The following paper was originally published in the
Proceedings of the USENIX 1996
Conference on Object-Oriented Technologies
Toronto, Ontario, Canada, June 1996.

Compiler Optimization of C++ Virtual Function Calls

Sara Porat, David Bernstein, Yaroslav Fedorov,
Joseph Rodrigue, and Eran Yahav
IBM Haifa Research Laboratory

Compiler Optimization of C++ Virtual Function Calls

Sara Porat
David Bernstein
Yaroslav Fedorov
Joseph Rodrigue
Eran Yahav
VNET: porat at haifasc3
E-mail: porat@haifasc3.vnet.ibm.com
Tel: (972) 4 8296309
FAX: (972) 4 8296114
IBM Haifa Research Laboratory
Matam, Haifa 31905, Israel

Abstract

We describe two generic optimization techniques to improve run-time performance of C++ virtual function calls: *type specification* and *type prediction*. Both involve program analysis that results in a set of call sites to be optimized, and code transformations that replace the original dispatching mechanism in these sites by more efficient call expressions. We implement two special cases. The first is a type-specification optimization, called *unique name*, that requires static global view of the whole program in order to substitute indirect virtual function call sites by direct calls. The other is a type-prediction kind of optimization, referred to as *single type prediction*, that uses type-profiling information to replace virtual function calls by conditional expressions which involve direct function calls.

These optimizations were implemented in IBM's compiler, the C Set ++ for AIX/6000, and evaluated on a set of C++ standard benchmarks. We received encouraging run-time results with improvements in order of 20% that demonstrate the vitality of these techniques.

1 Introduction

One of the most powerful concepts in C++ programming is *polymorphism* as provided by virtual functions. This is a powerful abstraction mechanism, but it carries on a significant run-time penalty. A virtual function call is usually implemented as an indirect function call through a table of functions generated by the compiler, so called Virtual Function Table or VFT. This indeed ensures reasonable efficiency for such calls, but it has by no doubt certain run-time overhead as compared to direct function calls. Moreover, and possibly even more important with respect to performance criteria, is the fact that a compiler cannot apply inline substitution to unqualified virtual function calls. Such inline substitution could have totally eliminate the calling overhead, and enable the compiler to further apply standard optimizations on inlined bodies of virtual functions.

In this study, we analyze and develop techniques to optimize virtual function calls in C++. We were motivated by other studies ([6]) that convince us that performance is most critical for C++, and that efficiency problems can become a key inhibitor to widespread acceptance of this language.

The subject of our work is replacing indirect virtual function call expressions in C++ by more effective expressions that include direct calls to virtual functions. In some cases, a static analysis of the whole program [4] can determine for a particular virtual function call site that there is a *unique* implementation of this function that can possibly be called at that point. In such case, the virtual function call can be replaced by a direct call. Even when the particular virtual function call is not always resolved to a single implementation, we may explore type-feedback information [10] that indicates that there is a particular implementation that is resolved most of the time. Using this information, an indirect virtual function call can be replaced by a *conditional* (type-predicted) expression, so that the likely invoked function is called directly and others are invoked through the regular indirect mechanism.

The topic of virtual function call optimization has been of great interest in the last years. Some papers ([8], [11]) just report dynamic program characteristics of C++ programs and show that virtual calls are very common in C++ programming style. These studies are not proposing solutions, but provide the basic motivation for subsequent research works. Others, like [7], propose techniques for optimizing virtual function calls, e.g. by transforming them to conditional expressions, and analyze the effects of such optimizations, but do not provide concrete implementations. An interesting comparison of optimization techniques in the context of SELF can be found in [1]. In contrast, we describe *compiler-based implementations* that *find* opportunities for optimizations, and consequently *apply* code transformations.

Many papers are closely related to our work and introduce similar implementations to optimize virtual function calls. We cite only a subset of all these works, and primarily reveal the differences between them and ours. Other interesting studies can be found in those papers that we do reference.

Algorithms that determine for virtual function call sites whether or not they are resolved to a unique implementation can be found in [3], [4] and [12]. These papers focus on whole-program analysis of C++ code but none of them involve compiler implementations to generate optimized virtual calls.

Run-time type feedback is used in [10] and [9] to optimize dynamically-dispatched calls in SELF and Cecil. This technique is also examined in [9] in the context of C++, but it is limited to the stage of code analysis only, i.e. no transformations are applied to the code.

A very recent paper by Aigner and Holzle ([2]) (that was published after a draft of our paper was already submitted for publication) seems to be the most related work to that presented here. The main distinction between their approach and ours is that they use a source-to source process to compile a *baseline all-in-one* C++ source into C++ optimized code, whereas we change an *existing production compiler* that compiles original source code and generates optimized code. Our solution is much more in the spirit of other compiler optimization techniques, and seems to be much more efficient than that proposed in [2]. To conclude, our work pioneers the use of an existing C++ compiler to close the whole loop of optimizing C++ virtual function calls: performing program analysis, evaluating the results of the analysis, feeding back the results to the compiler, and generating optimized virtual function calls given this information.

A different approach for optimizing virtual function calls is introduced through *subtype cloning*. This technique removes dynamic dispatches resulting from parametric polymorphism through code replication. A recent work [13] presents an efficient cloning algorithm, and a comprehensive summary of related work.

Our development environment consists of IBM's production compiler, the C Set ++ for AIX/6000 ([14]). Here the compilation process consists of several code processing phases: front-end processing, subsequent function inlining processing, the invocation of which is controlled by the user, and which attempts to inline functions that are marked as inline, and finally back-end processing. Since the inliner is invoked right after the front-end, the idea is to let the front-end detect and exploit the possibilities for virtual function call conversions, so as to allow subsequent inlining of transformed virtual function calls. In addition to that, the back-end is not capable of naturally detecting virtual function calls and optimizing them, since its intermediate language lacks the high-level concepts of virtual functions and VFTs. Thus, our goal is to implement a variation of the C++ compiler front-end that will automatically replace virtual function calls when possible and when such replacement will reduce the path-length. We strongly believe that such optimization can play a major role in optimizing C++ programs, since it reduces the calling overhead and allows further back-end optimization as a result of inlining.

We apply our modified C++ compiler to several C++ programs, and the results so far are very encouraging. Of special interest are those programs that make extensive use of virtual functions. One of these is Tal-Dict, a performance test program of the Taligent dictionary class, where our optimization succeeds to improve the execution time by 20%.

The remainder of this paper is organized as follows. In the following section we describe our optimization techniques in general terms of type specification and type prediction. In Section 3 we illustrate our optimizations with a small example program. In Section 4 we give more details on our implementation work with the unique-name optimization and the single-type-prediction optimization. Next, in Section 5 we evaluate our techniques and present the benchmarks that we used to experiment our optimizations. Finally, in Section 6 we discuss our conclusions and plans for future work.

2 Optimization via Analysis and Transformation

In this section we will clarify the key-concepts in our optimizations, without getting into specific implementation details.

Our optimizations comprise two phases, *analysis* and *transformation*. In the first, we collect information about the program to result in a set of *candidate* unqualified virtual function call (VFC) sites. Each such candidate identifies an exact source location (file name, line and column) of a particular call expression that invokes a virtual function through the VFT mechanism. A candidate VFC also introduces a set of *specified* or *predicted* class names. In the second phase, we use the candidate set to apply transformations that try to improve performance. Each candidate VFC is optimized using its corresponding set of class names.

2.1 Type Specification and Type Prediction

Type-specification optimization is aimed at transforming a set of candidate VFC sites. Each candidate VFC is associated with a list of *specified* class names. Each of these class names defines a possible target function implementation that may be invoked in this call expression. The whole list associated with a particular candidate VFC represents *all* possible target functions to which this call can be resolved. In the transformation phase we look up the list of n target functions corresponding to each candidate call site, and replace the indirect call with an n-way branch expression to direct calls to the target functions.

In the analysis phase of type-specification optimization we need to traverse the complete class hierarchy. Moreover, determining for a particular VFC site the precise set of target functions is NP-Hard ([12]). Each approximation results in pessimistic sets, i.e. some target function implementations that are listed for a particular VFC site may in fact never be invoked. Another important consideration is the cost of an n-way branch, for $n > 1$. In most architectures such branch is implemented through a complex conditional expression that depends on nested comparisons, each of which compares the type of the object, against which the call is currently invoked, with some specified type. Usually type-specification transformation should only be performed if n is smaller than some threshold value, depending on the cost of conditional branches on the target machine and the effectiveness of the optimizer. These considerations may limit the size of the candidate set in type-specification optimization, and hence its applicability for some programs.

Type-prediction optimization is aimed at transforming a larger set of candidate VFC sites than in the case of type specification. Here we identify for each VFC site a set of m *predicted* class names. Each predicted class name defines a particular predicted function implementation, that may be invoked in this call expression, and probably more often than other implementations. In the transformation phase we replace the unqualified call with an $m + 1$-way branch, so that the predicted functions are called directly, and all the rest are invoked through the regular dispatching mechanism.

In this research, we present only two special cases of these generic optimizations. *Unique name* (UN) is a special case of type specification, where the transformation is done only if the set of possible target functions is a singleton, i.e. $n = 1$. In *single type prediction* (STP) we identify, for each candidate VFC site, a unique predicted target, i.e. $m = 1$. Our chosen predicted function is the one that is called most of the time.

2.2 Unique Name

The analysis phase for this optimization consists of finding VFC sites for which only one destination exists. For each UN VFC candidate, a unique class name is identified.

There are many methods suggested in the literature to find a UN candidate set, varying from simple algorithms ([3] [7]) up to approximations using class-hierarchy pruning ([4]) and complex data flow analysis ([12]).

We implement a simple algorithm that explores information on VFC sites and knowledge on the complete class-hierarchy. For each VFC, we find all of the *possible methods* that can be called at that point. The basic algorithm that we used is referred in [3] as *simple call graph construction*. For more implementation details see Section 4. The set of UN candidates can be obtained using any program-database or compiler based tool. In any case, the analysis phase requires a global view of the whole program.

In the transformation phase, we chose to visit all the candidate VFC sites found, and replace the indirect function call expression by a direct call expression to the corresponding destination.

An alternative way to explore the fact that some virtual function has a unique implementation over some class hierarchy is by *nullifying its virtuality*, i.e. modify the characterization of this function as a virtual function, and let it be non-virtual. This approach can be easily implemented within a compiler front-end. From a performance perspective, the affects of such a transformation are similar to those gained by transforming UN candidates and get them be direct calls. On the other hand, removing the virtual characterization has the advantage of reducing code size by removing entries in VFTs. As far as we know, this technique hasn't been yet implemented and evaluated.

2.3 Single Type Prediction

STP optimization is a special case of type-prediction optimization in which we identify for each VFC site a single target function that is called most of the time. In this case, we replace the indirect call by a conditional expression so that the likely invoked function is called directly and others through the regular indirect call mechanism.

We chose to find STP candidates using off-line profiling information, as it is done in [2]. (In contrast, the SELF system ([10]) uses on-line profiling information.) We implement a tool that provides information about how often different target functions are invoked when the program is running. Our tool shows for each VFC site how many times each possible method was invoked over a series of executions of the program. Every VFC that is invoked in these executions serves as an STP candidate, and its associated predicted class name is the one that defines the target function that was invoked more (or at least not less) than each other target function.

Note that profiling analysis requires running the program on a series of inputs. For those applications where profiling is too expensive, STP candidates may be somehow provided by the programmer.

To implement our profiling tool, we use a mechanism similar to the Test Coverage Tool in C Set ++ for AIX/6000. We implement a variation of the compiler that instruments every compilation unit of the program to facilitate collection of type-profiling information about VFC sites.

In the STP transformation phase we visit all candidate VFC sites, and replace the indirect function call expression <indirect-call> by a conditional expression of the form <exp> ? <direct-call> : <indirect-call>, where <exp> is evaluated to TRUE if we reach the VFC site with a *predicted-type* object. When reaching the site with an object of another type, the original indirect call is taken. The direct call in the expression invokes the predicted target function. In what follows, we refer to <exp> as the *type-info* expression.

When we have a *hit* (an object of the predicted type reaches the call site), we only pay the overhead of evaluating the type-info expression and the cost of a direct call, saving the cost of an indirect call. When we have a *miss*, though, we pay the cost of the original indirect call and the overhead of evaluating the type-info expression.

The decision whether to transform a call can be based on some heuristics. Unlike the UN optimization, STP involves a penalty (in case of a *miss*). A heuristic can be used to check that the hit-rate is high enough to ensure that we do not pay additional cost over the original indirect call cost (due to *miss* penalties) too often.

2.4 Combining Unique Name and Single Type Prediction

The combination of both UN and STP, denoted by UN&STP, refers to a combined transformation phase, where each UN candidate is transformed to a direct call, and each STP candidate which is not a UN candidate is transformed to a conditional expression.

3 Example and Supporting Results

We demonstrate our ideas on a simple C++ program. The program consists of four files - two *.h* files and two *.C* files:

```
//A.h
class A {
public:
 int data;
 virtual void init();
 virtual void inc() { ++data; }
 virtual void dec() { --data; }
};

extern void foo1(A*);
extern void foo2(A*);
```

```
//B.h
#include "A.h"

class B: public A {
public:
 virtual void dec() {data-=2;}
};
```

```
//A.C
#include "A.h"

void A::init() { data = 0; }
```

```
5
6  void foo1(A* ptrA) {
7   ptrA->A::inc();
8   for(int i=0;i<50000000;i++)
9     ptrA->inc();
10 }
11
12 void foo2(A* ptrA) {
13   ptrA->dec();
14 }
```

```
1  //B.C
2  #include "B.h"
3
4  void main() {
5   A* ptrA = new A,*ptrB = new B;
6   ptrA->init();
7   ptrB->init();
8   foo1(ptrA);
9   foo2(ptrB);
10  for(int j=0;j<50000000;j++)
11    ptrB->dec();
12 }
```

The example program introduces five (indirect) virtual function call sites, two in lines 9 and 13 of *A.C* and the other three in lines 6, 7 and 11 of *B.C*. These are the only calls that use the VFT mechanism. In contrast, the call site in line 7 of *A.C* is a direct virtual function call (fully qualified call).

By examining the assembly code as generated by the C Set ++ compiler for AIX/6000, we can see the difference between the direct and indirect virtual calls. We present here unoptimized code, i.e. code compiled without the -O option.

Direct call in line 7 of *A.C*:

```
 7| L4Z       gr3=ptrA(gr1,88)
 7| CALL      inc__1AFv,1,gr3...
```

The direct call is translated to two instructions:

1. Load the *this* pointer value.
2. Call function.

Indirect call in line 9 of *A.C*:

```
 9| L4Z       gr3=ptrA(gr1,88)
 9| L4Z       gr5=(*A).__vfp(gr3,0)
 9| L4A       gr4=(*struct {...}).
                    __tdisp(gr5,20)
 9| L4Z       gr11=(*struct {...}).
                    __faddr(gr5,16)
 9| A         gr3=gr3,gr4
 9| CALL      gr11,1,gr3,
               __func__ptr__0...
```

The indirect call is translated here to six instructions:

1. Load the *this* pointer value.
2. Load the address of the appropriate VFT, using the *__vfp* field.
3. Load the *this* pointer adjustment value from the appropriate entry in the VFT.
4. Load the virtual function address from the VFT.
5. Adjust the *this* pointer.
6. Call through pointer to function.

3.1 Unique Name

The UN optimization finds and transforms VFC sites, each of which has a unique implementation. In our example, *A::inc()* is the only implementation of *inc()*, and therefore our tool recognizes the call in line 9 of *A.C* as a UN candidate which is always resolved to *A::inc()*. In order to transform this call to a direct call, we feed our compiler with information on that candidate as follows

A.C, line 9, column 14: A

This indicates the candidate's location and the class in which the single implementation is defined. The compiler front-end then translates the candidate call to a direct call as it does for the call in line 7.

Note that for this UN VFC candidate, the single implementation is defined in *A*, and the call is invoked using a pointer to *A* (*ptrA*). Thus, there is no need to adjust the *this* pointer that points to the beginning of an *A* object. In general, when an adjustment is needed (e.g. when the pointer points to an object of a virtual base class, and the predicted type is a subclass) we save less instructions per transformed location.

Our UN optimization optimizes three calls in the example, and leaves the calls to *dec()* untouched.

3.2 Single Type Prediction

The STP optimization finds VFC sites that are accessible during program execution, and transforms these calls using profiling information. Our STP analysis phase recognizes all five VFC sites in the example as STP candidates, and for each of them the profiling data indicates that all calls are resolved to the same function. For example, all calls in line 11 of *B.C* use *B::dec()*.

For a VFC site, the actual implementation being called depends on the class of the pointed object. Evaluating the *type-info* expression in the optimized VFC should effectively fetch the type of an object at run-time. Since our compiler does not support RTTI [1] (at least at the moment), we use the address of the VFT to which the object points as an indication for its type.

In our example, we feed the STP optimizing front-end compiler with information on the STP VFC candidate as follows

B.C, line 11, column 14: __vft1B1A

This indicates the candidate's location and the VFT which is used most of the times in this VFC. The compiler front-end then generates optimized code, and instead of the original six instructions it translates the candidate call to:

```
11| L4Z      gr3=ptrB(gr1,64)
11| L4Z      gr3=(*B).__vfp(gr3,0)
11| LR       gr4=gr31
11| CL4      cr1=gr3,gr4
11| BT       CL.13,cr1,0x4/eq
11| L4Z      gr3=ptrB(gr1,64)
11| L4Z      gr5=(*B).__vfp(gr3,0)
11| L4A      gr4=(*struct {...}).
                  __tdisp(gr5,28)
11| L4Z      gr11=(*struct {...}).
                   __faddr(gr5,24)
11| A        gr3=gr3,gr4
11| CALL     gr11,1,gr3,
              __func__ptr__2...
11| CL.13:
11| L4Z      gr3=ptrB(gr1,64)
11| CALL     dec__1BFv,1,gr3...
```

The first four lines evaluate the *type-info* expression by comparing the value of $this \rightarrow \_\_vfp$, the VFT which is pointed by our object, to the address of *__vft1B1A*, the VFT that is used to invoke the predicted target function. Execution will then either branch to the two instructions that stand for the direct call, or else use the six instructions of the indirect mechanism.

[1] run-time type information

3.3 Run-Time Performance Comparison

Finally we illustrate our techniques by comparing execution times of our example program, when compiled with each of our proposed optimizations. In addition, we will illustrate the results when function inlining is enabled, versus those when function inlining is disabled. This will clarify the gain achieved by enabling inline substitutions to virtual function calls, and applying further optimizations. Table 1 shows the execution times for the example program on IBM RISC System/6000, Model 390 (POWER2, 66MHz). Times are measured in seconds. The leftmost column shows the compiler options, where -Q / -Q! mean inlining enabled / disabled, and -O means back-end compiler optimizations enabled. The UN and STP columns presents run times when UN or STP (respectively) optimization is applied on the program. The UN&STP column is the result when first applying UN, and then applying STP where UN cannot be applied.

Options	Original	UN	STP	UN&STP
-O -Q!	25.7	20.3	16.8	15.9
-O -Q	25.6	15.1	13.7	9.1

Table 1: Example run-time measurements

As can be easily seen, when applying both optimizations and inlining, execution time drops from *25.6* to *9.1*. Our example is an extreme case in which the transformation gives room for other global optimizations. The small methods, when inlined and globally optimized, turn into a mere set of few instructions, resulting in a very effective optimized version of code.

4 Implementation

This section describes our implementation work. As stated in Section 2, we develop four independent components:

1. UN analysis component
2. UN transformation component
3. STP analysis component
4. STP transformation component

We also implement a UN&STP transformation component.

The UN (or STP) analysis component generates a list of candidates. This list is the input for the UN (or STP, respectively) transformation component.

The two analysis components are very different. While the UN analysis component requires a static view of the whole program, the STP analysis component requires a dynamic view. The first is achieved by compiling the whole program once, while the second involves executions of an instrumented program in order to collect profiling information.

In contrast, the UN transformation component and the STP transformation component are similar to each other. In both we replace VFC sites by other kinds of expressions offering performance improvement.

Except for the UN analysis component, all other components are in fact variations of the compiler that either instrument the original code, or optimize it. We decided to generate variations of the compiler front-end. Modifying the front-end turns out to be a very comfortable and efficient way to achieve our goals. The UN analysis phase can be accomplished using tools other than the compiler. Even though, our implementation involves yet another front-end variation.

4.1 The Unique Name Analysis Component

We implement a basic algorithm that finds UN VFC candidates. Our implementation does not use any data flow analysis nor class-hierarchy pruning. Thus, we find a subset of all UN candidate call sites. Using a more sophisticated data-collection phase for this optimization may lead to results that are better than the results presented in this paper.

Generating this list is done using a matching-graph. The matching-graph is a bi-partite graph with two kinds of nodes - VFC site nodes, and method implementation nodes. Edges in the graph connect each VFC site to all possible methods that can be called at that site. Building the matching graph consists of:

1. Find all of the VFC sites. Each VFC site is a triplet (*loc, f, A*), where *loc* defines the exact location of this call expression in the C++ source code, *f* is the virtual function name that appears in this call expression, and *A* is a class type name. If the call expression is coded via a class member access, i.e. using dot (.) or arrow ($\rightarrow$), preceded by a class object, or a pointer to class object, then the type name *A* is that class. Otherwise, the call is invoked against *this* and it appears in a function member. In that case, the type name is the containing class.

2. Find for a class type *A*, and for one of its virtual functions *f*, all the definitions of *f* that can possibly be called at VFC sites (*loc, f, A*). The list of possible target methods consists of the implementations of *f* introduced by all types derived from *A*, and the implementation of *f* used by *A* (either defined by *A* or inherited from an ancestor of *A*).

3. Connect each VFC site (*loc, f, A*) with all possible methods as found for *A* and *f*. Note that each method definition specifies a class type, namely the type that introduces this definition. Thus, the set of arcs from a particular VFC defines a set of specified types as described in the general type-specification technique.

After building the matching-graph, each VFC site node connected to a single possible target method is included in the list of candidates which is the output of the UN analysis component.

To accomplish Step 2, we use the browser files that are optionally generated by our compiler [14]. A *.pdb* file is created, while *-qbrowse* is specified, for every input C++ file. This is a binary file containing all information that satisfies the C Set ++ browser requirements. By parsing this file, we are able to reconstruct the class hierarchy, and keep track of all virtual methods defined in the program.

The *.pdb* file, however, was found to be unsuitable for Step 1 above. The need to specify a class type with each VFC site, as described above, is too complicated using the *.pdb* structure. In contrast, a very simple variation of the compiler front-end provides this kind of information.

It is worth noting that the global view of the whole program can be obtained using any program data-base or compiler based tool. Our analysis phase could have used other tools such as CodeStore [5] to build the matching-graph.

4.2 The Single Type Prediction Analysis Component

In STP analysis phase we can point out two main stages:

1. Collection of dynamic information (using profiling).

2. Analysis of the above information to result in a candidate list.

In order to collect relevant dynamic information, we implement a special component. This component is a variation of the compiler front-end which transforms each

VFC expression to a comma expression. The comma expression prefixes a call to a special *update* function before the original VFC expression. The *update* function takes two parameters: the address of the VFT used to resolve the call, and the source location of the VFC site. The first parameter cannot be computed at compile-time; it represents the dynamic type of the object against which the call is invoked. Every time control goes through such an instrumented expression, some accumulative variable is updated, recording the fact that a particular VFT is used in the corresponding VFC site.

Running the instrumented program, we collect information on the various VFTs used in each VFC site, and the distribution of calls among these VFTs. When the original program execution ends, the instrumented program writes collective information to binary files.

Summing it up, the phase of dynamic information collection consists of two parts - compiling the program using the special profiling component, and running the program. The result of this phase is the distribution of calls among VFTs in each VFC site. Note that we consider the address of a VFT to be an indication of the type of the object pointing to that VFT.

In the next stage, analysis of the dynamic information, we analyze the output binary file and decide for each VFC site, whether or not to consider it as a candidate site. Also, each candidate VFC site gets associated with some *predicted* VFT. Notice first that since our *type-info* expression relates to a VFT's address, a predicted VFT must be uniquely defined in all compilation units. In other words, a VFT which gets defined in more than one module, so called *duplicated* VFT, cannot serve as a predicted VFT. Other than that, we apply no special heuristic in choosing candidates and predicted VFTs. Every VFC site that is visited at least once during our sample executions is considered to be a candidate VFC site, unless all its associated VFTs are duplicated ones. We choose the most frequently-used unduplicated VFT in that site to be its *predicted* VFT.

4.3 The Transformation Components

Each transformation component is a variation of our compiler front-end, which receives a file containing candidates to be transformed and generates optimized code at the designated places. Each optimizing front-end checks during semantic processing of a VFC site whether additional information regarding this site was supplied as input from the analysis phase. If so, transformation is performed using the name of the *specified* or *predicted* type associated with this candidate.

We implement three transformation components: UN transformation component, STP transformation component and UN&STP one.

The UN transformation component replaces each UN candidate. If the candidate VFC refers to a virtual function *f*, and the specified type is *A*, the optimizing compiler does not generate the original indirect function call expression, but rather a direct call expression to *A::f*.

The STP transformation component replaces each STP candidate. For a candidate VFC that refers to a virtual function *f*, and to some predicted VFT, the optimizing compiler generates a conditional expression <exp> ? <call-exp1> : <call-exp2>. The expression <exp> is an equality expression that compares between $this \rightarrow \_\_vfp$, the dynamic address of the VFT which is used to resolve the call, and the address of the predicted VFT. The <call-exp1> expression is a direct call, whereas <call-exp2> is the original indirect call expression. The direct call invokes the definition of *f*, the address of which appears in the predicted VFT.

An STP candidate may also be found as a UN candidate. Applying an STP transformation to such a candidate site involves evaluation of <exp>, in addition to the direct invocation in <call-exp1>. The UN&STP transformation component is a combination of UN and STP that prevents this unnecessary overhead. Each UN candidate is replaced by a direct call, and each STP candidate, which is not a UN candidate, is translated to a conditional expression.

4.4 Limitations

Our current transformation components suffer a few limitations:

1. Casting from a virtual base
2. Out of scope VFC sites

Suppose one of our optimizing transformation components processes a VFC site, whose associated class type is *A*, and suppose this VFC is a candidate, the specified (or predicted) type of which is other than *A*, say *B*.

Casting from a virtual base

The direct call involves adjustment of the *this* pointer, as if we were casting *A* to *B*. Currently, no optimization is made if *A* is a virtual base class and *B* is a derived class. Such casting cannot be done at compile-time, thus extra commands have to be generated. We plan to facilitate such casting using data encompassed in VFTs.

Out of scope VFC sites

No optimization is made if *B* cannot be used in the VFC's location, i.e. *B* is not yet defined. Refer to the example in Section 3. The VFC in line 13 of *A.C* corresponds to the base class *A*. This is an STP candidate, the predicted type of which is *B*. But, the class *B* is not defined in *A.C*. This call is not optimized.

We strongly believe that this limitation can be solved within a compiler front-end, by forcing it to refer to *B*'s function members as *extern* symbols. This strengthens our direction to optimize VFCs using compiler variations. An alternative direction, that of single-pass source-to-source translation will remain limited; there is no way in C++ source code to overcome the out of scope problem, unless some code re-arrangement is done in advance (as it is done in [2]).

5 Evaluation

We experiment our implementations against C++ programs that are being used by other groups in IBM as C++ benchmarks [11]. We are examining only those programs that exhibit polymorphism through virtual functions, and can be compiled using the C Set ++ compiler. A brief description of the benchmarks is brought in the following sub-section.

5.1 Benchmarks Used

MetaWare

A performance test created by MetaWare. No input required. The program prints run-time measurements of three different tests. The first two tests construct complicated objects and make heavy use of virtual function calls. The focus of the third test is in applying copy constructors. Obviously, our optimizations significantly affect the first two tests.

Tal-Dict (SOMB)

A performance test of Taligent dictionary class. No input required. The program creates a dictionary of small objects, inserts some objects into the created dictionary, and makes a lot of look-ups into the dictionary. Since none of the methods are inline functions, we consider this program a good benchmark to test the improvement gained without inlining.

LCOM

A compiler for the hardware description language L. We used one of the inputs supplied with the distribution package (circuit2.l).

5.2 Profiling Results

To evaluate the potential cost and benefit of our optimizations, we use a set of measurements:

1. Function call sites

   We count all call sites in the user-compiled code including compiler-generated calls (and calls within compiler-generated functions), such as constructors and destructors. Files are compiled with no inlining and no optimization. For example, *B.C* in our running example includes ten function call sites: four are used to define *ptrA* and *ptrB*, five are explicitly coded in lines 6, 7, 8, 9 and 11. Another call appears in the default compiler-generated constructor for *B*. The latter calls *A*'s constructor.

2. Active function calls

   Number of function calls performed during a program execution. We count only those calls invoked from sites referred in item 1 above. In our example we have 100,000,011 active function calls.

For the above two items we implemented a profiling tool, based on a compiler front-end variation, that instruments the code. Running this instrumented program yields the required numbers.

3. VFC sites

   Number of all call sites in the user code where unqualified virtual function call expression is coded. In our example there are 5 VFC sites.

The next seven measurements are computed using the STP analysis component. All numbers refer to a specific program execution. Recall that the STP analysis component computes distribution of calls among unduplicated VFTs in each VFC site. Thus, for each VFC site, i, we get n_i numbers, each of which represents the number of times a particular VFT is used.

4. Active virtual function calls

   Number of calls corresponding to VFC sites performed during a program execution, and its percentage out of all active function calls. For our example we get 100000003 active virtual function calls.

5. Unaccessible VFC sites

   VFC sites that are never reached in a program execution. In our example, there are no unaccessible VFC sites.

6. Virtual calls using a single table

   Summing up all numbers that correspond to VFC sites for which $n_i = 1$, and the percentage out of all active virtual function calls. In our example, each VFC site uses a single table.

7. Virtual calls using two tables

   As before with $n_i = 2$.

8. Virtual calls using more than two tables

   As before with $n_i > 2$.

9. Virtual calls using most frequent table

   The number we get from summing the number of calls taken through the predicted table (i.e. the most frequently used VFT) in each candidate VFC site, and the percentage out of all active virtual function calls. This is the total number of calls that will turn to be direct calls after our STP transformation.

10. Virtual calls using table with >50% frequency

    As in the previous item, but referring only to those VFC sites for which the predicted table is used more than in 50% of the cases.

The last two items are obtained using the UN analysis component, and profiling information.

11. Virtual calls with unique inline implementation

    Number of calls corresponding to UN VFC candidate sites performed during a program execution, and its percentage out of all active virtual function calls. We consider here only those candidates for which the single implementation is an inline function.

12. Virtual calls with unique non-inline candidates

    As in the previous item, but the single implementation is not an inline function.

Table 2 presents the results of these profiling measurements on our benchmarks.

5.3 Run-Time Performance

Run-time measurements for our benchmarks can be found in Table 3. Files are compiled with inlining (-Q) and back-end optimizations (-O). Times are measured in seconds on IBM RISC System/6000, Model 390.

Results for MetaWare show an improvement of 20%. Although the percentage of virtual functions in the benchmark is relatively low, the inlining that follows the transformation makes the improvement extremely significant. Note that all VFC sites in MetaWare are UN candidates, and hence the run-time result after applying UN&STP is identical to that achieved by UN optimization alone. When applying STP optimization to MetaWare, the cost of evaluating the conditional expression causes the improvement to be less significant than the one gained by UN. The first two tests in MetaWare make heavy use of the virtual function call dispatching mechanism. Run times for MetaWare's first two tests are brought on a separate line to emphasize that opportunities for transformation are local to some part of the program, in which improvement of 43% is reached.

For Tal-Dict, improvement performance achieved by STP is better than the one achieved with UN since only 41% of the VFC sites are UN candidates. When combining UN&STP , we reach a run time of 11.4 seconds, which is a 20% improvement of the original run time.

Poor results for LCOM are due to the low percentage of active virtual function calls in the program. In addition to that, only about 1% of them correspond to UN candidates.

It is worth noting that all of the results brought here were achieved using our limited transformation (see Section 4.4). In fact, we are unable to transform about 25% (average) of all VFC sites. Better results are expected in future implementations of our components.

6 Summary

The C++ community recognizes the importance of C++ performance, and turns to be more and more conscious about the need to enhance run-time results. Given that many of C++ users come from the arena of C programming, and given the well-established notion of compiler optimization in C, it is very much expected that C++ compilers are to put more effort on code optimization. The fact that C++ programs perform indirect calls more often than C programs is one of the most significant dynamic performance characteristics of C++ versus C. Most indirect branches are caused by virtual function calls. This all

	MetaWare	Tal-Dict	LCOM
Function call sites	60	1706	2871
Active function calls	74000030	53125698	5831087
VFC sites	5	80	455
Active virtual function calls	5000000 6%	35060980 66%	1100949 19%
Unaccessible VFC sites	0	67	154
Virtual calls using single table	5000000 100%	35060980 100%	569218 52%
Virtual calls using two tables	0 0%	0 0%	14903 1%
Virtual calls using more than two tables	0 0%	0 0%	516828 47%
Virtual calls using most frequent table	5000000 100%	35060980 100%	1071925 97%
Virtual calls using table with >50% frequency	5000000 100%	35060980 100%	1055810 96%
Virtual calls with unique inline implementation	5000000 100%	0 0%	5443 0.5%
Virtual calls with unique non-inline implementation	0 0%	14321318 41%	4822 0.4%

Table 2: Profiling results for benchmarks

	Original	UN	STP	UN&STP	Performance improvement
MetaWare	6.2	5.0	5.6	5.0	20%
MetaWare's first two tests	2.8	1.6	2.1	1.6	43%
Tal-Dict	14.3	12.4	12.3	11.4	20%
LCOM	3.7	3.65	3.55	3.55	4%

Table 3: Run-time measurements for benchmarks

brings us to our strong belief that optimizing C++ virtual function calls is one of the most challenging directions in C++ compiler optimizations.

As shown in this article, replacing indirect virtual function calls by direct calls or conditional expressions seems to be a very promising solution to reduce the cost of these calls. Our main contribution is in introducing techniques that support the whole spectrum of code analysis and code transformation. The fact that we develop variations of a production compiler makes the opportunity of introducing such improvements into a real compiler firm and realistic.

Still, there is much room for further investigation. In order to better assess our contribution we intend to apply our techniques on additional benchmarks. We are now communicating with other groups to accept additional C++ programs that may serve as benchmarks for our optimization work. We are facing the following problems:

1. Language compatability. Many programs do not get compiled with our version of the C Set ++ compiler.

2. Object-orientation style. Many programs do not explore polymorphism to the extent that make them good candidates for our techniques.

3. Size. Some interesting real programs are too heavy and large to be examined and processed under our limited development resources. An example of such a system is described in [6].

More experiments will facilitate a thorough analysis of the cost and benefit of our transformations, as well as evaluating the significance in introducing inlining opportunities. We plan to apply techniques to measure the stability of our type-profiling information, similar to what is done in [9].

There are many papers that focus on the UN analysis phase. We plan to communicate with some leading researchers in this area, and make serious comparisons between their work and ours. Of particular interest is a recent work by Bacon and Sweeney ([4]).

More implementation effort should be put to eliminate current limitations of our transformation tools (see Section 4.4). We also plan to implement the general *type-specification* and *type-prediction* optimizations. Obviously enough, the main effort in carrying out these generic optimizations is expected to be exactly at that point that was overly simplified in the special cases, UN and STP, namely the need to apply heuristics to decide on the candidate VFC list. The process of evaluating cost and benefit becomes very complicated in this case.

References

[1] O. Agesen and U. Holzle. Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages. In *OOPSLA '95*, Austin, Texas, 1995.

[2] G. Aigner and U. Holzle. Eliminating Virtual Function Calls in C++ Programs. Technical Report TRCS 95-22, Dept. of Computer Science, University of California, Santa Barbara, December 1995.

[3] D.F. Bacon, M. Burke, G. Ramalingam, H. Srinivasan, and P.F. Sweeney. Compiler Optimizations for C++. Technical report, IBM Thomas J. Watson Research Center, March 1995.

[4] D.F. Bacon and P.F. Sweeney. Rapid Type Inference in a C++ Compiler. Technical report, IBM Thomas J. Watson Research Center. Submitted to PLDI'96, 1995.

[5] J. Barton, P. Charles, Y.-M. Chee, M. Karasick, D. Lieber, and L. Nackman. Codestore: Infrastructure for C++ - Knowledgeable Tools. In *OOPSLA '94*, 1994.

[6] W. Berg, M. Cline, and M. Girou. Lessons Learned from the OS/400 OO Project. *Communications of the ACM*, 38(10):54–64, October 1995.

[7] B. Calder and D. Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. In *Conference Record of the Twenty-First ACM symposium on Principles of Programming Languages (POPL)*, pages 397–408, Portland, Oregon, January 1994. ACM Press, New York, New York.

[8] B. Calder, D. Grunwald, and B. Zorn. Quantifying Behavioral Differences Between C and C++ Programs. Technical Report CU-CS-698-94, University of Colorado at Boulder, January 1994.

[9] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-Guided Receiver Class Prediction. In *OOPSLA '95*, Austin, Texas, 1995.

[10] U. Holzle and D. Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *SIGPLAN '94 Conference on Programming Languages Design and Implementation*, pages 326–336. ACM Press, June 1994.

[11] Y. Lee and M.J. Serrano. Dynamic Measurements of C++ Program Characteristics. Technical Report TR ADTI-1995-001, IBM, January 1995.

[12] H.D. Pande and B.G. Ryder. Static Type Determination for C++. In *Proceedings of the Sixth Usenix C++ Technical Conference*, pages 85–97, April 1994.

[13] J. Plevyak and A. Chien. Type Directed Cloning for Object-Oriented Programs. Technical report, University of Illinois at Urbana-Champaign, www-csag.cs.uiuc.edu/individual/jplevyak, 1996.

[14] IBM Publication. *IBM C Set++ for AIX/6000: User's Guide*, 1993.