Fast Static Analysis of C++ Virtual Function Calls

David F. Bacon and Peter F. Sweeney

IBM Watson Research Center, P.O. Box 704, Yorktown Heights, NY, 10598
Email: {dfb,pfs}@watson.ibm.com

Abstract

Virtual functions make code easier for programmers to reuse but also make it harder for compilers to analyze. We investigate the ability of three static analysis algorithms to improve C++ programs by resolving virtual function calls, thereby reducing compiled code size and reducing program complexity so as to improve both human and automated program understanding and analysis. In measurements of seven programs of significant size (5000 to 20000 lines of code each) we found that on average the most precise of the three algorithms resolved 71% of the virtual function calls and reduced compiled code size by 25%. This algorithm is very fast: it analyzes 3300 source lines per second on an 80 MHz PowerPC 601. Because of its accuracy and speed, this algorithm is an excellent candidate for inclusion in production C++ compilers.

1 Introduction

A major advantage of object-oriented languages is abstraction. The most important language feature that supports abstraction is the dynamic dispatch of methods based on the run-time type of an object. In dynamically typed languages like Smalltalk and SELF, all dispatches are considered dynamic, and eliminating these dynamic dispatches has been essential to obtaining high performance [9, 14, 24].

C++ is a more conservatively designed language. Programmers must explicitly request dynamic dispatch by declaring a method to be virtual. C++ programs therefore suffer less of an initial performance penalty, at the cost of reduced flexibility and increased programmer effort. However, virtual function calls still present a significant source of opportunities for program optimization.

The most obvious opportunity, and the one on which the most attention has been focused, is execution time overhead. Even with programmers specifying virtual functions explicitly, the execution time overhead of virtual function calls in C++ has been measured to be as high as 40% [16]. In addition, as programmers become familiar with the advantages of truly object-oriented design, use of virtual functions increases. The costs associated with developing software are so high that the performance penalty of virtual functions is often not sufficient to deter their use. Therefore, unless compilers are improved, the overhead due to virtual function calls is likely to increase as programmers make more extensive use of this feature.

Other researchers have shown that virtual function call resolution can result in significant performance improvements in execution time performance for C++ programs [6, 3, 16]; in this paper we concentrate on comparing algorithms for resolving virtual function calls, and investigating the reasons for their success or failure.

Another opportunity associated with virtual functions is code size reduction. For a program without virtual function calls (or function pointers), a complete call graph can be constructed and only the functions that are used need to be linked into the final program. With virtual functions, each virtual call site has multiple potential targets. Without further knowledge, all of those targets and any functions they call transitively must be included in the call graph.

As a result, object-code sizes for C++ programs have become a major problem in some environments, particularly when a small program is statically linked to a large object library. For instance, when a graphical "hello world" program is statically linked to a GUI object library, even though only a very small number of classes are actually instantiated by the program, the entire library can be dragged in.

Finally, virtual function calls present an analogous

Appears in the Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96), October 1996, San Jose, California. SIGPLAN Notices volume 31 number 10.

problem for browsers and other program-understanding tools: if every potential target of a virtual function call is included in the call graph, the user is presented with a vastly larger space of object types and functions that must be comprehended to understand the meaning of the program as a whole.

In this paper, we compare three fast static analysis algorithms for resolving virtual function calls and evaluate their ability to solve the problems caused by virtual function calls in C++. We also use dynamic measurements to place an upper bound on the potential of static analysis methods, and compare the analysis algorithms against more sophisticated analyses like alias analysis. Finally, we present measurements of the speed of the analysis algorithms, which demonstrate that they are fast enough to be included in commercial-quality compilers.

1.1 Outline

Section 2 briefly describes and compares the mechanics of the three static analysis algorithms that are evaluated in this paper. Section 3 describes our benchmarks, presents the results of our measurements, and explains the reason behind the success or failure of the analysis algorithms. Section 4 describes related work, and Section 5 presents our conclusions.

2 Static Analysis

In this paper we will be comparing three static analysis algorithms, called *Unique Name* [6], *Class Hierarchy Analysis* [11, 13], and *Rapid Type Analysis* [4]. We will sometimes abbreviate them as UN, CHA, and RTA, respectively.

In this section we give a brief overview of the three algorithms, and use a small example program to illustrate the differences between them. We then briefly compare them in power to other static analyses, and discuss the interaction of type safety and analysis.

2.1 Unique Name

The first published study of virtual function call resolution for C++ was by Calder and Grunwald [6]. They were attempting to optimize C++ programs at link time, and therefore had to confine themselves to information available in the object files. They observed that in some cases there is only one implementation of a particular virtual function anywhere in the program. This

```
class A {
  public:
    virtual int foo() { return 1; };
};

class B: public A {
  public:
    virtual int foo() { return 2; };
    virtual int foo(int i) { return i+1; };
};

void main() {
    B* p = new B;
    int result1 = p->foo(1);
    int result2 = p->foo();
    A* q = p;
    int result3 = q->foo();
}
```

Figure 1: Program illustrating the difference between the static analysis methods.

can be detected by comparing the mangled names [1] of the C++ functions in the object files.

When a function has a unique name (really a unique signature), the virtual call is replaced with a direct call. While it can be used within a compiler in the same manner as the other algorithms evaluated in this paper, Unique Name has the advantage that it does not require access to source code and can optimize virtual calls in library code. However, when used at link-time, Unique Name operates on object code, which inhibits optimizations such as inlining.

Figure 1 shows a small program which illustrates the power of the various static analyses. There are three virtual calls in `main()`. Unique Name is able to resolve the first call (that produces `result1`) because there is only one virtual function called `foo` that takes an integer parameter – `B::foo(int)`. There are many `foo` functions that take no parameters, so it can not resolve the other calls.

2.2 Class Hierarchy Analysis

Class Hierarchy Analysis [11, 13] uses the combination of the statically declared type of an object with the class hierarchy of the program to determine the set of possible targets of a virtual function call. In Figure 1, `p` is a

[1] The *mangled name* of a function is the name used by the linker. It includes an encoding of the class and argument types to distinguish it from other identically named functions.

pointer whose static type is B*. This means that p can point to objects whose type is B or any of B's derived classes.

By combining this static information with the class hierarchy, we can determine that there are no derived classes of B, so that the only possible target of the second call (that produces result2) is int B::foo().

Class Hierarchy Analysis is more powerful than Unique Name for two reasons: it uses static information (as in Figure 1), and it can ignore identically-named functions in unrelated classes.

Class Hierarchy Analysis must have the complete program available for analysis, because if another module defines a class C derived from B that overrides foo(), then the call can not be resolved.

In the process of performing Class Hierarchy Analysis, we build a call graph for the program. The call graph includes functions reachable from main() as well as those reachable from the constructors of global-scope objects. Note that some other researchers use the term "Class Hierarchy Analysis" to denote only the resolution of virtual calls, not the building of the call graph.

2.3 Rapid Type Analysis

Rapid Type Analysis [4] starts with a call graph generated by performing Class Hierarchy Analysis. It uses information about instantiated classes to further reduce the set of executable virtual functions, thereby reducing the size of the call graph.

For instance, in Figure 1, the virtual call q->foo() (which produces result3) is not resolved by Class Hierarchy Analysis because the static type of q is A*, so the dynamic type of the object could be either A or B. However, an examination of the entire program shows that no objects of type A are created, so A::foo() can be eliminated as a possible target of the call. This leaves only B::foo().

Note that RTA must not consider instantiation of sub-objects as true object instantiations: when an object of type B is created, A's constructor is called to initialize the A sub-object of B. However, the virtual function table of the contained object still points to B's foo() method.

Rapid Type Analysis builds the set of possible instantiated types *optimistically*: it initially assumes that no functions except main are called and that no objects are instantiated, and therefore no virtual call sites call any of their target functions. It traverses the call graph created by Class Hierarchy Analysis starting at main. Virtual call sites are initially ignored. When a constructor for an object is found to be callable, any of the virtual methods of the corresponding class that were left out are then traversed as well. The live portion of the call graph and the set of instantiated classes grow iteratively in an interdependent manner as the algorithm proceeds.

Rapid Type Analysis inherits the limitations and benefits of Class Hierarchy Analysis: it must analyze the complete program. Like CHA, RTA is flow-insensitive and does not keep per-statement information, making it very fast.

Rapid Type Analysis is designed to be most effective when used in conjunction with class libraries. For instance, a drawing library defines numerous objects derived from class shape, each with their own draw() method. A program that uses the library and only ever creates (and draws) squares will never invoke any of the methods of objects like circle and polygon. This will allow calls to draw() to be resolved to calls to square::draw(), and none of the other methods need to be linked into the final program. This leads to both reduced execution time and reduced code size.

Another approach to customizing code that uses class libraries is to use class slicing [23].

2.4 Other Analyses

There are several other levels of static analysis that can be performed. First, a simple local flow-sensitive analysis would be able to resolve this call:

```
A* q = new B;
q = new A;
result = q->foo();
```

because it will know that q points to an object of type A. Rapid Type Analysis would not resolve the call because both A and B objects are created in this program.

An even more powerful static analysis method is alias analysis, which can resolve calls even when there is intervening code which could potentially change an object's type. Alias analysis is discussed more fully in Section 4.2, with related work.

2.5 Type Safety Issues

An important limitation of CHA and RTA is that they rely on the type-safety of the programs. Continuing to use the class hierarchy from Figure 1, consider the following code fragment:

```
void* x = (void*) new B;
B* q = (B*) x;
int case1 = q->foo();
```

Despite the fact that the pointer is cast to void* and then back to B*, the program is still type-safe because we can see by inspection that the down-cast is actually to the correct type. However, if the original type is A, as in

```
void* x = (void*) new A;
B* q = (B*) x;
int case2 = q->foo();
```

then the program is not type-safe, and the compiler would be justified in generating code that raises an exception at the point of the virtual function call to `foo()`. However, because `foo()` is in fact defined for `A`, most existing compilers will simply generate code that calls `A::foo()`; this may or may not be what the programmer intended. If the call had instead been

```
int case3 = q->foo(666);
```

then the program will result in a undefined run-time behavior (most likely a segmentation fault) because `A`'s virtual function table (VFT) does not contain an entry for `foo(int)`.

The computation of `case1` is clearly legal, and the computation of `case3` is clearly illegal. In general it is not possible to distinguish the three cases statically. Unfortunately, in `case2`, Class Hierarchy Analysis would determine that the call was resolvable to `B::foo()`, which is incorrect. Rapid Type Analysis would determine that there are no possible call targets, which is correct according to the C++ language definition but different from what is done by most compilers.

Therefore, Class Hierarchy Analysis and Rapid Type Analysis either need to be disabled whenever a downcast is encountered anywhere in the program, or they can be allowed to proceed despite the downcast, with a warning printed to alert the programmer that optimization could change the results of the program if the downcasts are truly unsafe (as in `case2` or `case3`).

We favor the latter alternative because downcasting is very common in C++ programs. This can be supplemented by pragmas or compiler switches which allow virtual function call resolution to be selectively disabled at a call site or for an entire module. We will discuss this issue further when we present the results for one of our benchmarks, `lcom`, which contained some unsafe code.

3 Experimental Results

In this section we evaluate the ability of the three fast static analysis methods to solve the problems that were outlined in the introduction: execution time performance, code size, and perceived program complexity. Where possible, we will use dynamic measurement information to place an upper limit on what could be achieved by perfect static analysis.

3.1 Methodology

Our measurements were gathered by reading the C++ source code of our benchmarks into a prototype C++ compiler being developed at IBM. After type analysis is complete, we build a call graph and analyze the code. Since the prototype compiler is not yet generating code reliably enough to run large benchmarks, we compile the programs with the existing IBM C++ compiler on the RS/6000, xlC. The benchmarks are traced, and their executions are simulated from the instruction trace to gather relevant execution-time statistics. We then use line number and type information to match up the call sites in the source and object code.

We used both optimized and unoptimized compiled versions of the benchmarks. The unoptimized versions were necessary to match the call sites in the source code and the object code, because optimization includes inlining, which distorts the call graph. However, existing compilers can not resolve virtual function calls, so optimization does not change the number of virtual calls, although it may change their location, especially when inlining is performed. Therefore, turning optimization (and inlining) off does not affect our results for virtual function resolution. Unoptimized code was only used for matching virtual call sites. All measurements are for optimized code unless otherwise noted.

Because our tool analyzes source code, virtual calls in library code were not available for analysis. Only one benchmark, `simulate`, contained virtual calls in the library code. They are not counted when we evaluate the efficacy of static analysis, since had they been available for analysis they might or might not have been resolved.

The information required by static analysis is not large, and could be included in compiled object files and libraries. This would allow virtual function calls in library code to be resolved, although it would not confer the additional benefits of inlining at the virtual call site.

3.2 Benchmarks

Table 1 describes the benchmarks we used in this study. Of the nine programs, we consider seven to be "real" programs (`sched, ixx, lcom, hotwire, simulate, idl` and `taldict`) which can be used to draw meaningful conclusions about how the analysis algorithms will perform. `idl` and `taldict` are both programs made up of production code with demo drivers;

Benchmark	Lines	Description
sched	5,712	RS/6000 Instruction Timing Simulator
ixx	11,157	IDL specification to C++ stub-code translator
lcom	17,278	Compiler for the "L" hardware description language
hotwire	5,335	Scriptable graphical presentation builder
simulate	6,672	Simula-like simulation class library and example
idl	30,288	SunSoft IDL compiler with demo back end
taldict	11,854	Taligent dictionary benchmark
deltablue	1,250	Incremental dataflow constraint solver
richards	606	Simple operating system simulator

Table 1: Benchmark Programs. Size is given in non-blank lines of code

the rest are all programs used to solve real problems. The remaining two benchmarks, `richards` and `deltablue`, are included because they have been used in other papers and serve as a basis for comparison and validation.

Table 2 provides an overview of the static characteristics of the programs in absolute terms. Library code is not included. The number of functions, call sites, and virtual call arcs gives a composite picture of the static complexity of the program. Live call sites are those which were executed in our traces. Non-dead virtual call sites are those call sites, both resolved and unresolved, that remained in the program after our most aggressive analysis (RTA) removed some of the dead functions and the virtual call sites they contained.

Table 3 provides an overview of the dynamic (execution time) program characteristics for optimized code. Once again, all numbers are for user code only. The number of instructions between virtual function calls is an excellent (though crude) indication of how much potential there is for speedup from virtual function resolution. Under IBM's AIX operating system and C++ run-time environment a virtual function call takes 12 instructions, meaning that the user code of `taldict` could be sped up by a factor of two if all virtual calls are resolved (as they in fact are).

The graphs in the paper all use percentages because the absolute numbers vary so much. Tables 2 and 3 include the totals for all subsequent graphs, with the relevant figure indicated in square brackets at the top of the column.

Figure 2 is a bar graph showing the distribution of types of live call sites contained in the user code of the programs; Figure 3 shows the analogous figures for the number of dynamic calls in user code. Direct (non-virtual) method calls account for an average of 51% of the static call sites in the seven large applications, but only 39% of the dynamic calls. Virtual method calls account for only 21% of the static call sites, but a much more significant 36% of the total dynamic calls.

Indirect function calls are used sparely except by `deltablue`, and pointer-to-member calls are only used by `ixx`, and then so infrequently that they do not appear on the bar chart.

Since non-virtual and virtual method calls are about evenly mixed, and direct (non-method) calls are less frequent, we conclude that the programs are written in a relatively object-oriented style. However, only some of the classes are implemented in a highly reusable fashion, because half of the method calls are non-virtual. The exception is `taldict`, with 89% of the dynamic function calls virtual: `taldict` uses the Taligent frameworks, which are designed to be highly re-usable. As use of C++ becomes more widespread and code reuse becomes more common, we expect that programs will become more like `taldict`, although probably not to such an extreme.

Note that we assume that trivially resolvable virtual function calls are implemented as direct calls, and count them accordingly throughout our measurements. That is, the call to `foo()` in

```
A a;
a.foo();
```

is considered a direct call even if `foo()` is a virtual function. This is consistent with the capabilities of current production C++ compilers, but different from some related work.

Our results differ, in some cases significantly, from those reported in two previous studies of C++ virtual function call resolution [6, 3]. This would seem to indicate that there is considerable variation among applications.

Program	Code Size (bytes) [6]	Functions [7]	Call Sites	Live Call Sites [2]	Virtual Call Sites	Non-Dead V-Call Sites [4]	Virtual Call Arcs [8]
sched	99,888	237	530	184	34	33	58
ixx	178,636	1,108	3,601	767	467	399	1,752
lcom	164,032	779	2,794	1,653	458	446	3,661
hotwire	45,416	230	1,204	550	48	6	83
simulate	28,900	242	580	141	36	23	41
idl	243,748	856	3,671	882	1,248	1,198	3,486
taldict	20,516	429	783	47	79	14	116
deltablue	N.A.	103	372	201	3	3	11
richards	9,744	78	174	68	1	1	5

Table 2: Totals for static (compile-time) quantities measured in this paper. All quantities are measured for user code only (libraries linked to the program are not included). Numbers in brackets are the numbers of subsequent figures for which the column gives the total.

Program	Instrs. Executed	Function Calls [3]	Virtual Calls [5]	Instrs. per Virtual Call
sched	106,901,207	2,302,003	967,789	110
ixx	7,919,945	248,391	47,138	168
lcom	107,826,169	4,210,059	1,099,317	98
hotwire	4,842,856	189,160	33,504	145
simulate	1,230,305	57,537	10,848	113
idl	776,792	33,826	14,211	55
taldict	837,496,535	39,401,445	35,060,980	23
deltablue	10,492,752	558,028	205,100	51
richards	86,916,173	2,407,782	657,900	132

Table 3: Totals for dynamic (run-time) quantities measured in this paper. All quantities are for user code only (libraries linked to the program are not included). Numbers in brackets are the numbers of subsequent figures for which the column gives the total.

Considerable additional work remains to be done for benchmarking of C++ programs. While the SPEC benchmark suite has boiled down "representative" C code to a small number of programs, it may well be that such an approach will not work with C++ because it is a more diverse language with more diverse usage patterns.

3.3 Resolution of Virtual Function Calls

When a virtual call site always calls the same function during one or more runs of the program, we say that it is *monomorphic*. If it calls multiple functions, it is *polymorphic*. If the optimizer can prove that a monomorphic call will *always* call the same function, then it can be resolved statically. Polymorphic call sites can not be resolved unless the enclosing code is cloned or type tests are inserted.

The performance of the analyses for resolving virtual function calls is shown in Figures 4 (which presents the static information for the call sites) and 5 (which presents the dynamic information for the calls in our program traces). Together with the remaining graphs they compare the performance of the three static analysis algorithms, and they all use a consistent labeling to aid in interpretation. Black is always used to label the things that could not possibly be handled by static analysis; in the case of virtual function resolution, black represents the call sites or calls that were polymorphic. White represents the region of possible opportunity for finer analysis; for virtual function resolution, this is the call sites or calls that were dynamically monomorphic but were not resolved by any of the static analysis methods we implemented. For graphs of static quantities, the diagonally striped section labels an additional region of opportunity in unexecuted code; for virtual function resolution, this is the call sites that were not resolved and were not executed at run-time. They may be dead, monomorphic, or polymorphic.

Since Class Hierarchy Analysis (CHA) resolves a superset of the virtual calls resolved by Unique Name (UN), and Rapid Type Analysis (RTA) resolves a superset of the virtual calls resolved by CHA, we show their cumulative effect on a single bar in the chart. Therefore, to see the effect of RTA, the most powerful analysis, include all the regions labeled as "resolved" (they are outlined with a thick line).

If the region of opportunity is very small, then the dynamic trace has given us a tight upper bound: we *know* that no static analysis could do much better. On the other hand, if the white region (and for static graphs, the striped region) is large, then the dynamic trace has only given us a loose upper bound: more powerful static analysis might be able to do better, or it might not.

Call sites identified as dead by Rapid Type Analysis were not counted, regardless of whether they were resolved. This was done so that the static and dynamic measurements could be more meaningfully compared, and because it seemed pointless to count as resolved a call site in a function that can never be executed. However, this has relatively little effect on the overall percentages.

Figure 5 shows that for for five out of seven of the large benchmarks, the most powerful static analysis, RTA, resolves all or almost all of the virtual function calls. In other words, in five out of seven cases, RTA does an essentially perfect job. On average, RTA resolves 71% of the dynamic virtual calls in the seven large benchmarks. CHA is also quite effective, resolving an average of 51%, while UN performs relatively poorly, resolving an average of 15% of the dynamic virtual calls.

We were surprised by the poor performance of Unique Name, since Calder and Grunwald found that Unique Name resolved an average of 32% of the virtual calls in their benchmarks. We are not sure why this should be so; possibly our benchmarks, being on average of a later vintage, contain more complex class hierarchies. UN relies on there only being a single function in the entire application with a particular signature.

Our benchmarks are surprisingly monomorphic; only two of the large applications (`ixx` and `lcom`) exhibit a significant degree of polymorphism. We do not expect this to be typical of C++ applications, but perhaps monomorphic code is more common than is generally believed.

A problem arose with one program, `lcom`, which is not type-safe: applying CHA or RTA generates some specious call site resolutions. We examined the program and found that many virtual calls were *potentially unsafe*, because the code used down-casts. However, most of these potentially unsafe calls *are in fact safe*, because the program uses a collection class defined to hold pointers of type `void*`. Usually, inspection of the code shows that the down-casts are simply being used to restore a `void*` pointer to the original type of the object inserted into the collection.

We therefore selectively turned off virtual function call resolution at the call sites that could not be determined to be safe; only 7% of the virtual calls that would have been resolved by static analysis were left unresolved because of this (they are counted as unresolved monomorphic calls). We feel that this is a reasonable course because a programmer trying to optimize their own program might very well choose to follow this course rather than give up on optimization altogether; readers will have to use their own judgment as to whether this would be an acceptable programming practice in their environment.

The only benchmark to use library code containing virtual calls was simulate, which uses the task library supplied with AIX. Slightly less than half of the virtual calls were made from the library code, and about half of those calls were monomorphic (and therefore potentially resolvable). We have not included virtual calls in library code in the graphs because the corresponding code was not available to static analysis.

3.3.1 Why Rapid Type Analysis Wins

Since Class Hierarchy Analysis is a known and accepted method for fast virtual function resolution, it is important to understand why RTA is able to do better.

RTA does better on four of seven programs, although for idl the improvement is minor. For ixx, RTA resolves a small number of additional static call sites (barely visible in Figure 4), which account for almost 20% of the total dynamic virtual function calls. The reason is that those calls are all to frequently executed string operations. There is a base class String with a number of virtual methods, and a derived class UniqueString, which overrides those methods. RTA determines that no UniqueString objects are created in ixx, and so it is able to resolve the virtual call sites to String methods. These call sites are in inner loops, and therefore account for a disproportionate number of the dynamic virtual calls.

RTA also makes a significant difference for taldict, resolving the remaining 19% of unresolved virtual calls. RTA is able to resolve two additional call sites because they are calls where a hash table class is calling the method of an object used to compare key values. The comparison object base class provides a default comparison method, but the derived class used in taldict overrides it. RTA finds that no instances of the base class are created, so it is able to resolve the calls.

The hotwire benchmark is a perfect example of the class library scenario: a situation in which an application is built using only a small portion of the functionality of a class library. The application itself is a simple dynamic overhead transparency generator; it uses a library of window management and graphics routines. However, it only creates windows of the root type, which can display text in arbitrary fonts at arbitrary locations. All of the dynamic dispatch occurs on redisplay of subwindows, of which there are none in this application. Therefore, all of the live virtual call sites are resolved.

3.3.2 Why Fast Static Analysis Fails

One benchmark, sched, stands out for the poor performance of all three static analysis algorithms evaluated in this paper. Only 10% of the dynamic calls are resolved, even though 30% of the static call sites are resolved, and 100% of the dynamic calls are monomorphic. Of course, a function may be monomorphic with one input but not with another. However, sched appears to actually be completely monomorphic.

The unresolved monomorphic virtual call sites are all due to one particular programming idiom: sched defines a class Base and two derived classes Derived1 and Derived2 (not their real names). Base has no data members, and defines a number of virtual functions whose implementation is always assert(false) – in other words, they will raise an exception when executed. In essence, Base is a strange sort of abstract base class.

Derived1 and Derived2 each implement a mutually exclusive subset of the methods defined by Base, and since Base has no data members, this means that these two object types are totally disjoint in functionality. It is not clear why the common base class is being used at all.

RTA determines that no objects of type Base are ever created. However, the calls to the methods of Derived1 and Derived2 are always through pointers of type Base*. Therefore, there are always two possible implementations of each virtual function: the one defined by one of the derived classes, and the one inherited from Base by the other derived class.

Depending on your point of view, this is either an example of the inability of static analysis to handle particular coding styles, or another excellent reason not to write strange code.

The other benchmark for which none of the static analyses do a very good job is lcom: 45% of the virtual calls are monomorphic but unresolved. 40% of the virtual calls are from a single unresolved call site. These calls are all through an object passed in from a single procedure, further up in the call graph. That procedure creates the object with **new**, and it is always of the same type. While it would probably not be resolved by simple flow analysis, it could be resolved by alias analysis.

What kinds of programming idioms are not amenable to fast static analysis? CHA will resolve monomorphic virtual calls for which there is only a single possible target. RTA will also eliminate monomorphic calls when only one of the possible target object types is used in the program. The kind of monomorphic calls that can't be resolved by RTA occur when multiple related object types are used independently, for instance if Square and

`Circle` objects were each kept on their own linked list, instead of being mixed together. We call this *disjointed polymorphism*.

Disjointed polymorphism is what occurs in `lcom` and, in a degenerate fashion, in `sched`. While there are certainly situations in which it does make sense to use disjointed polymorphism, we believe it to be relatively uncommon, and this is borne out by our benchmarks. Disjointed polymorphism presents the major opportunity for alias analysis to improve upon the fast static techniques presented in this paper, since it can sometimes determine that a pointer can only point to one type of object even when multiple possible object types have been created.

3.4 Code Size

Because they build a call graph, Class Hierarchy Analysis and Rapid Type Analysis identify some functions as dead: those that are not reachable in the call graph. RTA is more precise because it removes virtual call arcs to methods of uninstantiated objects from the call graph.

Figure 6 shows the effect of static analysis on user code size. As before, white represents the region of opportunity for finer analysis – those functions that were not live during the trace and were not eliminated by static analysis.

Our measurements include only first-order effects of code size reduction due to the elimination of entire functions. There is a secondary code-size reduction caused by resolving virtual call sites, since calling sequences for direct calls are shorter than for virtual calls. We also did not measure potential code expansion (or contraction) caused by inlining of resolved call sites. Finally, due to technical problems our code size measurements are for unoptimized code, and we were not able to obtain measurements for `deltablue`.

On average, 42% of the code in the seven large benchmarks is not executed during our traces. Class Hierarchy Analysis eliminates an average of 24% of the code from these benchmarks, and Rapid Type Analysis gets about one percent more.

CHA and RTA do very well at reducing code size: in five of the seven large benchmarks, less than 20% of the code is neither executed nor eliminated by static analysis. Only `ixx` and `idl` contain significant portions of code that was neither executed nor eliminated (about 40%).

We were surprised to find that despite the fact that RTA does substantially better than CHA at virtual function resolution, it does not make much difference in reducing code size.

Unique Name does not remove any functions because it only resolves virtual calls; it does not build a call graph.

3.5 Static Complexity

Another important advantage of static analysis is its use in programming environments and compilers. For instance, in presenting a user with a program browser, the task of understanding the program is significantly easier if large numbers of dead functions are not included, and if virtual functions that can not be reached are not included at virtual call sites.

In addition, the cost and precision of other forms of static analysis and optimization are improved when the call graph is smaller and less complex.

Figure 7 shows the effect of static analysis on eliminating functions from the call graph. This is similar to Figure 6, except that each function is weighted equally, instead of being weighted by the size of the compiled code. As we stated above, since Unique Name does not build a call graph, it does not eliminate any functions.

Once again, Class Hierarchy Analysis eliminates a large number of functions, and Rapid Type Analysis eliminates a few more.

Figure 8 shows the effect of static analysis on the number of virtual call arcs in the call graph. At a virtual call site in the call graph for a C++ program, there is an arc from the call site to each of the possible virtual functions that could be called.

Class Hierarchy Analysis removes call arcs because it eliminates functions, and so any call arcs that they contain are also removed. Rapid Type Analysis can both remove dead functions and remove virtual call arcs in live functions. For example, refer back to Figure 1 at the beginning of this paper: even though `main()` is a live function, RTA removes the call arc to `A::foo()` at the call that produces `result3` because it discovers that no objects of type `A` are ever created.

Surprisingly, despite the large number of virtual call sites that are resolved in most programs, relatively few virtual call arcs are removed in three of the seven large benchmarks. In those programs, the virtual function resolution is due mostly to Class Hierarchy Analysis. CHA, by definition, resolves a function call when there is statically only a single possible target function at the call site. Therefore, the call site is resolved, but the call arc is not removed. On the other hand, because RTA actually removes call arcs in live functions, it may eliminate substantial numbers of call arcs, as is seen in the case of `hotwire`.

Benchmark	Size (lines)	Analysis Time CHA	Analysis Time RTA	Compile Time	RTA Overhead
sched	5,712	1.90	1.94	921	< 0.1%
ixx	11,157	5.12	5.22	367	1.4%
lcom	17,278	6.27	6.50	218	3.0%
hotwire	5,335	2.05	2.06	160	1.3%
simulate	6,672	2.67	2.75	49	5.6%
idl	30,288	5.71	6.42	450	1.4%
taldict	11,854	1.66	1.78	45	4.0%
deltablue	1,250	0.42	0.44	18	2.4%
richards	606	0.30	0.32	9	3.6%

Table 4: Compile-Time Cost of Static Analysis (timings are in seconds on an 80 MHz PowerPC 601). Compile time is for optimized code, and includes linking. Rightmost column shows the overhead of adding RTA to the compilation process.

3.6 Speed of Analysis

We have claimed that a major advantage of the algorithms described in this paper is their speed. Table 4 shows the cost of performing the Class Hierarchy Analysis and Rapid Type Analysis algorithms on an 80 MHz PowerPC 601, a modest CPU by today's standards. The total time to compile and link the program is also included for comparison. We do not include timings for Unique Name because we implemented it on top of CHA, which would not be done in a real compiler. Since Unique Name performed poorly compared to CHA and RTA, we did not feel it was worth the extra effort of a "native" implementation.

RTA is not significantly more expensive than CHA. This is because the major cost for both algorithms is that of traversing the program and identifying all the call sites. Once this has been done, the actual analysis proceeds very quickly.

RTA analyzes an average of 3310 non-blank source lines per second, and CHA is only marginally faster. The entire 17,278-line `lcom` benchmark was analyzed in 6.5 seconds, which is only 3% of the time required to compile and link the code. On average, RTA took 2.4% of the total time to compile and link the program.

We expect that these timings could be improved upon significantly; our implementation is a prototype, designed primarily for correctness rather than speed. No optimization or tuning has been performed yet.

Even without improvement, 3300 lines per second is fast enough to include in a production compiler without significantly increasing compile times.

4 Related Work

4.1 Type Prediction for C++

Aigner and Hölzle [3] compared the execution time performance improvements due to elimination of virtual function calls via class hierarchy analysis and profile-based type prediction. Our work differs from theirs in that we compare three different static analysis techniques, and in that we demonstrate the ability of static analysis to reduce code size and reduce program complexity. We also use dynamic information to bound the performance of static analysis.

Type prediction has advantages and disadvantages compared with static analysis. Its advantages are that it resolves more calls, and does not rely on the type-correctness of the program. Its disadvantages are that it requires the introduction of a run-time test; it requires profiling; and it is potentially dependent upon the input used during the profile.

Ultimately, we believe that a combination of static analysis with type prediction is likely to be the best solution.

In Aigner and Hölzle's study, excluding the trivial benchmarks `deltablue` and `richards` and weighting each program equally, Class Hierarchy Analysis resolved an average of 27% of the dynamic virtual function calls (and a median of 9%). They said they were surprised by the poor performance of CHA on their benchmarks, since others had found it to perform well. In our measurements, CHA resolved an average of 51% of the dynamic virtual calls, so it seems that there is considerable variation depending upon the benchmark suite. In fact, we got different results for the one large benchmark that we had in common, ixx, due to a different input file and possibly a different version of the program.

Type prediction can always "resolve" more virtual calls than static analysis, because it precedes a direct call with a run-time test. Call sites resolved by static analysis do not need to perform this test, and one would therefore expect the execution time benefit from static resolution to be greater than that from type prediction. This trend is indeed evident in their execution time numbers: for only one of their benchmarks does type feedback provide more than a 3% speedup over Class Hierarchy Analysis. This is despite the fact that in all but one of the benchmarks, type prediction resolves a significantly larger number of virtual calls.

4.2 Alias Analysis for C++

The most precise, and also most expensive, proposed static method for resolving virtual function calls is to use interprocedural flow-sensitive alias analysis. Pande and Ryder [19, 18] have implemented an alias analysis algorithm for C++ based on Landi et al.'s algorithm for C [15]. This analysis is then used to drive virtual function elimination. They give preliminary results for a set of 19 benchmark programs, ranging in size from 31 to 968 lines of code.

In comparison with our RTA algorithm, which processes about 3300 lines of source code per second (on an 80 MHz PowerPC 601), the speed of their algorithm ranges from 0.4 to 55 lines of source code per second (on a Sparc-10). At this speed, alias analysis will not be practical in any normal compilation path.

We have obtained their benchmark suite; Figure 10 shows the performance of our static analysis algorithms on the 9 programs that we could execute (since their analysis is purely static, not all programs were actually executable). Of these 9, two are completely polymorphic (no resolution is possible), and two were all or almost all resolved by Rapid Type Analysis or Class Hierarchy Analysis. So for four out of nine, RTA does as well as alias analysis.

RTA resolved 33% of the virtual call sites in `objects`, compared to about 50% by alias analysis (for comparative data, see their paper [19]). For the remaining four (`deriv1`, `deriv2`, `family`, and `office`) fast static analysis did not resolve any virtual call sites, and significant fractions of the call sites were dynamically monomorphic. Alias analysis was able to resolve some of the virtual call sites in `deriv1` and `deriv2`, and all of the virtual call sites in `family` and `office`. However, the latter two programs are contrived examples where aliases are deliberately introduced to objects created in the `main` routine.

Because of the small size and unrealistic nature of the benchmarks used by Pande and Ryder, we hesitate to make any generalizations based on the results of our comparison. Two of our seven large benchmarks, `sched` and `lcom`, appear to be programs for which alias analysis could perform better than RTA. These programs make use of *disjointed polymorphism*, as discussed in Section 3.3.2.

Over all, our benchmarks and Pande and Ryder's indicate that for most programs, there is relatively little room for improvement by alias analysis over RTA. However, there are definitely cases where alias analysis will make a significant difference. The ideal solution would be to use RTA first, and only employ alias analysis when RTA fails to resolve a large number of monomorphic calls.

In a similar vein as Pande and Ryder, Carini et al. [7] have also devised an alias analysis algorithm for C++ based on an algorithm for C and Fortran [10, 5]. We are currently collaborating with them on an implementation of their algorithm within our analysis framework. This will allow a direct comparison of both the precision and the efficiency of alias analysis.

4.3 Other Work in C++

Porat et al. [21] implemented the Unique Name optimization in combination with type prediction in the IBM `xlC` compiler for AIX, and evaluated the results for 3 benchmark programs. Their two large benchmarks were identical to two of ours: `taldict` and `lcom`. They achieved a speedup of 1.25 on `taldict` and a speedup of 1.04 on `lcom`, using a combination of Unique Name and type prediction. Our estimates and experiments indicate that a significantly higher speedup is achievable for `taldict` using Rapid Type Analysis.

Calder and Grunwald [6] implemented the first virtual function resolution algorithm for C++. Their Unique Name algorithm (which might more accurately be called "Unique Signature") is very fast, since it only requires a linear scan over the method declarations in the program. Calder and Grunwald implemented Unique Name as a link-time analysis, and found it to be quite effective. With their benchmarks, it resolved anywhere from 2.9% to 70.3% of the virtual calls executed by the program. We found it to be not nearly so effective on our benchmarks, and it was significantly outperformed by Rapid Type Analysis.

Srivastava [22] developed an analysis technique with the sole object of eliminating unused procedures from C++ programs. He builds a graph starting at the root of the call graph. Virtual call sites are ignored; instead, when a constructor is reached, the referenced virtual methods of the corresponding class are added to the graph. His algorithm could also be used to resolve virtual function calls by eliminating uninstantiated classes from consideration and then using Class Hierarchy Analysis. His technique is less general than RTA because the resulting graph is not a true call graph, and can not be used as a basis for further optimization.

4.4 Other Related Work

Related work has been done in the context of other object-oriented languages like Smalltalk, SELF, Cecil, and Modula-3. Of those, Modula-3 is the most similar to C++.

Fernandez [13] implemented virtual function call elimination as part of her study on reducing the cost of opaque types in Modula-3. She essentially implemented Class Hierarchy Analysis, although only for the purpose of resolving virtual calls, and not for eliminating dead code.

Diwan et al. [12] have investigated a number of algorithms for Modula-3, including an interprocedural uni-directional flow-sensitive technique, and a "name-sensitive" technique.

For the benchmarks they studied, their more powerful techniques were of significant benefit for Modula-3, because they eliminated the `NULL` class as a possible target. However, when `NULL` is ignored (as it is in C++), in all but one case the more sophisticated analyses did no better than class hierarchy analysis. This is interesting because we found several cases in which Rapid Type Analysis was significantly better than Class Hierarchy Analysis – this may indicate that class instantiation information is more important than the flow-based information.

Because of the wide variation we have seen even among our C++ benchmarks, it seems unwise to extrapolate from Modula-3 results to C++. However, despite the difference between their and our algorithms, the basic conclusion is the same: that fast static analysis is very effective for statically typed object-oriented languages.

Dean et al. [11] studied virtual method call elimination for the pure object-oriented language Cecil, which includes support for multi-methods. They analyzed the class hierarchy as we do to determine the set of type-correct targets of a virtual method call, and used this information to drive an intraprocedural flow analysis of the methods. Their method is not directly comparable to RTA: it uses more precise information within procedures, but performs no interprocedural analysis at all. Measured speedups for benchmarks of significant size were on the order of 25%, and code size reduction was also on the order of 25%.

There has been considerable work on type inference for dynamically typed languages [20, 8, 1, 17]. In a recent paper [2], Agesen and Hölzle showed that type inference can do as well or better than dynamic receiver prediction in the SELF compiler, and proceeded to extrapolate from these results to C++ by excluding dispatches for control structures and primitive types. However, C++ and SELF may not be sufficiently similar for such comparisons to be meaningful.

5 Conclusions

We have investigated the ability of three types of static analysis to improve C++ programs by resolving virtual function calls, reducing compiled code size, and reducing program complexity to improve both human and automated program understanding and analysis.

We have shown that Rapid Type Analysis is highly effective for all of these purposes, and is also very fast. This combination of effectiveness and speed make Rapid Type Analysis an excellent candidate for inclusion in production C++ compilers.

RTA resolved an average of 71% of the virtual function calls in our benchmarks, and ran at an average speed of 3300 non-blank source lines per second. CHA resolved an average of 51% and UN resolved an average of only 15% of the virtual calls. CHA and RTA were essentially identical for reducing code size; UN is not designed to find dead code. RTA was significantly better than CHA at removing virtual call targets.

Unique Name was shown to be relatively ineffective, and can therefore not be recommended. Both RTA and CHA were quite effective. In some cases there was little difference, in other cases RTA performed substantially better. Because the cost of RTA in both compile-time and implementation complexity is almost identical to that of CHA, RTA is clearly the best of the three algorithms.

We have also shown, using dynamic traces, that the best fast static analysis (RTA) often resolves all or almost all of the virtual function calls (in five out of the seven large benchmarks). For these programs, there is no advantage to be gained by using more expensive static analysis algorithms like flow-sensitive type analysis or alias analysis. Since these algorithms will invariably be at least one to two orders of magnitude more expensive than RTA, RTA should be used first to reduce the complexity of the program and to determine if there are significant numbers of virtual call sites left to resolve. In some cases, this will allow the expensive analysis to be skipped altogether.

Acknowledgements

We thank Michael Burke, Susan Graham, and Jim Larus for their support of our work; Harini Srinivasan and G. Ramalingam for their assistance with the development of the analyzer; Mark Wegman for his many helpful suggestions; Ravi Nair for the use of his *xtrace* system, the accompanying benchmarks, and for his technical assistance; Vance Waddle for his NARC graph display system; and Yong-Fong Lee and Mauricio Serrano for sharing their benchmark suite and their insights. We thank Gerald Aigner, Urs Hölzle, Brad Calder, and Dirk Grunwald for helpful discussions and explanations of their work.

We also thank Rob Cecco, Yee-Min Chee, Derek Inglis, Michael Karasick, Derek Lieber, Mark Mendell, Lee Nackman, Jamie Schmeiser, and the other Montana team members for their invaluable assistance with their prototype C++ compiler upon which our optimizer was built.

Finally, we thank those who provided feedback on earlier drafts of this paper: Michael Burke, Paul Carini, German Goldszmidt, Urs Hölzle, Michael Karasick, Harini Srinivasan and Kenny Zadeck.

References

[1] Agesen, O. Constraint-based type inference and parametric polymorphism. In *Proceedings of the First International Static Analysis Symposium* (Namur, Belgium, Sept. 1994), B. Le Charlier, Ed., Springer-Verlag, pp. 78–100.

[2] Agesen, O., and Hölzle, U. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *Proceedings of the 1995 ACM Conference on Object Oriented Programming Systems, Languages, and Applications* (Austin, Tex., Oct. 1995), ACM Press, New York, N.Y., pp. 91–107.

[3] Aigner, G., and Hölzle, U. Eliminating virtual function calls in C++ programs. In *Proceedings of the Tenth European Conference on Object-Oriented Programming – ECOOP'96* (Linz, Austria, July 1996), vol. 1098 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 142–166.

[4] Bacon, D. F., Wegman, M., and Zadeck, K. Rapid type analysis for C++. Tech. Rep. RC number pending, IBM Thomas J. Watson Research Center, 1996.

[5] Burke, M., Carini, P., Choi, J.-D., and Hind, M. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the Seventh International Workshop on Languages and Compilers for Parallel Computing* (Ithaca, N.Y., Aug. 1994), K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds., vol. 892 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 234–250.

[6] Calder, B., and Grunwald, D. Reducing indirect function call overhead in C++ programs. In *Conference Record of the Twenty-First ACM Symposium on Principles of Programming Languages* (Portland, Ore., Jan. 1994), ACM Press, New York, N.Y., pp. 397–408.

[7] Carini, P., Hind, M., and Srinivasan, H. Type analysis algorithm for C++. Tech. Rep. RC 20267, IBM Thomas J. Watson Research Center, 1995.

[8] Chambers, C., and Ungar, D. Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs. *LISP and Symbolic Computation 4*, 3 (July 1991), 283–310.

[9] Chambers, C., Ungar, D., and Lee, E. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. *LISP and Symbolic Computation 4*, 3 (July 1991), 243–281.

[10] Choi, J.-D., Burke, M., and Carini, P. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages* (Charleston, South Carolina, Jan. 1993), ACM Press, New York, N.Y., pp. 232–245.

[11] Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the Ninth European Conference on Object-Oriented Programming – ECOOP'95* (Aarhus, Denmark, Aug. 1995), W. Olthoff, Ed., vol. 952 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 77–101.

[12] Diwan, A., Moss, J. E. B., and McKinley, K. S. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of*

the 1996 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (San Jose, Calif., Oct. 1996), pp. 292–305.

[13] Fernandez, M. F. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (La Jolla, Calif., June 1995), ACM Press, New York, N.Y., pp. 103–115.

[14] Hölzle, U., Chambers, C., and Ungar, D. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming – ECOOP'91* (Geneva, Switzerland, July 1991), P. America, Ed., vol. 512 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 21–38.

[15] Landi, W., Ryder, B. G., and Zhang, S. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, June 1993), ACM Press, New York, N.Y., pp. 56–67.

[16] Lee, Y., and Serrano, M. J. Dynamic measurements of C++ program characteristics. Tech. Rep. STL TR 03.600, IBM Santa Teresa Laboratory, Jan. 1995.

[17] Oxhøj, N., Palsberg, J., and Schwartzbach, M. I. Making type inference practical. In *Proceedings of the European Conference on Object-Oriented Programming – ECOOP'92* (Utrecht, Netherlands, June 1992), O. L. Madsen, Ed., vol. 615 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 329–349.

[18] Pande, H. D., and Ryder, B. G. Static type determination for C++. In *Proceedings of the 1994 USENIX C++ Conference* (Cambridge, Mass., Apr. 1994), Usenix Association, Berkeley, Calif., pp. 85–97.

[19] Pande, H. D., and Ryder, B. G. Data-flow-based virtual function resolution. In *Proceedings of the Third International Static Analysis Symposium* (1996), vol. 1145 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 238–254.

[20] Plevyak, J., and Chien, A. A. Precise concrete type inference for object-oriented languages. In *Proceedings of the 1994 ACM Conference on Object Oriented Programming Systems, Languages, and Applications* (Portland, OR, Oct. 1994), ACM Press, New York, N.Y., pp. 324–340.

[21] Porat, S., Bernstein, D., Fedorov, Y., Rodrigue, J., and Yahav, E. Compiler optimizations of C++ virtual function calls. In *Proceedings of the Second Conference on Object-Oriented Technologies and Systems* (Toronto, Canada, June 1996), Usenix Association, pp. 3–14.

[22] Srivastava, A. Unreachable procedures in object-oriented programming. *ACM Letters on Programming Languages and Systems 1*, 4 (December 1992), 355–364.

[23] Tip, F., Choi, J.-D., Field, J., and Ramalingam, G. Slicing class hierarchies in C++. In *Proceedings of the 1996 ACM Conference on Object Oriented Programming Systems, Languages, and Applications* (San Jose, Calif., Oct. 1996), pp. 179–197.

[24] Ungar, D., Smith, R. B., Chambers, C., and Holzle, U. Object, message, and performance: how they coexist in Self. *Computer 25*, 10 (Oct. 1992), 53–64.

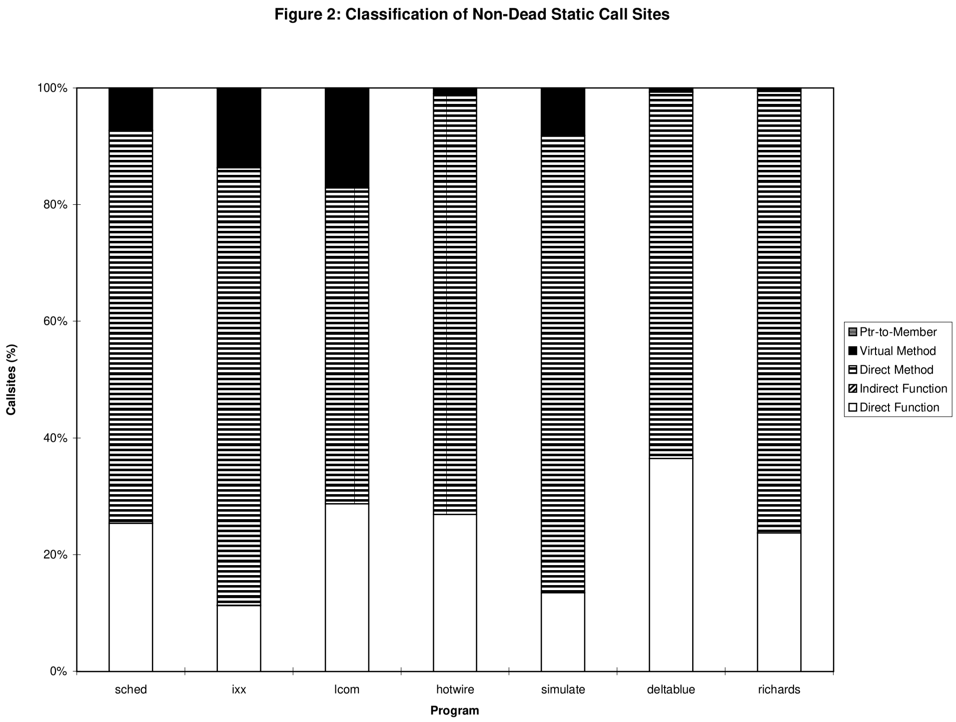

Figure 2: Static Distribution of Function Call Types

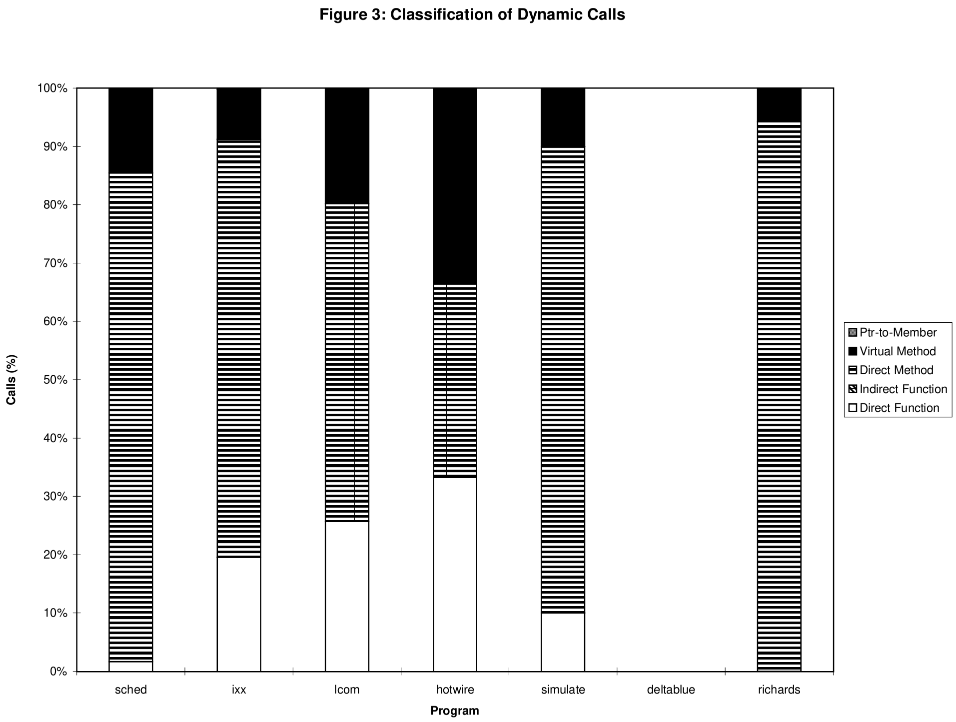

Figure 3: Dynamic Distribution of Function Call Types

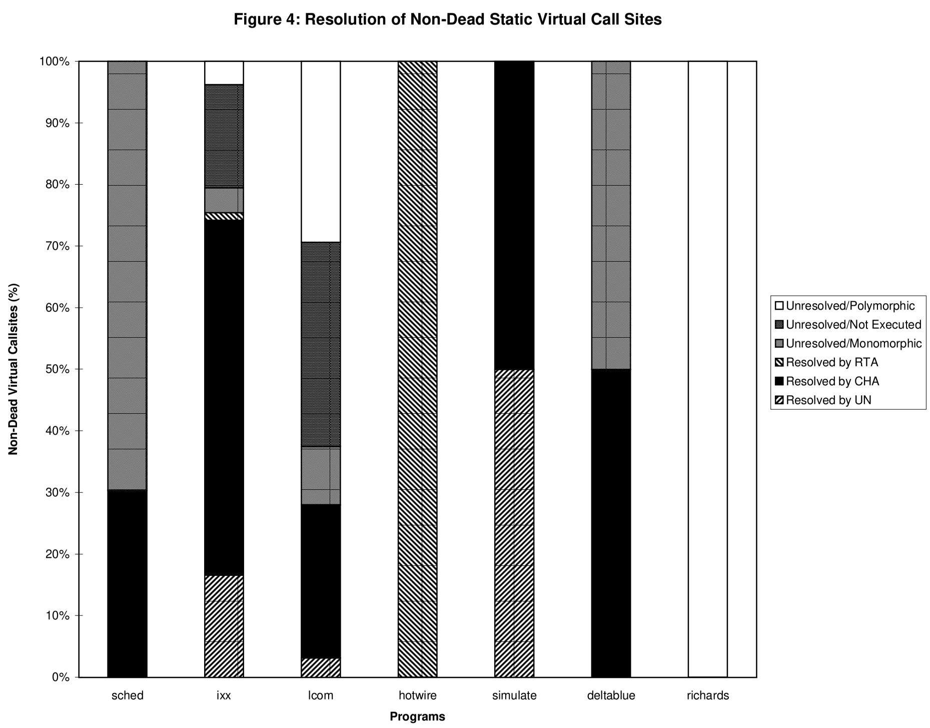

Figure 4: Resolution of Possibly Live Static Callsites

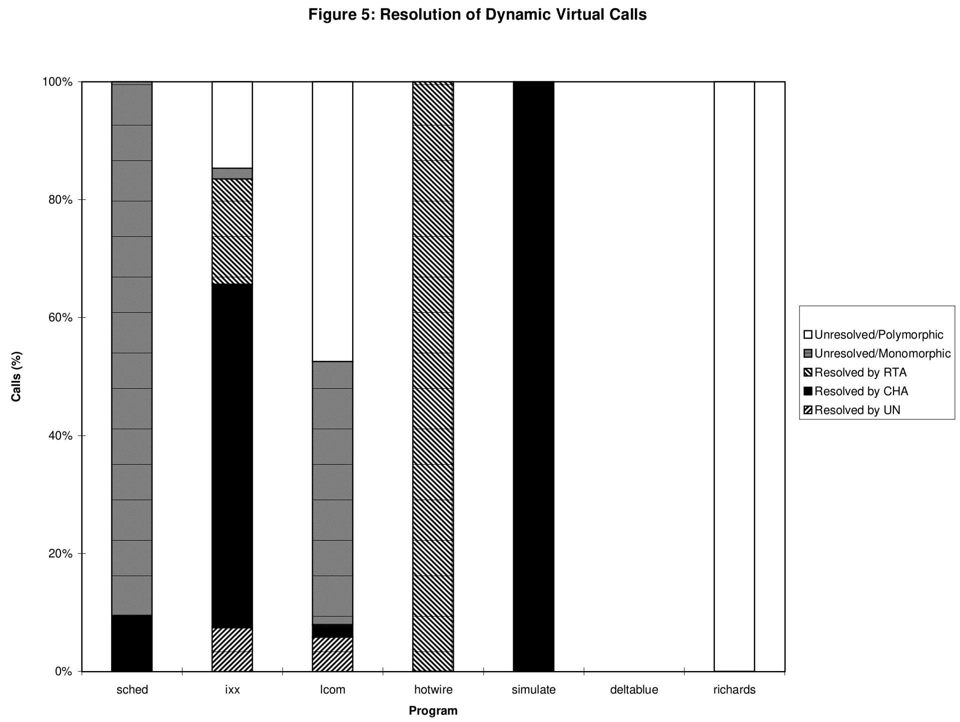

Figure 5: Resolution of Dynamic Calls

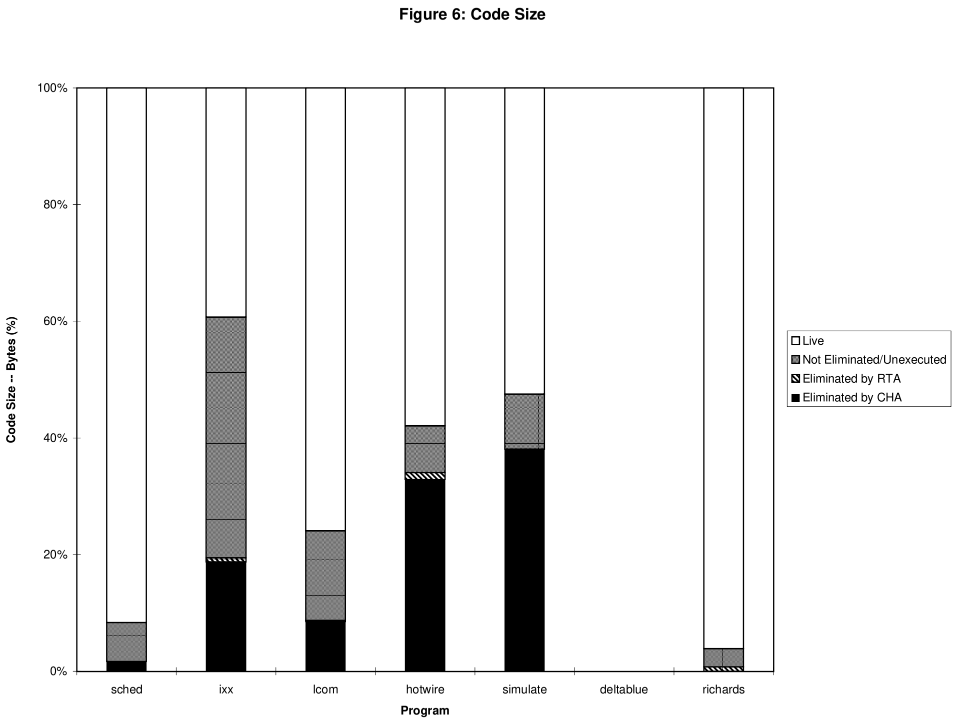

Figure 6: Code Size

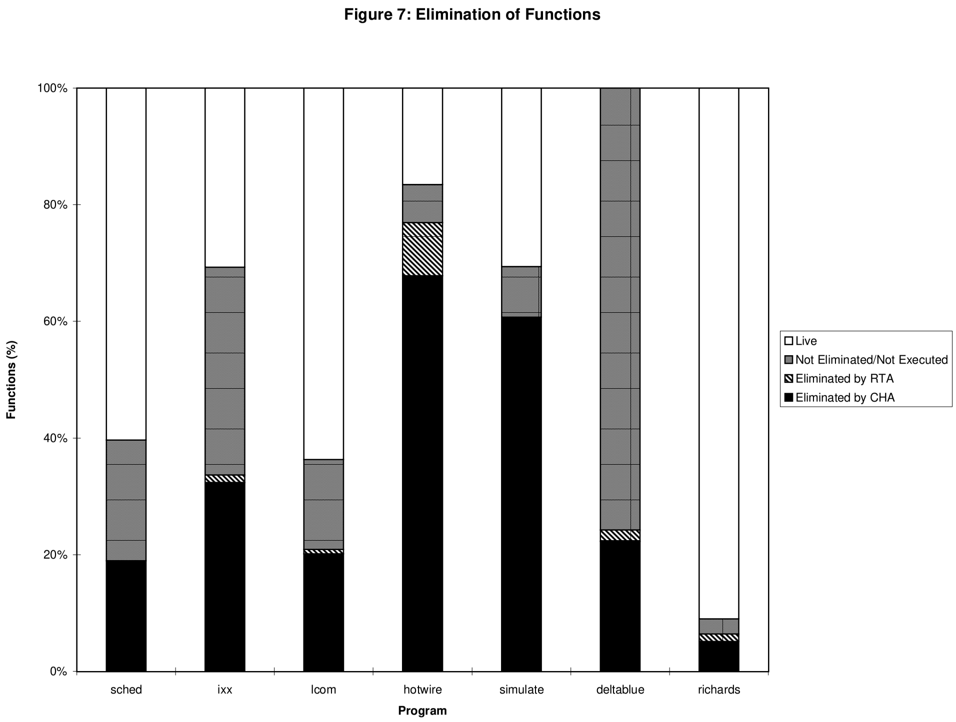

Figure 7: Elimination of Dead Functions by Static Analysis

Figure 8: Elimination of Static Virtual Call Arcs

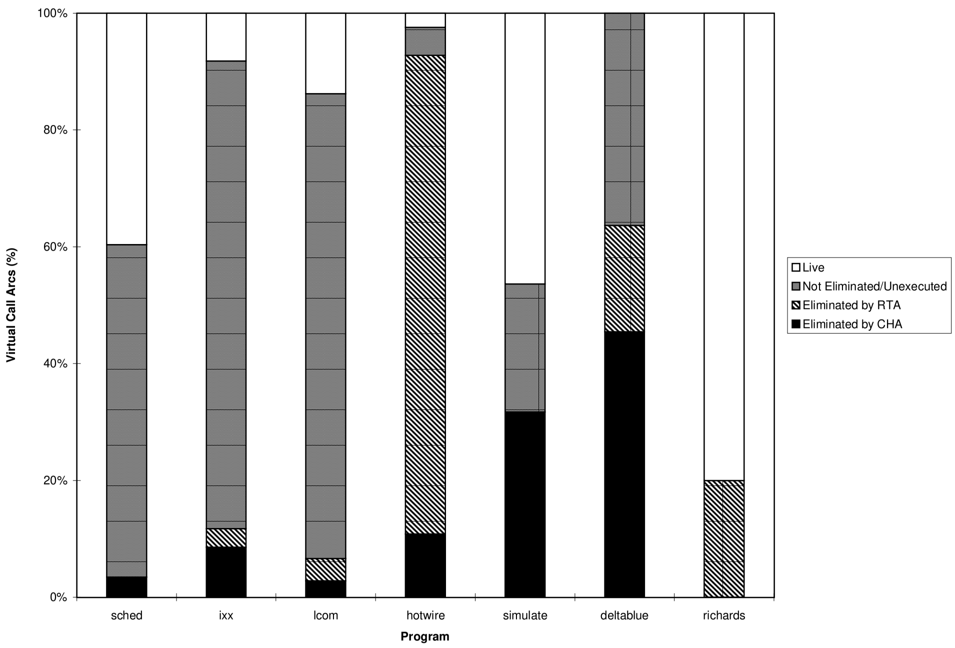

Figure 8: Elimination of Virtual Call Arcs by Static Analysis

Figure 9: Effectiveness of Type Prediction

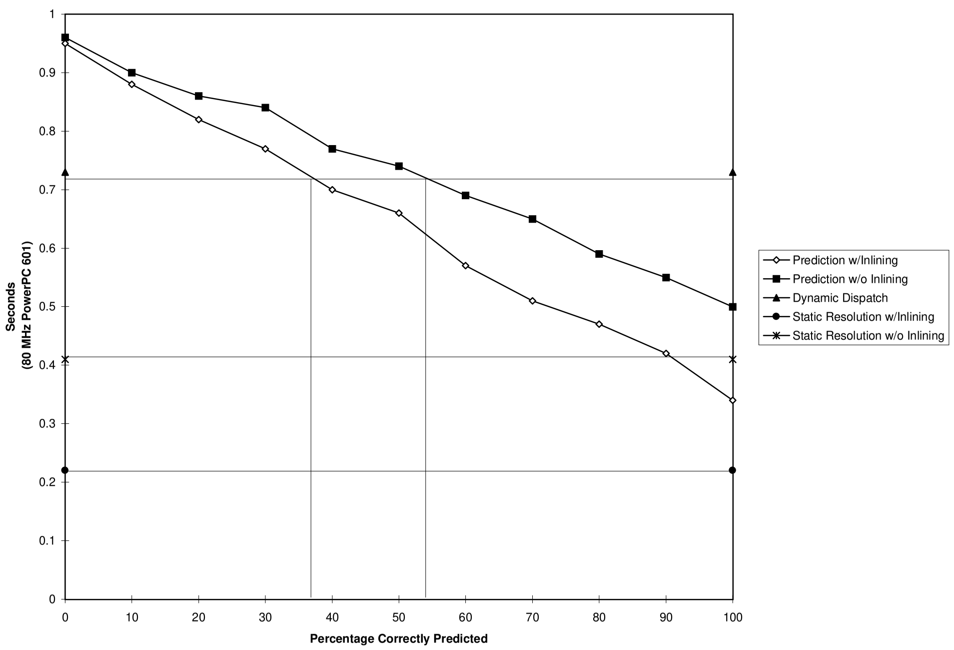

Figure 9: Type Prediction vs. Static Resolution on the PowerPC 601

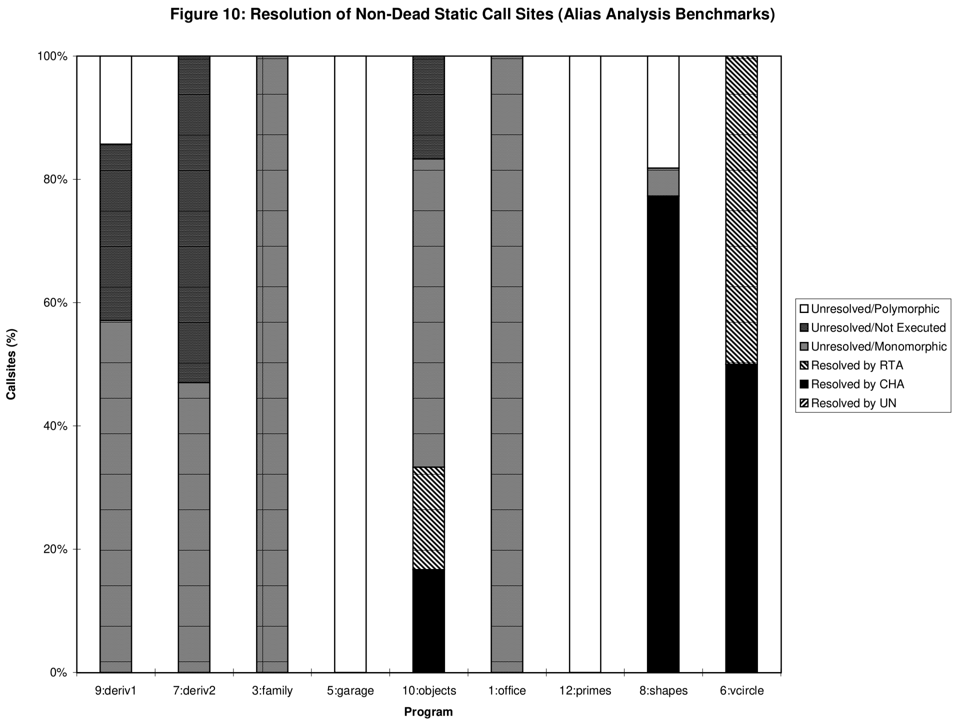

Figure 10: Resolution of Static Callsites – Alias Analysis Benchmarks