Eliminating Virtual Function Calls in C++ Programs

Gerald Aigner
Urs Hölzle

Department of Computer Science
University of California
Santa Barbara, CA 93106
http://www.cs.ucsb.edu/oocsb

December 10, 1995

Abstract: We have designed and implemented an optimizing source-to-source C++ compiler that reduces the frequency of virtual function calls. This technical report describes our preliminary experience with this system. The prototype implementation demonstrates the value of OO-specific optimization of C++. Despite some limitations of our system, and despite the low frequency of virtual function calls in some of the programs, optimization improves the performance of a suite of two small and six large C++ applications totalling over 90,000 lines of code by a median of 20% over the original programs and reduces the number of virtual function calls by a median factor of 5. For more call-intensive versions of the same programs, performance improved by a median of 40% and the number of virtual calls dropped by a factor of 21. Our measurements indicate that inlining does not necessarily lead to large increases in code size, and that for most programs, the instruction cache miss ratio does not increase significantly.

Technical Report TRCS 95-22, Department of Computer Science, University of California, Santa Barbara, December 1995. UCSB Computer Science technical reports are available via ftp from ftp.cs.ucsb.edu or via WWW from http://www.cs.ucsb.edu/TRs.

1. Introduction

Object-oriented programming languages confer many benefits, including abstraction, which lets the programmer hide the details of an object's implementation from the object's clients. Unfortunately, object-oriented programs are harder to optimize than programs written in languages like C or Fortran. There are two main reasons for this. First, object-oriented programming encourages code factoring and differential programming [Deu87]; as a result, procedures are smaller and procedure calls more frequent. Second, it is hard to optimize calls in object-oriented programs because they use *dynamic dispatch*: the procedure invoked by the call is not known until runtime since it depends on the dynamic type of the receiver. Therefore, a compiler usually cannot apply standard optimizations such as inline substitution or interprocedural analysis to these calls.

Consider the following example (written in pidgin C++):

```
class Point {
  virtual float get_x();              // get x coordinate
  virtual float get_y();              // ditto for y
  virtual float distance(Point p);    // compute distance between receiver and p
}
```

When the compiler encounters the expression `p->get_x()`, where `p`'s declared type is `Point`, it cannot optimize the call because it does not know `p`'s exact runtime type. For example, there could be two subclasses of `Point`, one for Cartesian points and one for polar points:

```
class CartesianPoint : Point {
  float x, y;
  virtual float get_x() { return x; }
  (other methods omitted)
}
class PolarPoint : Point {
  float rho, theta;
  virtual float get_x() { return rho * cos(theta); }
  (other methods omitted)
}
```

Since `p` could refer to either a `CartesianPoint` or a `PolarPoint` instance at runtime, the compiler's type information is not precise enough to optimize the call: the compiler knows `p`'s *declared type* (i.e., the set of operations that can be invoked, and their signatures) but not its *actual type* (i.e., the object's exact size, format, and the implementation of operations).

Since dynamic dispatch is frequent, object-oriented languages need optimizations targeted at reducing the cost of dynamic dispatch in order to improve performance. So far, much of the research on such optimizations has concentrated on pure object-oriented languages because the frequency of dynamic dispatch is especially high in such languages. Several studies (e.g., [CUL89], [HU94a], [G+95]) have demonstrated that optimization can greatly reduce the frequency of dynamic dispatch in pure object-oriented languages and significantly improve performance. However, so far no study has shown that these optimizations also apply to a hybrid language like C++ where the programmer can choose between dispatched and non-dispatched functions, and where programs typically exhibit much lower call frequencies.

We have developed a proof-of-concept optimizing compiler for C++ that demonstrates that optimization can significantly reduce the frequency of virtual function calls in C++, and that programs execute significantly faster as a result. On a suite of large, realistic C++ applications totalling over 90,000 lines of code, optimization improves performance by up to 48% and reduces the number of virtual function calls by up to 50 times.

2. Background

Since our main goal was to demonstrate the value of OO-specific optimizations for C++, we chose to implement and evaluate two relatively simple optimizations that have demonstrated high payoffs in implementations of pure object-oriented languages like SELF or Cecil. In this section, we briefly review these optimizations before describing our C++-specific implementation in the next section.

2.1 Profile-Based Optimization: Type Feedback

Type feedback [HU94a] is an optimization technique originally developed for the SELF language. Its main idea is to use profile information gathered at run-time to eliminate dynamic dispatches. Thus, type feedback monitors the execution characteristics of individual call sites of a program and records the set of receiver classes encountered at each call site, together with their execution frequencies. Using this information, the compiler can optimize any dynamically-dispatched call by *predicting* likely receiver types and inlining the call for these types. Typically, the compiler will perform this optimization only if the execution frequency of a call site is high enough and if one receiver class dominates the distribution of the call site. For example, assume that `p` points to a `CartesianPoint` most of the time in the expression `x = p->get_x()`. Then, the expression could be compiled as

```
if (p->class == CartesianPoint) {
  // inline CartesianPoint case (most likely case)
  x = p->x;
} else {
  // don't inline PolarPoint case because method is too big; this branch also covers all other receiver types
  x = p->get_x();    // dynamically-dispatched call
}
```

For `CartesianPoint` receivers, the above code sequence will execute significantly faster since the original virtual function call is reduced to a comparison and an assignment. Inlining not only eliminates the calling overhead but also enables the compiler to optimize the inlined code using dataflow information particular to this call site.

In the SELF-93 system, the system collects receiver type information on-line, i.e., during the actual program run, and uses dynamic compilation to optimize code accordingly. In contrast, a system using static compilation (like the present C++ compiler) gathers profile information off-line during a separate program run.

2.2 Static Optimization: Class Hierarchy Analysis

We also implemented a static optimization technique, class hierarchy analysis (CHA) [App88][DGC95][Fer95], which can statically bind some virtual function calls given the application's complete class hierarchy. The optimization is based on a simple observation: if `a` is an instance of class `A` (or any subclass), the call `a->f()` can be statically bound if none of `A`'s subclasses overrides `f`. CHA is simple to implement and has been shown to be effective for other systems [DGC95][Fer95], and thus we included it in our prototype compiler.

Another benefit of combining a static optimization like CHA with a profile-based optimization like type feedback is that they are complementary optimizations: each of them may improve performance over the other. CHA may provide better performance because it can inline or statically bind sends with zero overhead. Since CHA can prove that a call can invoke only a single target method in all possible executions, any dispatch overhead is completely eliminated. In contrast, a profile-based technique like type feedback can inline the same send only by using a type test; even if the profile shows that the send always invoked the same target method, a test must remain since other target methods may be reached in different executions of the program. Thus, while the send can still be inlined, some dispatch overhead remains. On the other hand, type feedback can optimize any function call, not just monomorphic sends. Furthermore, being profile-based, it can also better determine whether the send is actually worth optimizing (i.e., executed often enough).

3. Related Work

Profile information has been used for optimization in many systems; as usual, Knuth [Knu70] was the first to suggest profile-based optimization, and today it is part of many research systems (e.g., [CM+92], [Höl94] [G+95]) as well as production compilers. Studies of inlining for procedural languages like C or Fortran have found that it often does not significantly increase execution speed but tends to significantly increase code size (e.g., [DH88], [HwC89], [CHT91], [CM+92], [Hall91]). Our results indicate that these previous results do not apply to C++ programs.

In implementations of dynamic or object-oriented languages, profiling information has often been used to identify (and optimize for) common cases. For example, Lisp systems usually inline the integer case of generic arithmetic and handle all other type combinations with a call to a routine in the runtime system. The Deutsch-Schiffman Smalltalk compiler was the first object-oriented system to predict integer receivers for common message names such as "+" [DS84]. All these systems do not use application-specific profiles.

The SELF system pioneered the use of profile information for optimizing object-oriented languages. An experimental proof-of-concept system [HCU91] was the first one to use type feedback (then called "PIC-based inlining") for optimization purposes. The SELF-93 system [HU94a] used on-line profile information to select frequently executed methods for optimization and to determine receiver types via type feedback. Similarly, the Cecil compiler [G+95] uses off-line profiling for optimization and inlining. Grove et al. [G+95] also examined the cross-input stability of receiver class profiled in C++ and Cecil and found it good enough to be used for optimization.

Until now, few profile-based techniques have been applied to hybrid, statically-typed languages like Modula-3 or C++. Based on measurements of C++ programs, Calder and Grunwald [CG94] argued that type feedback would be beneficial for C++ and proposed (but did not implement) a weaker form of class hierarchy analysis to improve efficiency. Their estimate of the performance benefits of this optimization (2-24% improvements, excluding benefits from inlining) exceeds the improvements measured in our system, partially because they assume a more aggressively pipelined CPU (DEC Alpha) which benefits more from reduced pipeline stalls than the SuperSPARC system we measured. Fernandez [Fer95] applied link-time optimization to Modula-3 programs and found that class hierarchy analysis eliminated between 2% and 79% of the virtual calls in the Modula-3 applications measured, reducing the number of instructions executed by 3-11%. Profile-driven customization (procedure cloning) resulted in an additional improvement of 1-5%.

Several systems use whole-program or link-time analysis to optimize object-oriented programs, starting with the Apple Object Pascal linker [App88] which turned dynamically-dispatched calls into statically-bound calls if a type had exactly one implementation (e.g., the system contained only a `CartesianPoint` class and no `PolarPoint` class). To our knowledge, this system was the first to statically bind dynamically-dispatched calls, although it did not perform any inlining. As mentioned above, Fernandez [Fer95] used class hierarchy analysis for Modula-3, and Dean et al. [DGC95] describe its use for Cecil. In both studies, the analysis' impact on virtual call frequency was significantly higher than in our system, as discussed in section 6.1. Srivastava and Wall [SW92] perform more extensive link-time optimization but do not optimize calls.

More ambitious analyses such as concrete type inference systems (e.g., [Age95], [PR94], [PC94]) can determine the concrete receiver types of message sends. Compared to type feedback, a type inferencer may provide more precise information since it may be able to prove that only a single receiver type is possible at a given call site. However, its information may also be less precise since it may include types that could occur in theory but never happen in practice. (In other words, the information lacks frequency data.) For SELF, concrete type inference removed more dispatches than type feedback for most programs [AH95]. Like link-time optimizations, the main problem with type inference is that it requires knowledge of the entire program, thus precluding dynamic linking.

4. Implementation

This section describes the implementation of our optimizing source-to-source C++ compiler as well as the motivations for its design decisions.

4.1 Overview

The implementation consists of several parts (Figure 1). First, a pre-pass combines the original sources into a single baseline program to simplify the work of later phases (section 4.2). Then, the program is compiled with a modified version of GNU C++ to produce an instrumented executable which serves to collect receiver class profile information (section 4.4). The centerpiece of the system, the optimizing compiler, then uses this profile to transform the baseline source program into an optimized source program (section 4.5) which is subsequently compiled and optimized by the native host C++ compiler. The rest of this section describes these steps in more detail.

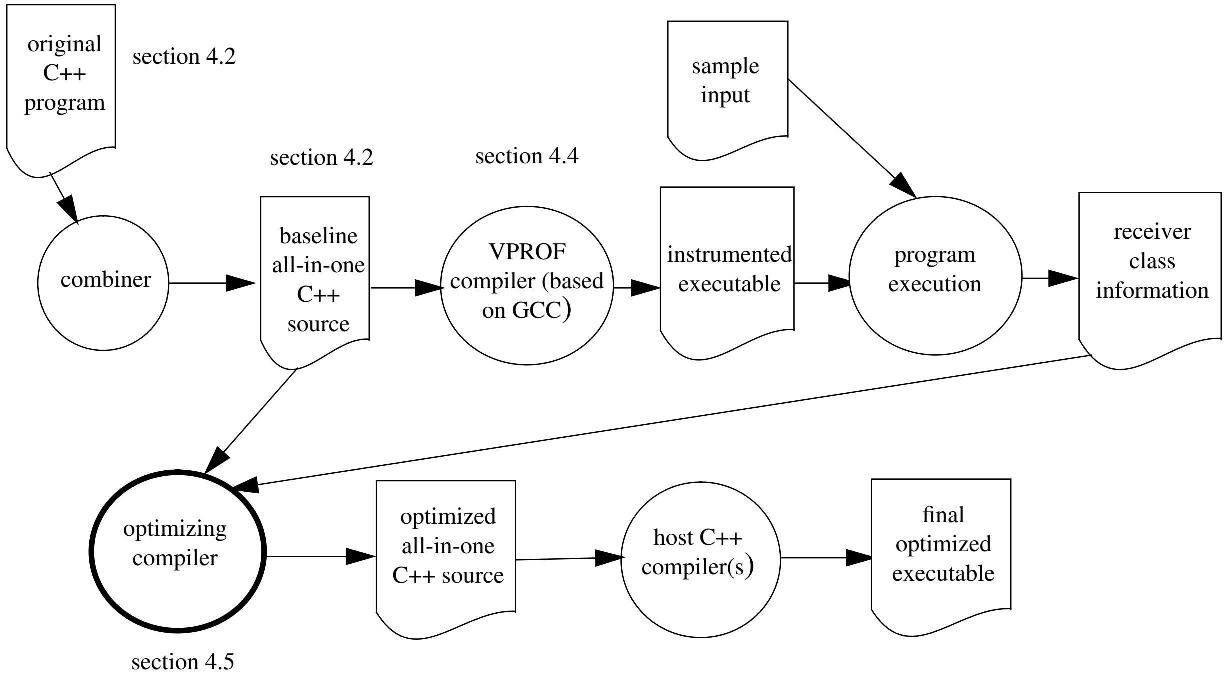

Figure 1. Overview of optimization process

4.2 Program Preparation

Before any optimization begins, a pre-pass merges the source files of the original C++ program into a single C++ source file that contains all the declarations, global variables, and functions of the original program. The various program parts are arranged in a suitable order so that all declarations precede any use (see Table 1). Within the "inline" and "function" parts, function

Source Section	Description
declarations	all declarations normally found in C/C++ header files, like prototypes, typedefs, class declarations, and external declarations; contains no function definitions
global	all global variables
inline	all inline functions
function	all non-inline functions

Table 1. The sections of a baseline C++ program

definitions are sorted topologically to improve subsequent inlining by the host compiler (some C++ compilers do not inline functions unless the definition of the inlinable function precedes the use).

In later sections, we will refer to programs compiled from the original source as "original" programs and those compiled from the all-in-one equivalent as "baseline" programs. Combining the entire program's code into the single baseline file simplifies the subsequent optimization process. In particular, the compiler does not need to handle cross-module (cross-file) inlining, i.e., obtaining the body of a function to be inlined is much simpler than if the program were divided into separate compilation units.

To implement this pre-pass, and for the actual optimizing compiler, we used a commercial tool, the CCAuditor C++ parser from Declarative Systems [Dec95]. CCAuditor contains an empty attribute grammar for C++ which can be extended to implement arbitrary analyses or transformations. We used CCAuditor to build a simplified parse tree of the program which is then traversed and transformed by a C++ program. CCAuditor proved to be an invaluable tool since it already handled the semantic analysis of the C++ code (i.e., resolution of overloaded functions, copy constructors, etc.), thus relieving us from dealing with C++'s difficulties.

4.3 Source-To-Source Transformation

We chose to implement our compiler using source-to-source transformation for several reasons:

- It is simpler to implement and test; in particular, the generated (source) code is much easier to debug than assembly code. In essence, the back-end C++ compiler serves as an output verification tool.
- Source-to-source compilation allows experimenting with several back-end compilers to determine how much the quality of back-end code generation affects performance. Also, using well-tested back-end compilers provides more reliable results (and usually better performance) by ensuring that performance isn't affected or obscured by unknown back-end deficiencies.
- Similarly, it simplifies porting the compiler to new architectures to compare optimization effects on different hardware platforms and to estimate the impact of architectural features on performance.

The main alternative, changing an existing compiler, implied using the GNU C++ compiler, as it is the only C++ compiler for which source code is readily available. Unfortunately, the GNU compiler does not build complete parse trees for functions; instead, it builds parse trees only as far as necessary to perform its task. Without parse trees, the high-level transformations performed by the optimizing compiler are hard to implement. (For example, it is hard to duplicate or modify the body of a function later.) This deficiency, coupled with the problems of understanding a compiler of gcc's size and the advantages of source-to-source transformation, tilted the scales towards the current solution.

On the other hand, source-to-source transformation is not without problems. In particular, it restricts the extent to which high-level information can be transmitted to the back end. For example, the optimizer may have a very good idea of which execution paths are more likely, but this information cannot easily be encoded in C++, and thus the back end does not benefit from this information. In general, the source-to-source compiler has less control over the final generated code; for example, it cannot force the back end compiler to inline a function since the `inline` modifier is only a hint, not an order. Finally, some constructs cannot be portably expressed in source form. For example, type tests are back-end-dependent since different back end compilers may use different vtable allocation strategies. In spite of these potential problems, we felt that the advantages of source-to-source optimization outweighed the disadvantages in a research setting.

4.4 Collecting Receiver Class Profiles

Before actual optimization starts, the baseline file is compiled with VPROF, a modified version of the GNU C++ compiler. VPROF inserts a non-virtual call to a run-time routine before each virtual call, passing in all information needed to identify the call site (e.g., file name, line number, and call number within the line). Additionally, the run-time routine receives a pointer to the receiver's dispatch table (vtable [ES90]) and the index of the vtable entry being used to dispatch the call. In order to obtain the class name of receiver and the method name, the compiler enhances the vtable with one extra element per entry containing the necessary information.

The resulting executable is then used to obtain receiver class distribution information for each call site. At the end of the program run, a small run-time library collects and outputs the data collected; this output is later used by the optimizing compiler. In Grove's terminology [G+95], VPROF produces 1-CCP profiles, i.e., individual receiver class distributions for each call site.

4.5 The Optimizing Compiler

The main optimization step in our system consists of a source-to-source optimizing compiler that eliminates virtual function calls using either the profile information, knowledge of the complete class hierarchy, or both. This section describes the main implementation difficulties we faced as well as their solutions (summarized in Table 2). The next section then describes the optimization policies used.

Problem	Solution
complexity of changing an existing C++ compiler	source-to-source optimization
body of methods to be inlined must be known during the optimization phase	transform the whole program into one file
implementation of a fast type check	use the address of a vtable to determine the class type
virtual function to be inlined may not be accessible (declared protected or private)	add dispatch_function to static receiver class
get back-end C++ compiler to inline a virtual function while leaving the original function for other callers	duplicate the original virtual function , change name, remove virtual attribute, add inline attribute
cast receiver pointer down from virtual base class	create a helper application computing the offsets for all possible class combinations
static functions or variables with the same name	move to global scope and rename

Table 2. Problems and Solutions for the source-to-source approach

4.5.1 Type Tests

One of the central transformations of the optimizing compiler is to replace virtual function calls with more efficient equivalents. Many of these transformations involve a type test that tests for a predicted class. So far, we sketched this test as a simple comparison, e.g., `p->_class == CartesianPoint`. Unfortunately, this isn't legal C++ code: neither do objects contain a `type` field, nor can a class be used as a run-time constant. So, how can such type tests be implemented? We considered three alternatives.

First, the compiler could add a new virtual function `__type` to each class, and have each of them return a unique type identifier (class number). Then, the type test could be written portably as `p->__type() == CartesianPointID`. Unfortunately, this approach isn't very attractive since in trying to eliminate a virtual function call the compiler introduces another virtual function call.

Alternatively, the compiler could add an additional instance variable to each object containing virtual functions and change the class constructors to initialize this field to the class' ID. This approach provides fast type tests (a load and a comparison) but leads to two major problems. First, the size of each object grows by one word; if multiple inheritance is used, the object even contains multiple extra fields, one for each base class that uses virtual functions. The additional type field not only uses more space but will also impose a run-time penalty since the additional field has to be initialized; also, the extra words may increase the data cache miss ratio. Second, subclasses cannot easily access the type field if they inherit base classes privately, so that the type field is hidden. To solve this problem, the code could directly access the type field using casts, as in `*((int*)(BaseClass*)this)`. However, this solution is non-portable since different C++ compilers may use different object layouts.

The third approach, which is the one we are currently using, is an extension of the method described above. Instead of adding a new variable and initializing it to a unique value, it uses a variable already added by the compiler, the vtable pointer (dispatch table pointer). Although there are some difficulties with this approach, they can all be solved:

- *A class may have multiple vtables*. C++ does not guarantee anything about vtables, and thus a compiler may generate multiple vtables for a single class (e.g., one per .c file where the class definition was included). Although this was a problem in early C++ implementations, today's compilers try hard to avoid multiple vtables in order to produce smaller executables, and thus multiple vtables have not been a problem in practice. Furthermore, their presence would only reduce the efficiency of the optimized programs, not their correctness.
- *A class may share the vtable of one of its superclasses* (since the superclass vtable is a prefix of (or identical to) the subclass vtable). This sharing isn't problematic since the compiler uses the type information only for dispatching methods. If two classes share the same vtable, then they will behave identically with respect to all functions called through this vtable, and thus can share inlined code as well. In fact, such sharing is beneficial for type feedback as it allows multiple receiver classes to share the same piece of inlined code.
- *The vtable isn't a C++ entity, so that a source program cannot name it*. This problem is caused by source-to-source optimization, and we circumvent it by using a dummy "vtable address" constant in the comparison and post-editing the assembly file generated by the back-end compiler.
- *The position of the vtable field is unknown.* This is the most serious problem, again caused by using source-to-source optimization. One solution is to introduce an additional base class for every base class in the program which has a superclass with virtual functions. The sole purpose of this base class is to contain a virtual function definition, forcing the compiler to place the vtable at the beginning of the object. This solution would work well for single-inheritance

hierarchies but breaks down for multiple inheritance since it doesn't force the placement of multiple vtable pointers in an object.

Instead, our system uses a helper application (automatically generated during the initial prepass over the original program) to compute the offsets of all subobjects within an object and assumes that each subobject's vtable is at offset 0. (If this assumption were false, the helper program could create a zeroed-out instance of each class in the original program and find then the positions of embedded vtable pointers.) The offset information is also needed to perform casts.

To summarize, source-to-source optimization poses some difficulties that are caused by the need to access implementation-dependent information at the source level. However, none of these difficulties are insurmountable. In practice, they mainly impact the portability of the generated optimized source code since different back-end compilers may require different vtable manipulation code.

<table>
<tr><th>original program</th><th>optimized program</th></tr>
<tr><td><pre>
class A {
  virtual int foo();
};

class B : public A {
private:
  virtual int foo();
};

int bar(A *a) {
  a->foo(); // a contains an instance
            // of class B for 90% of
            // all invocations
...
}
</pre></td><td><pre>
class A {
  ...
  inline int dispatch_B_foo();
};

class B : public A {
  ...
public:
  inline int inline_B_foo();
};

inline int B::inline_B_foo() {
  /* modified copy of the source of B::foo() */
}

inline int A::dispatch_B_foo() {
  if (this->type == type_B)
    return ((B*)this)->
      B::inline_B_foo();
  else
    return foo();
}

int bar(A *a) {
  a->A::dispatch_B_foo();
}
</pre></td></tr>
</table>

Figure 2. Source-to-source optimization example

4.5.2 Optimization Example

Figure 2 shows a program fragment optimized with type feedback. The left column shows unoptimized code, and the right column the code created after optimizing the call `a->foo()` in function `bar`. In order to optimize this call, the compiler creates the dispatch function `A::dispatch_B_foo` which has the same signature as `A::foo` but is declared inline, public, and non-virtual. Using this dispatch method minimizes the syntactic transformation needed at the call site, even with nested function calls. In case the dynamic receiver class is B, the dispatch method calls `B::inline_B_foo();` in all other cases, a normal virtual method call is performed. The `inline_B_foo()` method serves two purposes. First, it ensures that the called method is declared inline; some C++ compilers only inline functions explicitly declared `inline`. Second, the inline function may be specialized since its receiver type is precisely known to be a B (and only a B). Thus, implicit self calls within the method can be statically bound.

4.6 Inlining Strategies

Some calls should not be inlined; for example, if a call has 20 different receiver types, each of which occurs 5% of the time, inlining is unlikely to improve performance: inlining just one case improves only 5% of the call site's executions but slows down the other 95%. Thus, for each call site, the compiler must decide whether to optimize it or not. Currently, the compiler considers two factors in its inlining decisions. First, it exploits peaked receiver class distributions by inlining only classes whose relative frequency exceeds a certain threshold. The compiler inlines at most one case per send. For example, the compiler's default threshold is 60%, and thus a call site won't be optimized unless the most frequent class represents more than 60% of all receivers at that call site. (Section 4.6 will show the performance impact of varying the threshold value.) With lower thresholds, more calls will be inlined, but chances are lower that the inlined code is actually executed, and thus actual performance may degrade because of the overhead of testing for the inlined case.

Second, the compiler restricts optimization to the "hot spots" of an application by considering the call site's contribution to the total number of calls in the application. For example, with the default threshold of 0.1%, the compiler will not optimize call sites responsible for less than 0.1% of the (virtual) calls executed during the profile run.[1] By inlining only the important calls, the compiler reduces the potential code growth; often, relatively few call sites account for most of the calls, and thus good performance can be achieved with only moderate amounts of inlining.

Finally, our optimizer relies on the inlining strategies of the back-end compiler to some extent since the inline keyword is only a suggestion to that compiler; if the function is too large, the back end may decide not to inline it. Consequently, our compiler currently does not take function size into account when deciding whether to optimize a call or not. If the back end does not actually inline a call, the only benefit of optimization is the elimination of the dispatch (in the case of class hierarchy analysis) or the replacement of a virtual function call with a somewhat faster comparison-and-direct-call sequence (for type feedback).

[1] Some of the measurements in this preliminary report were taken with a call site contribution threshold of 0, i.e., all call sites with a dominating (60%) receiver type were optimized. Except for code space, this does not influence the results much.

5. Experimental Setup

To evaluate the performance of our optimizer, we used a suite of two small and six large C++ applications totalling over 90,000 lines of code (Table 3). In general, we tried to obtain large, realistic applications rather than small, artificial benchmarks. Unfortunately, the choice of publicly available C++ programs which compile with current C++ compilers on current operating system versions (Solaris 2.X and AIX 4.1) is still limited. Two of the benchmarks (`deltablue` and `richards`) are much smaller than the others; they are included for comparison with earlier studies (e.g., [HU94a][G+95]). `Richards` is the only synthetic benchmark in our suite (i.e., the program was never used to solve any real problem).

program		lines of code	
name	description	original	baseline
deltablue	incremental dataflow constraint solver	1,000	1,400
eqn	type-setting program for mathematical equations	8,300	10,800
idl	SunSoft's IDL compiler (version 1.3) using the demonstration back end which exercises the front end but produces no translated output.	13,900	25,900
ixx	IDL parser generating C++ stubs, distributed as part of the Fresco library (which is part of X11R6). Although it performs a function similar to IDL, the program was developed independently and is structured differently.	11,600	11,900
lcom	optimizing compiler for a hardware description language developed at the University of Guelph.	14,100	16,200
porky	back-end optimizer that is part of the Stanford SUIF compiler system	22,900	41,100
richards	simple operating system simulator	500	1,100
troff	GNU groff version 1.09, a batch-style text formatting program	19,200	21,500

Table 3. Benchmark programs

Recall that "original" refers to programs compiled from the original sources, and "baseline" refers to the same programs compiled from the all-in-one source file without any inlining of virtual function calls. The latter versions are longer since they also contain system include files (/usr/include/...) and since the combiner pre-pass splits some constructs into multiple lines. For both versions, the line counts exclude empty lines, preprocessor commands, and comments.

In addition to measuring the unchanged programs, we also ran "all-virtuals" versions of the benchmark programs where every function (with the exception of operators and destructors, which currently cannot be optimized by our compiler) is declared as `virtual`. We chose to include these program versions in order to simulate programming styles that extensively use abstract base classes defining virtual functions only (C++'s way of defining interfaces). For example, the Taligent CommonPoint frameworks provide all functionality through virtual functions, and thus programs using Commonpoint (or similar frameworks) will exhibit much higher virtual function call frequencies [Ser95]. Lacking real, large, freely available examples of this programming style, we created the "all virtual" programs to provide some indication of how optimization would impact such programs.

program	classes	unmodified programs				"all-virtuals" programs			
		functions		virtual call sites		functions		virtual call sites	
		virtuals	nonvirt.	all	used	virtuals	nonvirt.	all	used
deltablue	10	3	78	3	3	76	5[a]	213	145
eqn	56	169	102	174	100	238	33	231	122
idl	83	374	454	1,248	578	671	157	2,088	786
ixx	90	445	582	596	147	991	36	3,026	824
lcom	72	314	508	460	309	744	78	1286	825
porky	118	274	995	836	163	N/A[b]			
richards	12	4	62	1	1	66	0	105	100
troff	122	484	403	405	98	804	83	1148	340

Table 4. Basic characteristics of benchmark programs

[a] As explained in the text, the compiler currently cannot optimize operators and destructors, and thus there are kept nonvirtual. Furthermore, constructors are always nonvirtual.
[b] Porky cannot be compiled as "all virtual" since it contains identically-named functions with varying argument or return types.

For each program, Table 4 shows some basic program characteristics such as the number of classes, C++ functions (excluding static member functions and nonmember functions), and virtual function call sites. For the latter, "all" refers to the total number of virtual function call sites in the program, and "used" to those executed at least once during the test runs. All of the benchmarks were compiled with GNU C++ 2.6.3 with optimization flags "-O4 -msupersparc" and run on a SPARCstation-10 workstation with a 60 MHz SuperSPARC processor. In addition to measuring execution time on an idle machine, we also simulated the programs with an instruction-level simulator to obtain precise instruction counts and cache misses (simulating the SPARCstation-10 cache configuration[2]).

6. Results

This section presents the results of the empirical evaluation of our optimizing compiler. Unless mentioned otherwise, all numbers are dynamic, i.e., based on run-time frequencies.

6.1 Virtual Function Calls

Figure 3 shows that the optimizing compiler successfully removes many virtual function calls. Not surprisingly, the baseline programs execute the same number of virtual calls as the original programs: even though the back-end compiler has the entire program available at compile time, it cannot optimize virtual function calls. In contrast, type feedback is quite successful: on the large

[2] 16Kbyte 4-way primary instruction cache, 20Kbyte 5-way data cache, and 1Mbyte unified direct-mapped secondary cache.

programs, it reduces the number of virtual calls by a factor of 5 (for `idl`, by a factor of 50). On

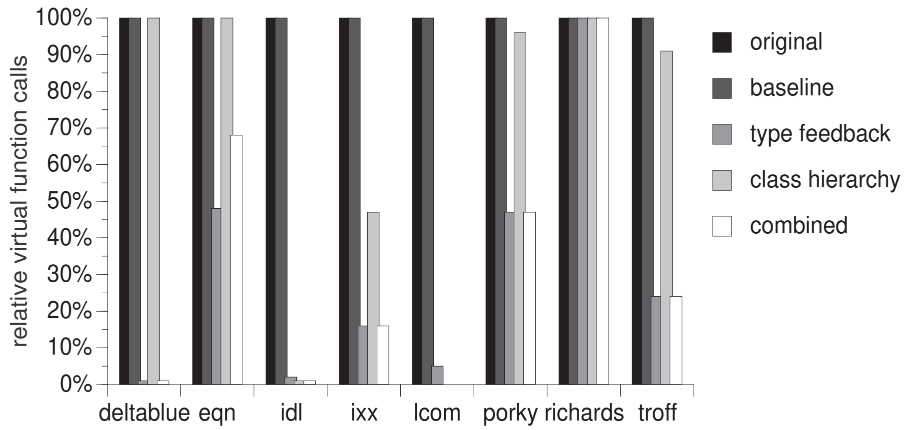

Figure 3. Virtual function calls

some programs, however, type feedback performs relatively poorly. For `richards`, the reason is simple: this program contains only a single virtual function call whose receivers are fairly evenly spread over four classes so that the call cannot be optimized. On `eqn`, type feedback removes less than half of the virtual function calls because some of the most frequently executed call sites are megamorphic, i.e., have many receiver classes (up to 17). Since no receiver class dominates these call sites, type feedback cannot eliminate the calls.

Class hierarchy analysis is much less successful: for many of the large programs, it fails to reduce the number of virtual calls appreciably, removing only a median of 9% of the calls.[3] This poor performance surprised us since others have reported very good results for CHA [DGC95][Fer95]; at first, we suspected an implementation bug. Apparently, most of these programs simply do not have enough monomorphic calls that could be optimized. For example, only 3% of the virtual function calls in `eqn` are from (dynamically) monomorphic call sites, and class hierarchy analysis can optimize only a single call site (a call to `font::handle_unknown_font_command` which is never executed). However, before concluding that class hierarchy analysis is ineffective, the reader should keep in mind that its effectiveness depends on programming style. In particular, CHA should do well in programs using "interfaces" expressed as abstract classes containing only virtual functions because these programs contain many virtual functions with only a single implementation.

The combined system generally performs as well as type feedback, with few exceptions (e.g., `eqn`) where it performs worse. Currently, the combined system chooses class hierarchy analysis over type feedback when optimizing a call site. I.e., if the call can be statically bound, it will do so rather than type-predict it based on profile information. Though it may seem that this system should always perform as well as type feedback (if no calls are statically bound) or better (if some calls can be inlined without a test), this is not true. The reason is somewhat subtle: even though class hierarchy analysis can statically bind (and inline[4]) a call, the inlined version cannot be specialized to a particular receiver class if several classes are possible (all of which inherit the same target method). In contrast, type feedback would predict the most likely receiver class for

[3] lcom cannot be optimized with class hierarchy analysis because it is not type-safe.

[4] The current implementation of CHA does not try to force inlining with an `inline` specifier.

the call and inline a specialized version of the method, possibly removing additional calls (with `this` as the receiver) within that method.

Figure 4 shows the number of virtual function calls performed by the "all virtual" versions of the benchmark programs. As expected, all programs perform significantly more virtual function calls (a median of 5 times more). However, optimization still removes most of them, bringing the number of virtual calls to less than that of the original programs for all benchmarks except richards. Relative to the baseline, type feedback reduces virtual calls by a median factor of 10.3, and class hierarchy analysis reduces them by a factor of 5.0. We will discuss this result further in section 7.

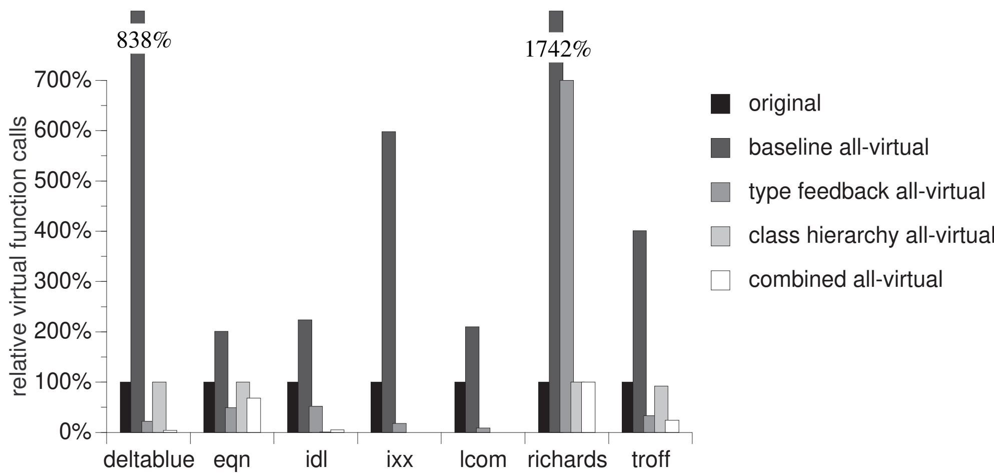

Figure 4. Virtual function calls of "all-virtual" programs

As a final remark, our implementations of both type feedback and class hierarchy analysis currently do not handle virtual operators and virtual destructors, so that such calls are not optimized. Thus, our results are conservative estimates (lower bounds) on the achievable performance.

6.2 Performance

Ultimately, the main goal of optimization is to increase performance. Figure 5 shows the execution time of the various program versions relative to the execution time of the original program. Overall, performance improves considerably relative to the original programs, with a median speedup of 20%. For all programs, type feedback or combined optimization produces the fastest executables whereas class hierarchy analysis does not significantly improve performance, which is not surprising given its ineffectiveness in removing virtual function calls. More surprisingly, about half of the speedup comes from combining all source files into a single file: the baseline programs run an average of 8% faster than the original programs. Why?

The main reason for this speedup is the proper inlining of nonvirtual inline functions in the baseline versions. Many C++ compilers (including GNU C++) do not inline function calls unless the inline function's definition is encountered before the call. In the original programs, function definitions often are not encountered in the correct order, and thus many calls to inline functions are not inlined. In contrast, our compiler topologically sorts all functions in the baseline versions so that all definitions precede their uses if possible (i.e., if there is no recursion).

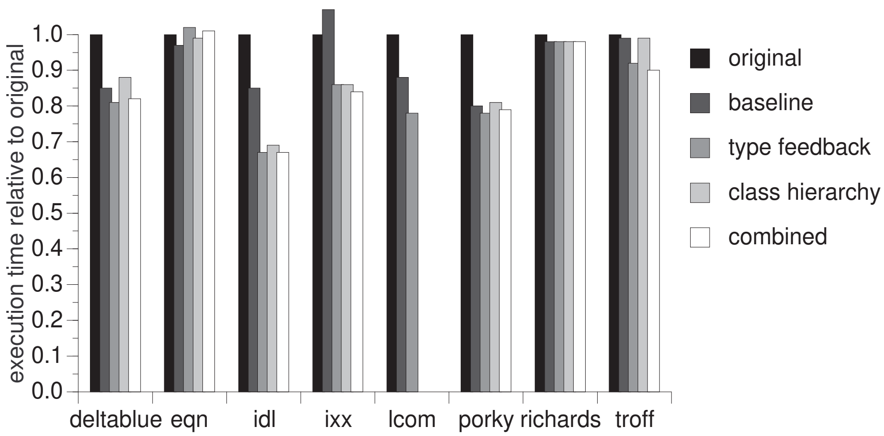

Figure 5. Execution time of optimized programs

We believe that our results are only a lower bound on the performance that could be achieved by a full-fledged optimizing compiler. Four factors contribute to our belief: the back-end compiler, hardware architecture, the set of optimizations our compiler performs, and our choice of benchmarks.

First, the current back-end compiler (GNU C++) does not take advantage of optimization opportunities exposed by inlining. In particular, it does not perform alias analysis, and thus it cannot remove redundant loads of instance variables and thus misses other opportunities for common subexpression elimination (including opportunities to CSE dispatch tests). We are planning to port our optimizer to other back-end compilers (Sun and IBM C++) to investigate the magnitude of this effect.

Second, our results may underestimate the performance impact on CPU architectures more advanced than the 3-year-old SuperSPARC chip used for our measurements. In particular, more aggressively pipelined superscalar CPUs are likely to benefit more from virtual call elimination since the cost of indirect calls tends to increase on such architectures [DHV95]. In fact, this trend is already visible on the SuperSPARC: whereas type feedback reduces the number of instructions executed by a median of only 5%, it reduces execution time by 20%. Clearly, optimization improves the effectiveness of superscalar issue and pipelining. Although further research is needed to resolve this question, we expect the speedups achieved by our system to increase on more recent processors like the UltraSPARC or the Intel P6 (Pentium Pro).

Third, type feedback could be complemented with additional optimizations to improve performance further. In particular, profile-based customization and some form of splitting [CU90] are attractive candidates, although the latter might not be needed if the back-end C++ compiler did a better job of alias analysis.

Finally, some of our benchmarks just don't execute that many virtual function calls to start with. Figure 6 shows that, as expected, speedups correlate well with call frequency: the more frequently a program uses virtual function calls, the better it is optimized. Several of our benchmark programs have a low virtual call frequency; for example, `eqn` executes 465 instructions between virtual function calls but only 73 instructions between direct function calls. We believe that such infrequent use of virtual calls is atypical of current C++ programs. In particular, the use of abstract classes as interfaces in application frameworks is becoming increasingly common and will drive up virtual function call frequencies. Unfortunately, we have been unable to find publicly available programs exhibiting this programming style.

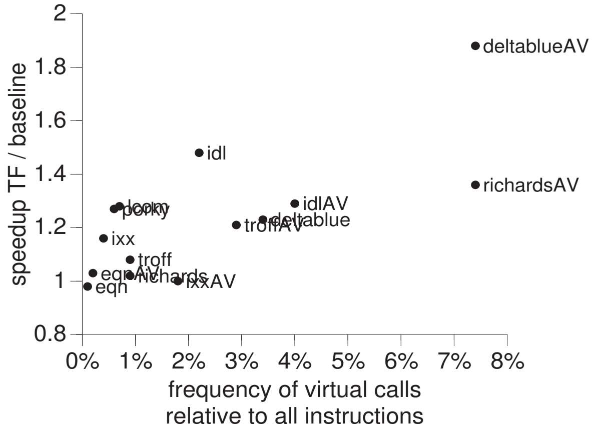

Figure 6. Correlation between call frequency and speedup

Figure 7 shows the performance of the "all-virtual" versions. In one group of programs (deltablue, `idl`, richards, troff), virtual function calls become much more frequent, and as a result the optimized programs achieve higher speedups. In the other group, the call frequency does not change significantly, and thus program behavior remains unchanged. Overall, the median speedup of type feedback increases from 20% to 65%, confirming our observations from section 6.2.

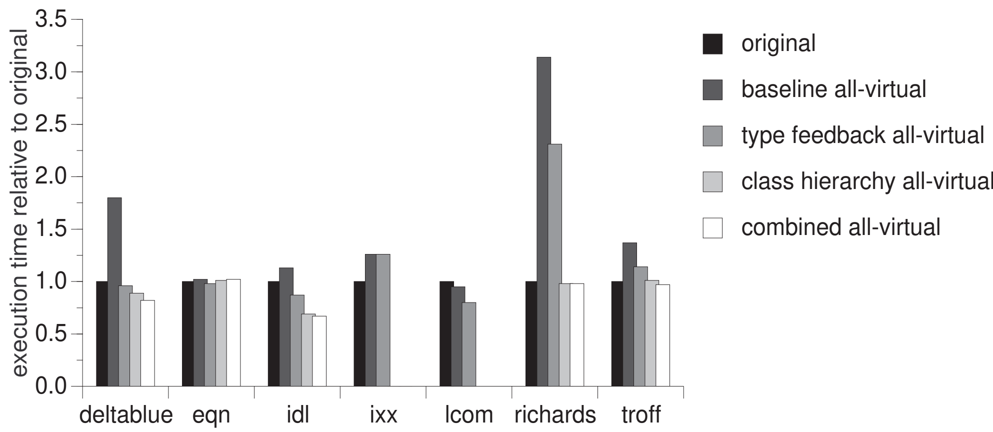

Figure 7. Execution time of "all-virtual" programs

To summarize, despite the relatively low frequency of virtual calls in the benchmarks, our optimizing compiler convincingly demonstrates the value of OO-specific optimization for C++ programs, speeding up a set of realistic applications by a median of 20%. Moreover, with a better back-end compiler or on more recent processors, this speedup is likely to increase even further.

6.3 Program Size

Inlining duplicates some code for (hopefully) better performance; Figure 8 shows the size of the programs before and after optimization. Program size was measured as the size of the text segment (i.e., instructions only, no data), excluding library code. Overall, code size increases only modestly with optimization; programs optimized with type feedback are a median 19% larger than the original programs.

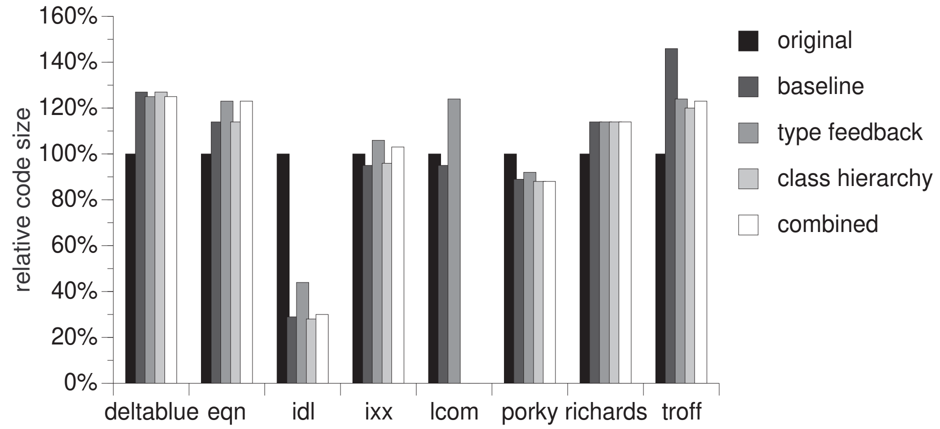

Figure 8. Code size

One program, `idl`, clearly sticks out: in its unoptimized form (baseline), it is three times smaller than the original program. The reason is simple: in the original program, the compiler generates multiple copies of inline functions (in particular, constructors and destructors), one per .C file that uses them. As a result, many inline functions are replicated 20 or more times. This problem still is a common problem with C++ compilers. Typically, compilers use a heuristic to decide when to generate a copy; since `idl` was written using a different compiler, it does not match GNU's heuristics. In the baseline version, of course, no duplication can occur since a single file contains the entire program.

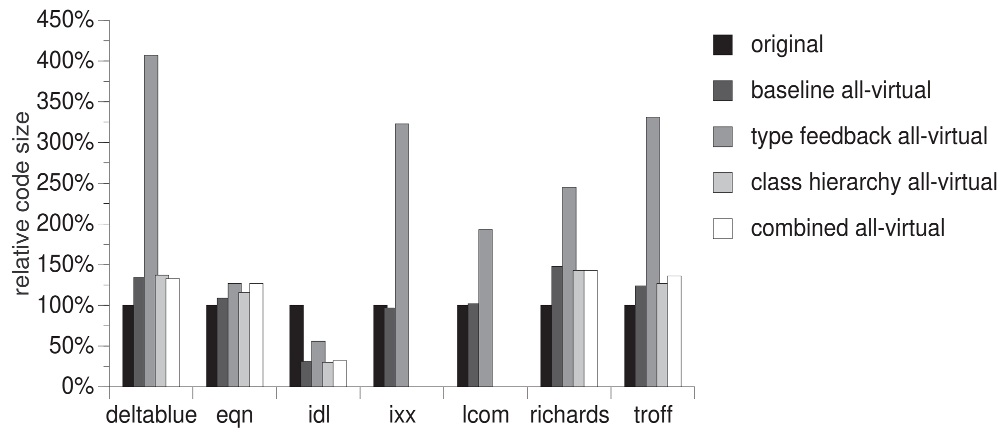

Figure 9. Code size of "all-virtual" programs

In the "all virtual" programs, code size increases more strongly (Figure 9), especially with type feedback in the small programs (`richards` and `deltablue`). However, the data in Figure 9 are for programs compiled without the call site contribution threshold described in section 4.6; i.e., all call sites (not just the frequently-executed ones) are optimized whenever possible. The final paper will contain data for a system where only frequently-executed call sites are optimized. Finally, with class hierarchy analysis, the maximum code expansion drops to a more reasonable 50%.

6.4 Instruction Cache Misses

The main time cost of larger executables lies in the increased instruction cache misses that larger programs can cause. Figure 10 shows the instruction cache misses on a SPARCstation-10 incurred by the optimized programs. Overall, differences are surprisingly small; for some programs, cache misses actually decrease slightly compared to the original programs (`richards`' miss ratio is virtually zero because it fits entirely into the cache). The major exception is `troff`, where misses

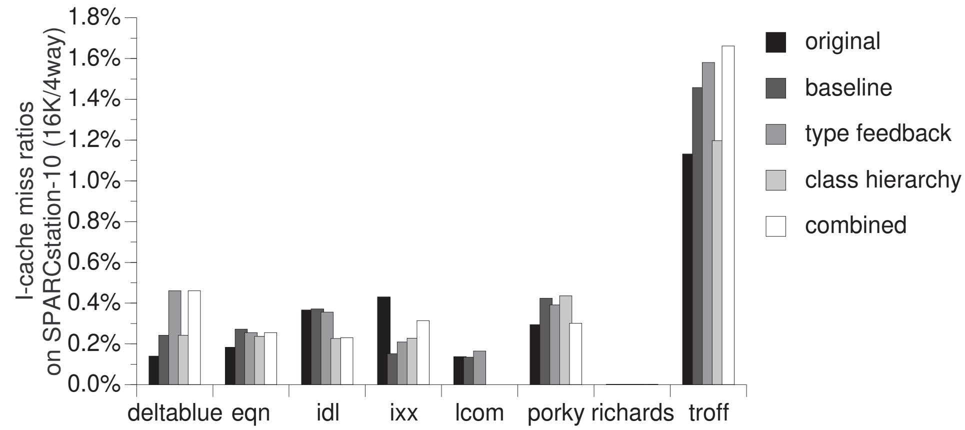

Figure 10. Instruction cache misses

increase by 35% over the original program when using type feedback. However, cache misses increase much less (6%) relative to the baseline program, indicating that the additional misses may be caused by the different relative code placement (i.e., conflict misses) rather than by a systematic effect (i.e., capacity misses). Another program, `deltablue`, also experiences wild swings in its miss ratios (from 0.14% in the original to 0.46% with type feedback) even though its size doesn't change much (see Figure 8). A separate simulation (modelling a 32-way associative cache of the same size as the standard 4-way associative cache) showed little variation between the various systems, confirming that indeed the differences in cache performance are caused by different code placement, i.e., are unrelated to the actual optimizations. Thus, our data show little significant difference in cache behavior between the optimized and original programs.[5]

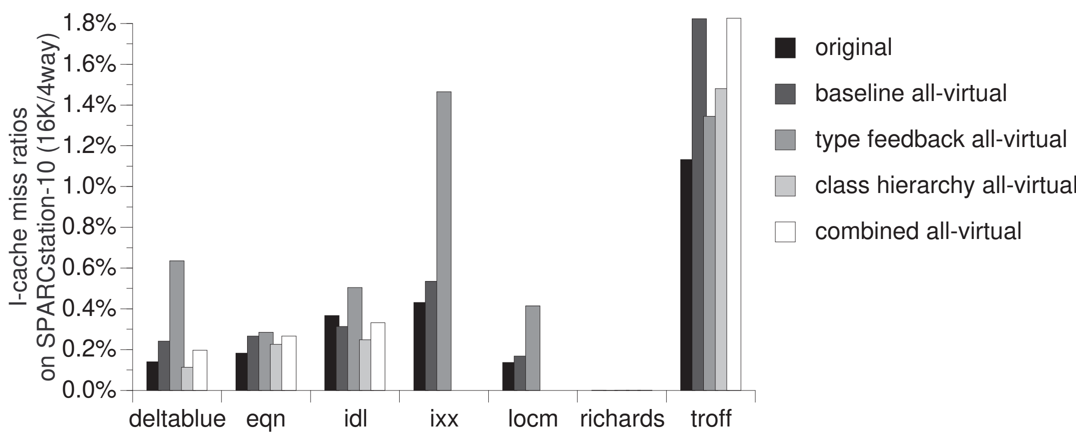

Figure 11. Instruction cache misses of "all-virtual" programs

[5] We have not yet verified whether the differences in the cache behavior of "all-virtual" programs (Figure 11) also result from conflict misses rather than capacity misses.

6.5 Influence of Inlining Strategies

Figure 12 shows the average of the four performance characteristics (time, virtual calls, size, and cache misses) as a function of the inlining threshold. Recall from section 4.5 that the compiler inlines a virtual call only if the most frequent case exceeds a certain threshold. For example, the data points at x = 60% in Figure 12 represent the average performance of programs compiled with a threshold of 60%. In general, the lower the threshold, the more inlining the compiler performs, and the larger the optimized executables become. For the programs in our benchmark suite, the "sweet spot" appears to be near a threshold value of 60%; below that, there is little improvement in performance whereas cache misses and executable size increase significantly. For that reason, we chose 60% as the default threshold in our compiler even though a lower value would have improved performance further by a few percent.

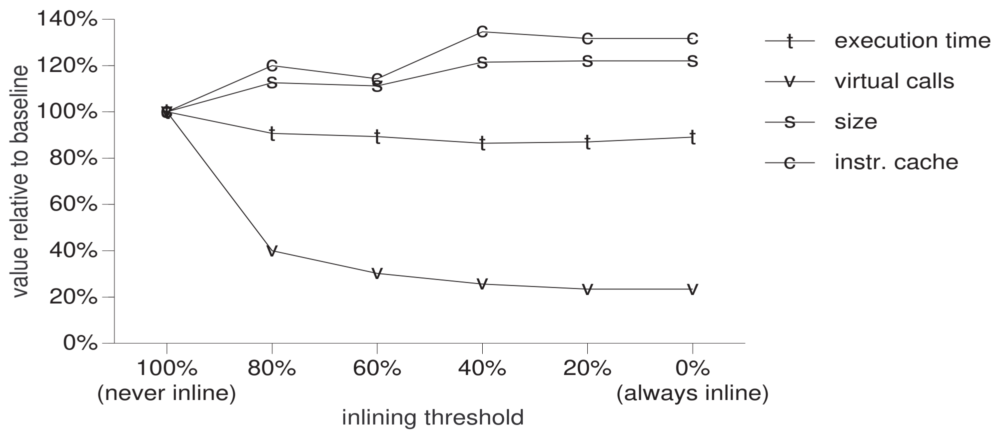

Figure 12. Performance characteristics as a function of inlining threshold (averages over all programs)

7. Discussion

One of the unintended effects of language implementations is that they can shape programming style. If a certain language construct is implemented inefficiently, programmers will tend to avoid it. Optimization of virtual function calls effectively lowers their average cost, and thus might change the way typical C++ programs are written. Figure 13 compares the original programs against the best optimized version (usually compiled by the "combined" system) and shows that optimization indeed enables the programmer more liberally: for each of the benchmarks (except for the artificial `richards`) optimization significantly lowers the number of virtual function calls actually executed at run time. Even the "all virtual" programs, which in their unoptimized form execute five times *more* virtual function calls than the original programs, perform a median of four times *fewer* calls when optimized. Similarly, all optimized programs, even the "all virtual" versions, execute faster than the original programs compiled with conventional optimization (Figure 14).

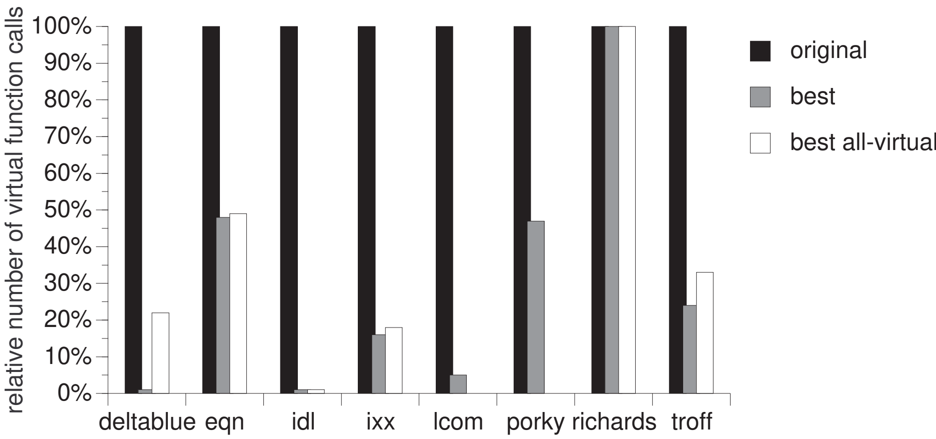

Figure 13. Virtual function calls

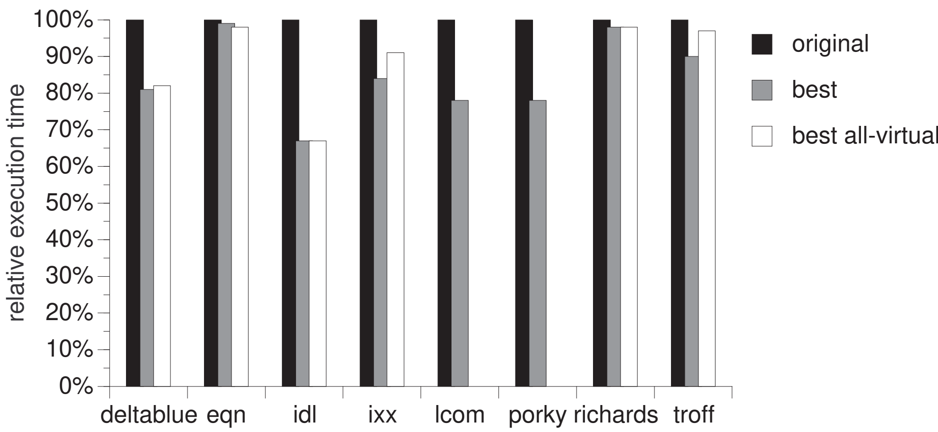

Figure 14. Execution time

In other words, even if the authors of these programs used virtual functions much more liberally (e.g., in order to make their programs more flexible and extensible), they would not have been penalized by inferior performance.

8. Future Work

Several areas for future work remain. First, we would like to investigate the impact of back-end optimizer quality by porting our compiler to another back-end C++ compiler that performs more optimizations. As mentioned above, we believe that a stronger back end would increase the performance gap between the original and optimized programs, but further measurements are needed to substantiate this hypothesis.

The compiler's inlining strategies could also be improved. A simple extension of our system would inline more than one case per call site (e.g., the two most frequent cases). Inlining should probably take into account the size of the inlinee [Höl94], and the compiler should estimate how much the inlinee's code can be simplified (i.e., because of constant arguments). Furthermore, type feedback should be extended with profile-driven customization to further improve performance. Profiling could be extended to use k-CCP profiles (i.e., take call chains into account), although the improvement from the additional precision may be small [G+95].

Also, a more detailed investigation of the interaction of optimization with superscalar architectures is needed. Modern processors are increasingly deeply pipelined, contain multiple execution units, and can execute instructions out-of-order or speculatively. All of these features can significantly impact performance, and further study is needed to determine their impact on the performance of object-oriented programs.

Finally, we will continue to look for other C++ applications that can be used for benchmarking. Although we are already using large programs totalling over 90,000 lines of code, we feel that currently available benchmarks (including those used in other studies [CGZ94]) do not represent the entire spectrum of program characteristics. In particular, programs using large class libraries and frameworks are underrepresented. Fortunately, these programs are very likely to benefit even more from optimization (as discussed in section 6.2), and thus this underrepresentation does not invalidate the results of our study.

9. Conclusions

We have designed and implemented an optimizing source-to-source C++ compiler. To the best of our knowledge, this compiler is the first C++ compiler able to reduce the frequency of virtual function calls. Our prototype implementation demonstrates the value of OO-specific optimizations for a statically-typed, hybrid object-oriented language like C++. Using a suite of two small and six large C++ applications totalling over 90,000 lines of code, we have evaluated the compiler's effectiveness. Despite some limitations of our system, and despite the low frequency of virtual function calls in some of the programs, optimization improves the performance of these C++ applications by up to 48% over the original programs (median: 20%) and reduces the number of virtual function calls by a median factor of five. For "all-virtuals" versions of the same programs, performance improved by a median of 40% and the number of virtual calls dropped by a factor of more than 20.

Our measurements produced some surprising results:

- On the original programs (but not on the "all-virtuals" programs, class hierarchy analysis is ineffective, contrary to the results previously published for Modula-2 programs [Fer95] or the pure object-oriented language Cecil [DGC95].
- Inlining does not necessarily lead to large increases in code size. On average, optimized programs only expand by 19%. Moreover, this code expansion does not impact performance much; for most programs, the instruction cache miss ratio does not increase significantly, and for some programs it even decreases.

We believe that our results underestimate the performance gains that could be obtained with a production-quality compiler. In other words, we believe that typical C++ programs can be sped up by more than the 20% improvement seen here. Several reasons lead us to this conviction:

- Our compiler uses source-to-source optimization for simplicity, which to some extent negatively impacts the quality of the generated code since the front-end cannot communicate all its information to the back end. Furthermore, our current back end (GNU C++) does not remove some of the redundancies exposed by inlining. In particular, better alias analysis could help remove redundant loads and type tests.
- The hardware platform used in our measurements (a SuperSPARC processor) probably benefits less from optimization than more recent aggressively pipelined processors. (This question remains an area for future study, however.)
- Several of our benchmark programs have a relatively low virtual function call frequency and thus benefit less from optimization. Programs using abstract base classes may be significantly more amenable to optimization since they will use virtual function calls more frequently.

If these optimizations are integrated into production compilers, programmers therefore can hope to see even better speedups on typical programs.

Finally (and perhaps most importantly), our results show that programmers may use virtual functions much more liberally in order to make their programs more flexible and extensible without being penalized by inferior performance.

Acknowledgments: This work was supported in part by MICRO Grant 95-077, IBM Corporation, and Sun Microsystems. The first author was partially supported by a Kurt Goedel Fellowship. We would like to thank Jerry Schwarz for making the CCAuditor parser available to us and for answering our many questions. Finally, thanks to Ole Agesen, Lars Bak, Karel Driesen, and Robert Griesemer for their comments on earlier versions of this paper.

References

[Age95] Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *ECOOP'95, Ninth European Conference on Object-Oriented Programming,* p. 2-26, Århus, Denmark, August 1995. Springer-Verlag LNCS 952.

[AH95] Ole Agesen and Urs Hölzle. Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages. In *OOPSLA'95, Object-Oriented Programming Systems, Languages and Applications,* p. 91-107, Austin, TX, October 1995.

[App88] Apple Computer, Inc. *Object Pascal User's Manual*. Cupertino, 1988.

[CGZ94] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying Behavioral Differences Between C and C++ Programs. *Journal of Programming Languages 2*:313-351, 1994.

[CG94] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. In *21st Annual ACM Symposium on Principles of Programming Languages*, p. 397-408, January 1994.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89, Object-Oriented Programming Systems, Languages and Applications*, p. 49-70, New Orleans, LA, October 1989. Published as *SIGPLAN Notices 24(10)*, October 1989.

[CU90] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, p. 150-164, White Plains, NY, June 1990. Published as *SIGPLAN Notices 25(6)*, June 1990.

[CM+92] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience* 22 (5): 349-369, May 1992.

[CHT91] K. D. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software—Practice and Experience* 21 (6): 581-601, June 1991.

[DH88] Jack W. Davidson and Anne M. Holler. A study of a C function inliner. *Software—Practice and Experience* 18(8): 775-90, August 1988.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP'95, Ninth European Conference on Object-Oriented Programming,* Århus, 1995. Springer-Verlag LNCS 952.

[Dea95] Jeffrey Dean. New Cecil performance numbers. Private communication, May 1995.

[Dec95] Declarative Systems. The C++ Auditor. Palo Alto, 1995. (auditor@declarative.com)

[DHV95] Karel Driesen, Urs Hölzle, and Jan Vitek. Message Dispatch On Pipelined Processors. *ECOOP '95 Conference Proceedings*, Aarhus, Denmark, August 1995. Published as *Springer Verlag Lecture Notes in Computer Science 952*, Springer Verlag, Berlin, 1995.

[DS84] L. Peter Deutsch and Alan Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, p. 297-302, Salt Lake City, UT, January 1984.

[Deu87] L. Peter Deutsch. Reusability in the Smalltalk-80 system. *Workshop On Reusability In Programming*, Newport, RI, 1983.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[Fer95] M. F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, p. 103-115, La Jolla, CA, June 1995. Published as *SIGPLAN Notices 30(6)*, June 1995.

[G+95] David Grove, Jeffrey Dean, Charles D. Garrett, and Craig Chambers. Profile-Guided Receiver Class Prediction. In *OOPSLA'95, Object-Oriented Programming Systems, Languages and Applications,* p. 108-123, Austin, TX, October 1995.

[Hall91] Mary Wolcott Hall. *Managing Interprocedural Optimization.* Technical Report COMP TR91-157 (Ph.D. Thesis), Computer Science Department, Rice University, April 1991.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *ECOOP'91, Fourth European Conference on Object-Oriented Programming,* p. 21-38, Geneva, July 1991. Springer-Verlag LNCS 512.

[Höl94] Urs Hölzle. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming*. Ph.D. Thesis, Department of Computer Science, Stanford University, August 1994. (Available via http://www.cs.ucsb.edu/~urs.)

[HU94a] Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, p. 326-336. Published as *SIGPLAN Notices 29(6)*, June 1994.

[HwC89] W. W. Hwu and P. P. Chang. Inline function expansion for compiling C programs. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, p. 246-57, Portland, OR, June 1989. Published as *SIGPLAN Notices 24(7)*, July 1989.

[Knu70] Donald Knuth. *An empirical study of FORTRAN programs*. Technical Report CS-186, Department of Computer Science, Stanford University, 1970.

[Lea90] Douglas Lea. Customization in C++. In *Proceedings of the 1990 Usenix C++ Conference*, p. 301-314, San Francisco, CA, April, 1990.

[PR94] Hemant D. Pande and Barbara G. Ryder. *Static Type Determination and Aliasing for C++*. Technical Report LCSR-TR-236, Rutgers University, December 1994.

[PC94] John B. Plevyak and Andrew A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. *OOPSLA '94, Object-Oriented Programming Systems, Languages and Applications*, p. 324-340, Portland, OR, October 1994. Published as *SIGPLAN Notices 29(10)*, October 1994.

[Ser95] Mauricio Serrano. Virtual function call frequencies in C++ programs. Private communication, 1995.

[SW92] Amitabh Srivastava and David Wall. *A Practical System for Intermodule Code Optimization at Link- Time*. DEC WRL Research Report 92/6, December 1992.

[US87] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *OOPSLA '87, Object-Oriented Programming Systems, Languages and Applications*, p. 227-241, Orlando, FL, October 1987. Published as *SIGPLAN Notices 22(12)*, December 1987. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June 1991.

[Wall91] David Wall. Predicting Program Behavior Using Real or Estimated Profiles. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, p. 59-70, Toronto, Canada, June 1991. Published as *SIGPLAN Notices 26(6)*, June 1991.