AN ENABLING OPTIMIZATION FOR C++ VIRTUAL FUNCTIONS

Bradley M. Kuhn[*] bkuhn@acm.org

David W. Binkley[*] binkley@cs.loyola.edu

Computer Science Department
Loyola College in Maryland
4501 N. Charles Street
Baltimore, Maryland 21210-2699

Keywords Virtual Functions, Optimization, Dynamic Binding

ABSTRACT

Gaining the code re-use advantages of object oriented programming requires dynamic function binding, which allows a new subclass to override a function of a superclass. Dynamic binding is obtained in C++ through the use of virtual functions. Unfortunately, virtual functions have two negative impacts on performance. First, they are traditionally compiled into indirect call instructions, which take longer to execute than direct call instructions. Second, it is difficult for the compiler to perform optimization since summary information from called procedures is hard or even impossible to obtain. The net effect is that C++ programmers avoid the use of virtual functions.

We present a new optimization that (1) removes the indirect function calls used for virtual functions, (2) *enables* other compiler optimizations such as inlining and constant propagation, and (3) requires no extensive data-flow analysis or profile information and thus is easily implemented in an existing compiler. We include experimental evidence that this optimization reduces execution time. Not surprisingly, the greatest benefits are obtained with programs that include a high proportion of virtual functions calls.

[*]supported in part by the National Science Foundation under grant CCR-9411861

1 INTRODUCTION

Compiler optimizations that attempt to reduce program execution time have become increasingly more aggressive. Modern compilers perform a variety of intraprocedural and interprocedural optimizations (*e.g.*, branch prediction, code motion, constant propagation, and subprogram inlining). To safely apply these optimizations requires summary information from called procedures; thus, a call to a function that cannot be resolved at compile time will invalidate or severely restrict optimization. C++ virtual functions produce such calls. This is unfortunate, as many functions in object-oriented programs are small and therefore excellent candidates for inlining. Because optimization is hampered by virtual functions, programmers hesitate to use them.

Reducing the cost of virtual functions is becoming increasingly important as programmers become more familiar with object-oriented design and programming techniques and, consequently, include more virtual functions in their code. For example, recent versions of the Interviews framework use more virtual functions and virtual function calls that a previous version [9].

This paper describes an optimization that improves the performance of C++ programs that contain virtual functions. The optimization is an *enabling* optimization: it attempts to enable other optimizations by replacing indirect function calls with direct functions calls. This allows existing compiler optimizations to be more effective. For example, enabling inlining can increase basic block size, which makes it easier to keep multiple pipelined functional units busy in today's modern super-scalar pipelined architectures such as the Pentium and powerPC. Finally, unlike other optimizations for the same problem [3, 10], our optimization requires no extensive data-flow analysis or profile information.

The next section describes our optimization. Following this, Section 3 presents performance data obtained by applying our optimization to a collection of C++ programs. Next, Section 4 describes related work, and finally, Section 5 provides a summary of our results.

2 ENABLING OPTIMIZATION

This section describes our *enabling* optimization, discusses the optimization's implementation, and provides examples showing how it interacts with (enables) inlining and constant propagation. The code shown in Figure 1 is used in this section to illustrate various aspects of the optimization. Note that the virtual function call in function `process()` does not call the version of `bar()` declared in class `Base` as would be the case with static binding. Rather, the function called is determined at runtime by the type of the object pointed to by `x`, which may be `Base` or a subclass of `Base`. For more details on virtual functions see [6].

2.1 The Enabling Optimization

Our optimization replaces each dynamic function call with a switch statement and a set of static function calls. Since existing compilers can analyze static functions calls, the compiler is in a better position to perform other optimizations after our optimization is applied. The replacement has the following steps:

Step 1. Compute the class hierarchy to determine the set of functions that each virtual function call may invoke.

Step 2. Replace each virtual function call with a switch statement and a set of static function calls. The switch statement uses the type of the receiving object to determine which function is called. The set of static calls used in the switch statement is determined by the following three rules:

- **Rule 1.** Based on the declared type of a pointer in C++, the set of functions that may be invoked by a virtual function call-site is limited. For example, consider a virtual function call to function `f` made though a pointer to class `C`. This call may result only in a call to the version of `f` declared in class `C` or any class derived from `C`.
- **Rule 2.** Classes that call the same virtual function are consolidated. For example, if `B` and `C` both derive from `A`, and `B` and `C` do not implement their own version of `A::f`, then `A`, `B`, and `C` all call `A::f`. Putting all three classes into one branch of the case statement shortens the code, and makes the compiler's jump table smaller.
- **Rule 3.** Any class that contains pure virtual functions can be eliminated from the switch statement. C++ forbids the creation of objects of a class that contains a pure virtual function; thus, there will never be a call to a function in such a class.

It is possible that, after applying these rules, only a single case remains. If so, we replace the switch statement with a single static call.

Example. Figure 2 shows the transformed functions `Base::bar()`, `Sub1::bar()`, and `process()` from Figure 1. Using Rule 2, the cases for `Base` and `Sub2` in function `process()` are consolidated because `Sub2` does not implement its own version of `bar()`. Also, since `Sub1` has no subclasses, Rule 1 leaves a case statement with only one arm. After further optimization, this becomes the static call `sub1::foo()`.

Like many interprocedural optimizations, ours is most effective when the entire source is analyzed at one time. Complete source allows the optimization to obtain complete information about class hierarchies and the use of virtual functions. It is possible to optimize procedures compiled separately, with some loss in the effectiveness. It is also possible to perform the optimization at link time; however, some local optimizations, such as constant propagation, that we hope to enable would be missed. An alternative is the use of a programming environment that incrementally computes summary information for large systems (often as a background task); thus, reducing the need to examine the entire program on each compile without sacrificing the effectiveness of the optimization. Such an environment for SELF programs is described in [8].

2.2 Implementation

The optimization is presently implemented as a source-to-source transformation that essentially applies the steps and rules given above. The three rules are currently implemented by hand. Implementing the optimization as a source-to-source transformation was intentional; it allows us to examine the resulting source code, which has led to several improvements in the three filtering rules. An internal (to the compiler) implementation should produce more efficient code and thus slightly better performance results.

```
class Base {
    protected:
       int  val;
    public:
      virtual foo() { val = 1; };
      virtual bar();
};

Base::bar() {
        foo();
        printf("%d\n", val);
}

void process(Base * x) {
        ....
        x->bar();
        ....
}
```

```
class Sub1 : public Base {
    public:
      virtual foo() { val = 2; };
              inc() { val++;   };
      virtual bar();
};

Sub1::bar() {
    inc();
    foo();
    printf("%d\n", val);
}

class Sub2 : public Base {
    public:
      virtual foo() { val = 3; };
}
```

Figure 1: A C++ Class Hierarchy

The only interesting part in the construction of the switch statements is the implementation of the function `typeof`. To implement `typeof`, each class is assigned a unique type code. This involves adding a new integer attribute, `type-of`, to each class. Class constructors assign the class' type code to this attribute when objects are created. The function `typeof()` simply inspects it to determine the type of an object. The algorithm requires only that each class have a unique type code; consequently, multiple inheritance poses no additional difficulties. As illustrated below, constant propagation of this attribute is one of the optimizations we hope to enable.

The implementation requires two modifications to handle separate compilation. First, to ensure that the same `type-of` attribute is assigned to a class when it appears in two different modules, the translator maintains a mapping from class names to `type-of` attributes. Second, when the complete class hierarchy is not available, a default case is added to each switch statement. The default case uses the unmodified virtual function call. For example, if the entire source were not available, the optimized switch statement for `Base::bar()` from Figure 2 would be the following:

```
switch(typeof(this)) {
    case Base:
      Base::foo(); break;
    case Sub1:
      Sub1::foo(); break;
    case Sub2:
      Sub2::foo(); break;
    default:
      this->foo(); break;
      // ‘this->’ is optional
}
```

Adding a default case restricts existing optimizations, but allows incremental program development. As with any interprocedural optimization the quality of the optimization improves when the entire source is available. Once development is complete, the entire program can be compiled without need for the default cases.

2.3 Examples

Our optimization increases the possibilities for inlining by turning dynamic function calls into static ones. This allows the compiler to match each call-

```
Base::bar() {                          Sub1::bar() {
     switch(typeof(this)) {                Sub1::foo();
       case Base:                          inc();
         Base::foo(); break;               printf("%d\n", val);
       case Sub1:                        }
         Sub1::foo(); break;
       case Sub2:
         Sub2::foo(); break;
     }
     printf("%d\n", val);
}
                      void process(Base * x) {
                          ...
                          switch(typeof(x)) {
                              case Base:
                              case Sub2:
                                  x->Base::bar(); break;
                              case Sub1:
                                  x->Sub1::bar(); break;
                          }
                          ...
                      }
```

Figure 2: Code from Figure 1 After Optimization

site with a single (static) procedure. For example, inlining has the following affect on `Base::bar()`:

```
Base::bar() {
    switch(typeof(this)) {
        case Base:
          val = 1; break;
        case Sub1:
          val = 2; break;
        case Sub2:
          val = 3; break;
    }
    printf("%d\n", val);
}
```

Since the resulting code for `bar()` is short, it is a candidate to be inlined into other functions such as `process()`.

The inlining we enable allows the compiler to perform other optimizations such as code motion. For example, Figure 3 shows Pentium assembler output generated by GNU C++ for geqn (geqn and GNU C++ are discussed in Section 3). Notice that in the second column of the figure, the instructions "`pushl $0`" and "`pushl %esi`" are moved from each case to just before the "`jmp *L963(,%eax,4)`" instruction.

In addition to inlining, our optimization enables constant propagation by providing opportunities for both interprocedural and intraprocedural constant propagation. In particular, we hope to enable the propagation of class types. We illustrate this with two examples. First, suppose that the code in Figure 4a is added to the program in Figure 1. The result of our optimization and inlining on this code is shown in Figure 4b. The type of `s1` can be propagated to the switch statement. For this example, `s1`'s type is a constant as it has no derived classes. (In general, knowing a type allows the number of cases to be pruned.) In the resulting code, shown Figure 4c, our optimization, inlining, and constant propagation have combined to transform a dynamic call into a static call.

As a second example, consider inlining the two calls to `bar()` in function `process()` shown in Figure 2. Because our optimization introduces switch statements into `bar()`, the resulting code contains nested switch statements. Within each case of the outer switch statement, the `typeof(x)` is constant. Constant propagation of this value allows the nested switch statements to be flattened; after inlining the calls on `foo()`, and performing code motion, which moves the call to `printf`, `process()` becomes

```
        Before Code Motion                        After Code Motion

        ...                                     ...
        movl (%esi),%eax                        movl (%esi),%eax
                                                pushl $0
                                                pushl %esi
        jmp *L963(,%eax,4)                      jmp *L963(,%eax,4)
L963:                                     L963:
        ... jump table ...                       ... jump table ...

L932:                                     L932:
        pushl $0                                 call _check_tabs__9delim_boxi
        pushl %esi                               jmp L1048
        call _check_tabs__9delim_boxi     L933:
        jmp L1048                                call _check_tabs__9limit_boxi
L933:                                            jmp L1048
        pushl $0                          L934:
        pushl %esi                               call _check_tabs__8list_boxi
        call _check_tabs__9limit_boxi            jmp L1048
        jmp L1048                                ...
L934:
        pushl $0
        pushl %esi
        call _check_tabs__8list_boxi
        jmp L1048
        ...
```

Figure 3: Enabling Code Motion

```
{                       {                                       {
    Sub1 *s1;               Sub1 *s1;                               Sub1 *s1;
    ...                      ...                                    ...
    process(s1);            switch(typeof(s1)) {                    s1->Sub1::bar();
}                               case Base:                      }
                                  s1->Base::bar(); break;
                                 case Sub1:
                                  s1->Sub1::bar(); break;
                                case Sub2:
                                  s1->Sub2::bar(); break;
                            }
                        }
        (a)                     (b)                                     (c)
```

Figure 4: Enabling Constant Propagation

```
void process(Base * x) {
      ...
      switch(typeof(x)) {
        case Base:
            val = 1; break;
        case Sub1:
            val = 2;
            val++; break;
        case Sub2:
            val = 3; break;
      }
      printf("%d\n", val);
      ...
}
```

Further constant propagation within the case for `Sub1` allows the cases for `Sub1` and `Sub2` to be combined.

3 EXPERIMENTAL DATA

This section reports our optimization's effect on code size and execution time. We applied our optimization to the C++ programs listed in Table 1. The original and optimized programs were each complied on two machines: a DECStation 5000 with a MIPS R3000 33Mhz processor running Ultrix 4.2A (the DEC machine), and a Gateway 2000 with a Pentium 90Mhz processor running Linux 1.0.9 (the Intel machine). Each program was compiled with GNU's C++ compiler (version 2.3.1 on the DEC machine and version 2.5.8 on the Intel machine) using the `-O3` optimization level[1], which does (limited) interprocedural optimization, (limited) inlining, and substantial intraprocedural optimization.

The standard Unix time utility was used to collect the number of CPU seconds used by each program. Each run was done when the machine was unloaded, so it received an average of 99.9% of the CPU time. As a result, elapse time and CPU time were virtually identical.

The choice of execution time—a direct measure of performance—rather than an indirect measure of performance such as instruction count is intentional.[2] Pipeline effects, cache effects, and the like impact the average number of cycles-per-instruction, which, in turn, impact the computed execution time [7]. This occurs in our last example where the computed execution times before and after our optimization are nearly identical while the observed execution times are not.

[1]The amount of interprocedural optimization actually performed by gcc on the Intel machine is suspect since the `-O2` optimization level produced the same object code as the `-O3` optimization level for each of our test programs.

[2]Execution time can be computed as the product of instruction count, average cycles-per-instruction, and cycle-time.

Each program was run thirty times at random physical memory locations to account for cache effects (see [2] for a description of why this is necessary and the pitfalls associated with not accounting for memory location induced cache effects when timing programs on modern architectures). We report in Table 2 the average times for the thirty runs including the margin of error and the object code size before and after optimization. Note that the percentage growth in the resulting binary files (which include library code) was considerably less: the first two programs showed essentially zero growth, while geqn showed about 50% growth. We now discuss each program in more detail.

The city simulation simulates traffic flow on the roads of a city. The program has only one virtual function and was chosen to illustrate the effects on programs that contain very few virtual functions. On the Intel machine, our optimization caused about a 4% decrease in execution time and a nominal 1% increase in object code size. On the DEC machine, the optimization made only a nominal difference in the execution time. However, it is interesting to note that it decreased the object code size. This provides indirect evidence that our optimization is enabling other optimizations.

The employee program, based on code from [5], has a high proportion of virtual functions and virtual function calls; it was chosen to provide expected results for programs that use virtual functions heavily. The program has a class hierarchy of eight classes and virtual functions that (1) print the employee's data (name, current wages, etc.), (2) report the employee's earnings for the current week, (3) give the employee a raise, and (4) reset the employee for a new week. Each type of worker implements these functions differently.

We created a test input file with 1000 employees (200 of each kind), and then iterated for 300 weeks, printing the data and the earnings for each employee each week and giving each employee a raise every ten weeks. For this example our results were very promising. On the Intel machine, we achieved an almost 4% decrease in execution time with only a 10% increase in code size. On the DEC machine, we achieved slightly over 4% decrease in execution time with a similar 10% increase in code size.

Finally, geqn is the equation formatter distributed with GNU's groff text formatting package. It was chosen for its size and to give results for a large "real world" program. The code contains 6400 source lines of which 400 are virtual function calls. Geqn was written with a single class (called `box`) from which its other seventeen classes are derived. Class `box` declares fourteen virtual functions

Program	**Description**
city simulation	A toy example that simulates cars, trucks, roads, and stop lights.
employee database	An example with different kinds of workers (*i.e.*, hourly, salary, by-piece, ...)
geqn	The equation formatter from the Groff text formatting package.

Table 1: Descriptions of Programs Used for Data Collection

	Size			**Time**		
Program	**Unopt**	**Optimized**	**Change**	**Unoptimized**	**Optimized**	**Change**
Intel:						
city simulation	12618	12762	1.14%	9.650 ± 0.067	9.235 ± 0.074	-4.29%
employee	7867	8715	10.78%	95.500 ± 2.706	90.978 ± 1.268	-4.74%
geqn	132331	264739	100.06%	51.265 ± 0.170	52.610 ± 0.156	2.62%
Dec:						
city simulation	25348	24052	-5.11%	41.865 ± 0.874	41.621 ± 0.766	-0.58%
employee	21852	24056	10.09%	295.613 ± 11.606	282.953 ± 13.364	-4.28%
geqn	302868	586416	93.62%	101.884 ± 0.162	85.878 ± 0.248	-15.71%

Size represents object code size in bytes. **Time** represents CPU time in seconds.

Table 2: Data Obtained with GNU C++'s optimizer enabled `-O3`

(on average only 7 are overridden). This class hierarchy leads to very large inlined functions and consequently large growth in code size. Better object-oriented analysis and design could lead to better class organization, and this explosive code size increase should be averted.

We used a large (3.4 MB) input file to obtain our data. For the Intel machine, our optimization provided no help. In contrast, on the DEC machine, we obtained a significant improvement (15%). The explanation lies in the cache sizes on the two machines. The Pentium processor in the Intel machine has separate 2-way associative 8K code and data caches. In the first two test programs, the size of the optimized program was small enough that cache conflicts were minimal. With geqn, the large code size leads to excessive cache misses; thus, leading to an increase in execution time. The MIPS processor in the DEC machine has a larger 64K cache. Thus, the optimized version of geqn did not encounter a restrictive number of cache misses.

The experimental results shown in Table 2 represent a promising step towards the efficient compilation of C++ programs containing virtual functions. Since our optimization enables inlining, we expected a code size increase. This was not a problem except for the largest program running on the Intel machine, which has the smallest cache. While execution time reduction was not universal, the employee program with its high percentage of virtual function calls showed a consistent performance improvement. Improved compiler optimization, for example interprocedural constant propagation (of types), should improve these results.

4 RELATED WORK

Reducing the cost of indirect function calls generated by a C++ compiler is addressed by Calder and Grunwald [3], and Pande and Ryder [10]. Similar optimization goals have been studied for the languages SELF [8] and Cecil [4]. In contrast to our work, which attempts to reuse optimizations already implemented as part of the compiler, previous techniques perform new analyses. The C++ techniques are complimentary. In particular, the technique of Pande and Ryder may be useful in providing heuristics that estimate the best approach for optimizing a given virtual function call.

Calder and Grunwald investigated using branch prediction to reduce the cost of indirect function calls. Branch prediction attempts to predict the direction taken by a branch instruction based on previous executions of the same branch instruction. This optimization can be done in software [1] or in hardware [7]. Calder and Grunwald focus on two techniques: the first is a 2-bit branch target buffer (a small cache which records the direction that the branch last took). The second uses static profile-driven prediction, which stores information on the directions that branches took in a previous execution of the program. In subsequent compilations this profile information is used to predict branch directions. Their results indicate that the first method tends to be very expensive while the second holds some promise.

Pande and Ryder investigated type determination at the call site to statically choose the correct virtual function [10]. Their algorithm attempts to find the type of an object (class instance) at each virtual function call site. The algorithm, based on an interprocedural control flow graph, traces pointer variables and type information through the graph and attempts to match up static type information with pointer variables. If the system can make a definite match between a pointer and its type, the virtual functions called using the pointer can be statically determined. Pande and Ryder are currently gathering data from their implementation to determine its practicality. In contrast, our optimization hopes to enable existing constant propagation (of class types) to determine essentially the same information without requiring a separate optimization.

Hölzle describes several techniques for improving the efficiency of virtual-function lookups[3] in SELF: a dynamically typed object-oriented programming language in which every operation (including assignment) involves a virtual-function lookup [8]. Hölzle's techniques include an expandable per-call-site cache that is updated on the fly and a runtime system that dynamically compiles code similar to the code described in Section 2. These techniques work well for SELF programs because executing virtual-function lookups is an expensive operation in a dynamically typed language, where standard dispatch tables, used in statically typed languages, cannot be employed. Hölzle is able to bring the performance of SELF programs to within a factor of two of the comparable C++ program. Unfortunately, many of these techniques are too costly to be applied in a statically typed language such as C++, where dispatch tables already provide an efficient implementation.

Dean *et al.* describe a profile driven system for specializing virtual function calls in Cecil programs [4]. Comparison of run-time improvements are difficult as they apply to different languages. Dean *et al.* state that they are currently working on a C++ implementation, but presently have no data. In comparison with our work, they appear to get less code size expansion at the cost of using run-time profile information. This information is used to selectively specialize heavily used virtual functions while ignoring (not creating a copy of the function for) seldom used functions. Our approach lacks such data and therefore cannot make such decisions. The cost for this improvement is an increase in compiler complexity and compile time.

[3]We use the C++ terminology in this paper rather than introduce the corresponding SELF terminology.

5 SUMMARY

This paper has presented a low overhead enabling optimization for C++ programs that contain virtual functions. This optimization replaces each dynamic function call with a switch statement and a collection of static function calls. Most compilers are better able to analyze programs with static function calls; thus, our optimization enables other optimizations.

Our experimental data are promising for a prototype implementation. We obtained a consistent performance improvement except on the largest program running on the machine with the smallest cache. The numbers for the employee example, with its high percentage of virtual function calls, are particularly encouraging. We would expect these numbers to improve if our optimization was integrated into a compiler, rather than being implemented as a source-to-source transformation.

References

[1] T. Ball and J. R. Larus. "Branch prediction for free". In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices*, pages 300–13, New York, NY, USA, June 1993. Association for Computing Machinery.

[2] D. W. Binkley and Bradley M. Kuhn. "Execution timing: some experience and advice". Submitted to *;login:* The Usenix Association Newsletter, 1995.

[3] Brad Calder and Dirk Grunwald. "Reducing indirect function call overhead in C++ programs". In *Proceedings of the Twenty First Annual Symposium on Principles of Programming Languages*, Portland, Oregon, USA, January 1994. Association for Computing Machinery.

[4] J. Dean, C. Chambers, and D. Grove. "Selective specilization for object-oriented languages". In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices*, pages 93–102, New York, NY, USA, June 1995. Association for Computing Machinery.

[5] H. M. Deitel and P. J. Deitel. *C++ how to program*, chapter 10, pages 531–540. Prentice Hall, Englewood Cliffs, NJ, USA, 1994.

[6] N Graham. *Learning C++*. Mc Graw Hill, New York, NY, USA, 1991.

[7] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Hall, San Mateo, CA, USA, 1990.

[8] U. Hölzle. *"Adaptive optimization for SELF: reconciling high performance with explotatory progrmming"*. PhD thesis, Stanford University, 1994.

[9] M. Linton, J. Vlissides, and P. Calder. "Composing user interfaces with interviews". *IEEE Computer*, 22(2):8–22, Feburary 1989.

[10] Hemant D. Pande and Barbara G. Ryder. "Static type determination in C++". In *Proc. Sixth C++ Technical Conference*, Cambridge, MA, USA, April 1994. Usenix Association.

Bradley M. Kuhn graduated from the Computer Science department at Loyola College in Maryland in May of 1995. He is currently working as a software consultant in Baltimore, Maryland.

David Binkley is an Assistant Professor of computer science at Loyola College in Baltimore Maryland where he has been a member of the faulty since 1991. In the Spring of 1992 he join the National Institute of Standards and Technology (NIST) as a visiting faculty researcher. His research interests include compiler backends, currently supported by NSF, software maintenance cost reduction tools, and software issues in high integrity software system assurance.