. Free Software Foundation, *GNU C++ Compiler*, Version 2.5.7, 1993.
18. Borland International Inc, *Borland C++*, Version 4.0, 1993.
19. Microsoft Corp, *Microsoft Visual C++*, Version 1.5, 1993.
20. Free Software Foundation, *GNU C Compiler*, Version 2.5.7, 1993.
21. J. M. Coggins, 'Speed of C++ vs C: Myths, Data, and Skepticism', *C++ Report*, Jan 1993, pp. 25–27.
22. B. Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994.
23. Borland International Inc, *Borland C++*, Version 3.1, 1992.