Reducing Indirect Function Call Overhead In C++ Programs

Brad Calder and Dirk Grunwald[*]
(Email:`{calder,grunwald}@cs.colorado.edu`)
Department of Computer Science,
Campus Box 430, University of Colorado,
Boulder, CO 80309-0430

Abstract

Modern computer architectures increasingly depend on mechanisms that estimate future control flow decisions to increase performance. Mechanisms such as *speculative execution* and *prefetching* are becoming standard architectural mechanisms that rely on *control flow prediction* to prefetch and speculatively execute future instructions. At the same time, computer programmers are increasingly turning to object-oriented languages to increase their productivity. These languages commonly use *run time dispatching* to implement object polymorphism. Dispatching is usually implemented using an indirect function call, which presents challenges to existing control flow prediction techniques.

We have measured the occurrence of indirect function calls in a collection of C++ programs. We show that, although it is more important to predict branches accurately, indirect call prediction is also an important factor in some programs and will grow in importance with the growth of object-oriented programming. We examine the improvement offered by compile-time optimizations and static and dynamic prediction techniques, and demonstrate how compilers can use existing branch prediction mechanisms to improve performance in C++ programs. Using these methods with the programs we examined, the number of instructions between mispredicted breaks in control can be doubled on existing computers.

Keywords: Object oriented programming, optimization, profile-based optimization, customization

1 Introduction

The design of computer architectures and languages are tightly entwined. For example, the advent of register displacement addressing enabled efficient implementation of Algol and the increased use of COBOL emphasized the use of BCD arithmetic. Likewise, the C and FORTRAN languages have become ubiquitous, strongly influenced RISC processor design. Object-oriented programming has recently gained popularity, illustrated by the wide-spread popularity of C++. Object-oriented languages exercise different aspects of computer architectures to support the object-oriented programming style. In this paper, we examine how indirect function calls, used to support object polymorphism, influence the performance of an efficient object oriented language. Modern architectures using deep instruction pipelines and speculative execution rely on predictable control flow changes, and indirect function calls cause unpredictable changes in program control flow. For example, the DEC Alpha AXP 21064 processor, one of the first widely-available deeply pipelined superscalar microprocessors, stalls for 10 instructions if the processor mispredicts the flow of control. This increases if the mispredicted target is not in the instruction cache and must be fetched. As systems increasingly rely on speculative execution [19, 16], the importance of control flow prediction will increase.

In most programs, conditional branches introduce the main uncertainty in program flow, and architectures use a variety of *branch prediction* techniques to reduce instruction cache misses and to insure instructions are available for the processor pipeline. Most function calls specify explicit call targets, and thus most function calls can be trivially predicted. Control flow prediction is just as important in object-oriented programs, but these languages tend to use indirect function calls, where the address of the call target is loaded from memory. Fisher *et al* [12] said indirection function calls "... are unavoidable breaks in control and there are few compiler or hardware tricks that could allow instruction-level parallelism to advance past them". By accurately predicting the calling address, the processor can reduce instruction stalls and prefetch instructions.

Our results show that accurately predicting the behavior of indirect function calls can largely eliminate the control-flow misprediction penalty for using static-typed object-oriented languages such as C++. Figure 1 shows the *normalized execution time* for a variety of C++ programs. We measured the number of instructions executed by each program, and collected information concerning the conditional branches and indirect function calls executed by each program. Although we measured the programs on a DECstation-5000, we simulated the branch misprediction characteristics of a deeply pipelined superscalar architecture, similar to the DEC Alpha AXP 21064.

For each program, each bar indicates the number of *machine cycles* spent executing instructions and suffering from the delay im-

[*]This paper appeared in the ACM Principles and Practice of Programming Languages, Portland, Oregon 1994.

posed by mispredicting control flow under different assumptions. A value of '1' indicates the program spends no additional time due to delays, while a value of '2' indicates the program would execute twice as slowly. The left-most bar indicates the delay that would be incurred if every control flow change was incorrectly predicted or there was no prediction. The next bar indicates the increase in execution if only *conditional branches* are predicted, using static profile based prediction. The next three bars indicate the decrease in execution time if branches and *indirect function calls* are correctly predicted, using three different techniques described in this paper. For the programs we measured, we found we could improve performance 2%-24% using these simple techniques; the final improvement depends on the number of indirect function calls, how program libraries are used and the underlying architecture. Architectures that have deeper pipelines or that issue more instructions per cycle would evince greater improvement.

We are interested in reducing the cost of indirect function calls (*I-calls*) for modern architectures. If a compiler can determine a unique or likely call target, indirect function calls can be converted to direct calls for unique call targets. Likewise, compilers may choose to inline likely or unique call targets. Inlining I-calls not only reduces the number of function calls, it exposes opportunities for other optimizations such as better register allocation, constant folding, code scheduling and the like. However, type inferencing for C++ programs is an NP-hard problem [22], and will be difficult to integrate into existing compilers, because accurate type inference requires information about the entire type hierarchy. This information is typically not available until programs are linked, implying that type-determination algorithms will require link-time optimization.

We are interested in more subtle optimizations that have a modest, albeit respectable, performance improvement. Many of the optimizations and code transforms rely on modifying an existing program executable and the possibility of assisting the compiler with profile-based optimizations. When we began this study, we asked the following questions:

- Could we predict how frequently compiler-based methods could eliminate indirect function calls?
- How accurate is profile-based prediction? Are dynamic prediction methods more accurate?
- Can we use existing branch prediction hardware to predict indirect function calls?
- How effective are combinations of prediction techniques?
- What type of other compiler optimizations can be applied towards indirect function calls, in order to improve their performance?

To date, our experimentation has demonstrated that although object-oriented libraries support object polymorphism, the target of most indirect function calls can be accurately predicted, either from prior runs of the program or during a program's execution. Furthermore, many existing C++ programs can be optimized naïvely by a linker, using information about the C++ type system. The optimization we examine involves converting an indirect function call that only calls a unique function name into a direct function call. For more sophisticated compilers with link-time code generation, significant optimization opportunities exist. Lastly, a compiler optimization that we term *I-call if conversion* can be used to increase the performance of indirect function calls. This information can be used by a simple profile-based binary modification to improve execution on existing C++ programs by 2-24% on modern architectures.

We measured the behavior of a variety of publicly available C++ programs, collecting information on instruction counts and function calls. This information is also part of a larger study to quantify the differences between C++ and conventional C programs [7]. In this study, we show that call prediction is important for many C++ programs and show to what extent static, dynamic and compile-directed methods can reduce indirect function call overhead. We also demonstrate the opportunity for profile-based optimization in the C++ programs we measured. These optimizations can profitably be applied to existing architectures that incur significant control-flow misprediction penalties.

The results we present are divided into two portions; the first considers applying hardware branch prediction mechanisms to existing C++ programs, while the second considers additional profile based optimizations that can be applied to C++ programs. In §2, we discuss some relevant prior work. In §3, we describe the experimental methodology we used and describe the programs we instrumented and measured. In §4 we compare the various *I-call prediction* mechanisms we studied and summarize how we can improve the performance of existing C++ programs.

2 Prior Work

A considerable amount of research has been conducted in reducing the overhead of method calls in dynamically typed object-oriented languages. Many of the solutions relevant in that domain apply to the optimization of compiled languages using I-calls, as in C++. We will discuss these shortly. Furthermore, numerous researchers have examined the similar issue of reducing *branch overhead*. Since both function calls and branches alter the control flow, there is considerable overlap in our work, yet substantial differences as well. One such difference is that a conditional branch has only two possible targets, while an indirect function call can have a large number of potential targets. We have measured programs with 191 different subroutines called from a single indirect function call. This makes branches easier to predict. Some dynamic branch prediction mechanisms achieve $\approx 95\% - 97\%$ prediction accuracy [21, 27, 29]. This level of accuracy is needed for super-scalar processors issuing several instructions per cycle [28].

The most relevant prior work was on predicting the destination of indirect function calls with hardware conducted by David Wall [26] while examining limits to instruction level parallelism. He states "*Little work has been done on predicting the destinations of indirect jumps, but it might pay off in instruction-level parallelism.*" He simulates static profile based prediction and infinite and finite dynamic last call prediction, and finds they accurately predict I-call destinations; however, the benefit of I-call prediction was minimal because he only examined C and Fortran programs. We go beyond his research by (1) showing that I-call prediction is important for C++ programs, (2) that compile-time optimizations can be combined with static profile based prediction to increase a programs

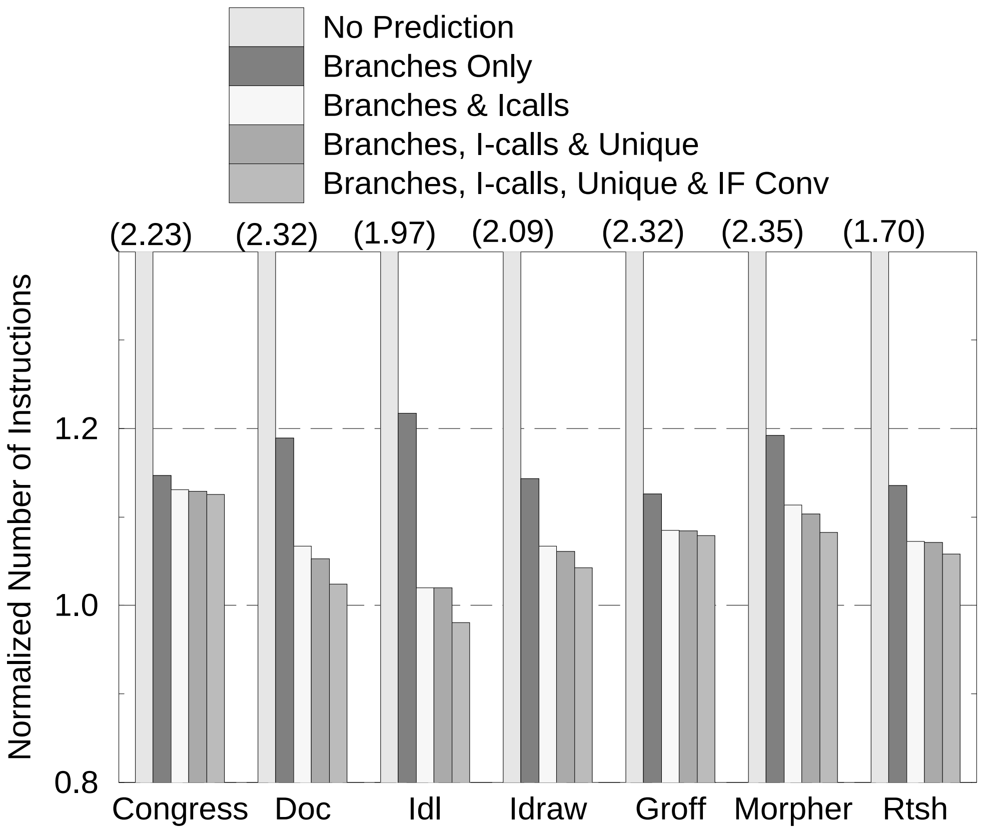

Figure 1: Normalized Execution Time of Various C++ Programs, Including Expected Instructions and Stalls Due to Mispredicted Control Flow, For A Computer Similar to DEC Alpha AXP 21064.

performance, (3) that simple techniques yield very accurate I-call prediction rates and, (4) we do a more in-depth comparison between different static and dynamic mechanisms for doing I-call prediction.

2.1 Compiler Optimizations

Two classes of compiler-oriented optimizations are most relevant to our research. A large body of research exists in dataflow analysis to determine the set of possible call targets for a given call site. Ryder[24] presented a method for computing the call-graph, or set of functions, that may be called by each call site using dataflow equations. More recent work by Burke[4] and Hall[13] has refined this technique. All of this work characterized programs where procedure values may be passed as function parameters. Others, including Hall [13], also examine functional programs. More recently, Pande and Ryder[22] have shown that inferring the set of call targets for call sites in languages with the C++ type system is a NP-hard problem.

In this paper, we seek to minimize pipeline stalls using information concerning the *frequency* of calling specific call sites. Unless the previous algorithms can determine a *unique* call target, more information is needed for I-call prediction. However, the results of this study indicates that single call targets occur frequently, and the techniques in e.g., Ryder [22] may be very successful in practice. To our knowledge, there has been little work in specifying the probability of specific call targets being called using dataflow techniques.

By comparison, there has been considerable work in adaptive runtime systems to reduce the cost of polymorphism in dynamically typed languages. Most recent, and foremost, in these efforts is the work by the SELF project [8, 9, 14] on *customization* of method dispatches and extensive optimization of method lookup. SELF is a dynamicly-typed language, providing a rich set of capabilities not present in statically-typed languages such as C++. However, statically-typed languages such as C++ are very efficient, using a constant-time method dispatch mechanism. Statically-type object-oriented languages are popular because less compiler effort is needed to achieve reasonable performance and the object-oriented programming style encourages software resources and structured software libraries.

Customization and other optimizations have produced considerable performance improvement in the SELF implementation. In many ways, we are extending some of the optimizations explored by the SELF project to the C++ language. However, we must rely on hardware for "customization" (e.g., prediction hardware) rather than software, because most C++ implementations are already very efficient. For example, an indirect function call in C++ takes seven instructions on a DECstation – reducing this cost to greatly improve the performance of an application is difficult. We also feel that the results of our research will benefit prototyping languages such as SELF and SmallTalk as those languages are further optimized for modern architectures. There are also secondary effects to these optimizations. In certain cases, our code transformations will allow function inlining, facilitating further optimization.

SELF uses an "inline cache" to speed indirect function calls. The inline cache records the last target address for each call site; when a method is called, the jump target is recovered from the software cache. In [14], Hölzle *et al* found that they could improve performance by converting an inline cache access into a *polymorphic inline cache* (PIC) lookup. The PIC encodes a data-dependent type check in a "stub" procedure, dynamically modifying the program; this reduces searching for the appropriate method for a given data type. We use a similar technique we termed "if-conversion," described later in this paper. We convert an indirect call into a type check and a direct method call when trying to optimize an indirect call site. The main differences between our two methods is that we are compiling for a staticly-type object oriented language (C++) and Hölzle *et al*. are compiling for the dynamicly typed language SELF. We use static profiling to gather our data and use an instruction cost model to decide whether a call site could benifit from 'if conversion'. Our method also benifits from indirect jump hardware predicition. The SELF system builds the PIC stub tables at run-time, adding all method types for a call site to the 'stub' function. Indirect jump hardware prediction is of little benifit for SELF because the dynamic method lookups are not implemented as indirect jump instructions.

2.2 Branch Prediction

There are a number of mechanisms to ameliorate the effect of uncertain control flow changes, including static and dynamic branch prediction, branch target buffers, delayed branches, prefetching both targets, early branch resolution, branch bypassing and prepare-to-branch mechanisms [18]. Conventional branch prediction studies typically assume there are two possible *branch targets* for a given *branch point*, because multi-target branches occur infrequently in most programs. Rather than present a comprehensive overview of the field, we focus on methods related to the techniques we consider in this paper.

Some architectures employ *static prediction* hints using either profile-derived information, information derived from compile time analysis or information about the branch direction or branch opcode [25, 20, 2]. Wall[26] found profile-driven static prediction to be reasonably accurate for indirect function calls. Fisher and Freudenberger [12] found profile-derived static prediction to be effective for conditional branches, but they hypothesize that inter-run variations occurred because prior input did not cover sufficient execution paths, and our results provide support for their hypothesis. In general, profile based prediction techniques outperform compile-time prediction techniques or techniques that use hueristics based on branch prediction or instruction opcodes.

Some architectures use *dynamic prediction*, either using tables or explicit branch registers. A *branch target buffer* (BTB) [17, 23] is a small cache holding the address of branch sites and the address of branch targets. There are myriad variations on the general idea. Typically, the cache contains from 32 to 512 entries and may be 2 or 4-way associative. The address of a branch site is used as a tag in the BTB, and matching data is used to predict the branch. Some designs include decoded instructions as well as the branch target. Other designs eliminate the branch target address, observing that most branches only go one of two ways (taken/not taken).

Still other designs eliminate the tag, or branch site address from the table. These designs use the branch site address as an index into a table of *prediction bits*. This information can actually be the prediction information for another branch. However, there's at least a 50% chance the prediction information is correct, and this can be improved somewhat [5]. The most common variants of table-based designs are 1-bit techniques that indicate the direction of the most recent branch mapping to a given prediction bit, and 2-bit techniques that yield much better performance for programs with loops [25, 17, 20]. The advantage of these bit-table techniques is that they keep track of very little information per branch site and are very effective in practice.

Lastly, some computers use explicit *branch target address registers* (BTARs) to specify the branch target [11, 1]. This has numerous advantages, because branch instructions are very compact and easily decoded. BTARs can be applied to both conditional branches and function calls, and the instructions loading branch targets can be moved out of loops or optimized in other ways. Furthermore, addresses specified by explicit branch target registers provide additional hints to instruction caches, and those instructions can be prefetched and decoded early. However, most proposed implementations provide only 4-8 BTARs[1]. The contents of these registers will probably not be saved across function calls. Thus, instructions manipulating the BTARs must occur early in the instruction stream to effectively use BTARs.

At first glance, some of these techniques, such as bit-table techniques, do not appear applicable to I-call prediction, because indirect function calls can jump to a number of call targets. Later, we show how profile-based *I-call if conversion* can use these mechanisms.

3 Experimental Design

Our comparison used trace-based simulation. We instrumented a number of C++ programs, listed in Table 1, using a modified version of the QPT[3] program tracing tool. We emphasized programs using existing C++ class libraries or that structured the application in a modular, extensible fashion normally associated with object-oriented design. A more extensive comparison between the characteristics of C and C++ programs can be found in [7]. Empirical computer science is a labour-intensive undertaking. The programs were compiled and processed on DECstation 5000's. Three C++ compilers (Gnu G++, DEC C++, AT&T C++ V3.0.2) were required to successfully compile all programs; much of this occurred because the C++ language is not standardized. This collection of programs, when instrumented, consumed $\approx$ 1Gb of disk space. Despite the good performance of the QPT tracing tool for conventional programs, it offers little trace compression in programs using indirect calls.

We constructed a simulator to analyze the program traces. Typically the simulator was run once to collect information on call and branch targets, and a second time to use prediction information from the prior run. For one program (GROFF), we compared predictions using input from differing runs to better assess the robustness of our results.

We modified QPT to indicate what caused a basic-block transition (a direct branch, indirect branch or fall-through) and record whether a function call was caused by a direct or indirect call. For most programs, we were also able to indicate what functions were C++ methods. [1] We classified unpredictable breaks in control into three classes: a **2Brs** is a conditional branch, a **>2Brs** is a branch with multiple destinations, usually arising from `switch` or `case` statements, and an **I-call** is an indirect function call. Table 2 lists the number of occurrences for each type of unpredictable break in control for the different programs. We show three entries for GROFF because we use these three executions for coverage analysis later. Entries in the column "*Sites From Trace*" lists the number of branch or call sites of that type encountered during the program execution. For example, DOC actually has $2,367$ indirect function calls, but we only encountered $1,544$ of those calls during the program execution. The heading "*Occurrences During Execution*" lists the number of times breaks of that type appear during program execution. Thus, there were $5,310,059$ indirect function calls when we traced DOC. Approximately 99% of the indirect calls were to C++ methods, except in GROFF, where 95% were to methods.

4 Performance comparison

There are many metrics that can be used to compare I-call prediction and combined I-call and branch prediction techniques. Table 3 shows the *number of instructions between breaks* (NIBBs) without branch or I-call prediction. We tracked only breaks in control flow that will cause a long pipeline delay. These breaks are conditional branches and indirect calls. Returns also cause a long pipeline stall, but returns can be accurately predicted by using a return stack [15], and we did not track these. We assume that unconditional branches, procedure calls and assigned gotos are accurately predicted. These control-transfer instructions, conditional branches and I-calls can also cause an *instruction misfetch penalty* because they must be decoded before the instruction stream knows the instruction type. Thus, the instruction fetch unit may incorrectly fetch the next instruction, rather than the target destination. In another paper [6], we show how these misfetch penalties can be avoided using extra *instruction type bits* and/or a simple *instruction type prediction* table, coupled with the techniques discussed in this paper.

If we are only considering conditional branches and indirect calls. We would expect the parameters in Table 3 to be similar for C and C++ programs. To the contrary, we found that our sample C++ programs had a higher number of instructions between breaks, indicating that C++ programs tend to either be more predictable than C programs or, they use a different linguistic construct for conditional logic. For example, consider a balanced-tree implementation in C and C++. A C programmer might implement a single procedure to balance a tree, passing in several flags to control the actual balancing. A C++ programmer, on the other hand, would tend to use inheritance and the object model to provide similar functionality. Thus, it is not surprising that C++ programs tend to have more procedure calls and fewer conditional operations [7].

In the remaining tables, we only show the mean NIBB for each program and use the harmonic mean of the NIBB as a summary. We also use the *percent of breaks predicted (%BP)* to understand how well various techniques predict breaks. This metric, or the more common misprediction rate, is commonly used to compare branch prediction mechanisms. However, note that the %BP metric does

[1]We could not extract this information for IDL, which was compiled by DEC C++

Name	Description
CONGRESS	Interpreter for a PROLOG-like language. Input was one of the examples distributed with CONGRESS for configuration management.
DOC	Interactive text formatter, based on the InterViews 3.1 library. Input was briefly editing a 10 page document.
GROFF	Groff Version 1.7 — A version of the "ditroff" text formatter. One input was a collection of manual pages, another input was a 10-page paper.
IDL	Sample backend for the Interface Definition Language system distributed by the Object Management Group. Input was a sample IDL specification for the Fresco graphics library.
IDRAW	Interactive structured graphics editor, based on the InterViews 2.6 library. Example was drawing and editing a figure.
MORPHER	Structured graphics "morphing" demonstration, based on the InterViews 2.6 library. Example was a morphed "running man" example distributed with the program.
RTSH	Ray Tracing Shell – An interactive ray tracing environment, with a TCL/TK user interface and a C++ graphics library. Example was a small ray traced image distributed with the program.

Table 1: C++ Programs Instrumented

not account for the density of breaks in a program. For example, there may be a single conditional branch in a 100,000,000 instruction program. While the the branch may always be mispredicted, the number of instructions between breaks remains high. Therefore, it is useful to look at both the NIBB and %BP when comparing prediction techniques across different programs.

4.1 Bounds on Compile-time I-Call Prediction

We were interested in determining how well interprocedural dataflow analysis could predict indirect method calls [22] and we compared these results to profile-based static prediction methods. Method names in C++ are encoded with a unique type signature. If a linker knew the intended type signature at a call site,[2] and there was a single function with that signature, then no other function could be appropriate for that call site and the indirect call could be replaced by a direct call. We call this the *Unique Name* measure, and feel it represents a lower bound on what could be accomplished by a dataflow optimization algorithm – in practice, a dataflow method should be more accurate, because a symbol table in UNIX system typically includes methods from classes that are never invoked. At the other extreme, we recorded the number of *Single Target* I-call sites, or I-call sites that record a single call target during a trace. Compiler directed I-call prediction is most useful if a *single* target can be selected. In the number of traces we recorded, the *Single Target* measure represents an upper bound on the target prediction we could expect from dataflow-based prediction algorithms. Both *Unique Name* and *Single Target* values measure the number of dynamic occurrences. In general, our results indicate significant promise from static analysis of C++ programs. In particular, because it is so effective and simple to implement, we feel the *Unique Name* measure should be integrated into existing compilers and linkers.

[2]Currently, compilers don't provide this information at call sites, but this information is easy to capture.

4.2 Static vs. Dynamic Prediction

Although compiler techniques appear promising, we found that profile-based and dynamic prediction techniques were clearly better. We implemented a simple majority profile-based technique. We ran the programs, recorded the most likely target for each call site and used that to predict call targets in future runs. The results are shown in the column labeled "*Static*" in Table 4. This simple technique accurately predicted a surprisingly large number of I-Calls. In each of these runs, we used the same program input to generate the prediction trace and the measurements shown. To determine how accurate these prediction rates are for different inputs, we ran GROFF with two other inputs. Table 5 shows the percentage of I-Calls predicted using all combinations of the different input files. We found a small number of "prediction sets" appear sufficient to provide accurate predictions. In some cases, profile-based methods have poor performance. In our experience, this usually occurs because the inputs, that were used to establish the profile used to predict the branches and I-calls, did not provide adequate coverage of all the branches and indirect function calls. This problem has been mentioned by others [26, 12], but has not be studied in detail.

Table 4 also shows the effectiveness of idealized *dynamic* prediction techniques. We simulated two infinitely large Branch Target Buffers. The first BTB ("*1-bit*") simply used the previous I-Call target as the prediction for future I-calls, much like the method caching used in Self, where the most recent method is saved. The second ("*2-bit*") used a *2-bit* strategy that avoids changing the prediction information until the previous prediction is incorrect twice in a row. I-Calls were considered unpredicted when first encountered. Surprisingly, the *1-bit* mechanism has worse performance than static prediction; however, it does not require profiling runs. The improvement shown by the *2-bit* technique illustrates that the *1-bit* technique changes prediction too rapidly. For example, if a call site calls the sequence of methods A::X(), B::X(), A::X(), the *1-bit* method would miss three times, while the *2-bit* method would miss only once. This information is important for designers of wide-issue processors. For example, some recent design proposals consider using up to 16KB of memory for such BTB's. In another paper, we show how to eliminate the need for most of those

Program	Sites From Trace			Occurences During Execution			
	2Brs	ICalls	$>$ 2Brs	2Brs	ICalls	$>$ 2Brs	Instructions
congress	1,817	309	4	18,352,179	342,266	43,593	152,658,312
doc	5,398	1,544	6	48,536,165	5,310,059	13,585	406,673,898
groff-1	1,974	671	9	4,525,946	214,219	63,116	37,655,949
groff-2	2,110	722	9	6,047,864	205,487	70,759	46,416,495
groff-3	2,370	759	10	7,977,220	304,948	94,298	63,641,709
idl	1,011	525	6	11,809,125	2,991,355	38,157	151,419,127
idraw	6,473	2,279	12	18,699,046	1,490,460	51,664	184,930,700
morpher	4,050	1,200	6	6,613,548	425,072	7,807	52,131,648
rtsh	1,390	59	3	52,357,435	5,547,705	489	822,054,191

Table 2: Detailed statistics on number of instructions and breaks in control for each program measured.

Metric	congress	doc	groff-3	idl	idraw	morpher	rtsh
Mean	8.15	7.55	7.60	10.20	9.14	7.40	14.20
Median	6.00	6.00	5.00	9.00	5.00	4.00	7.00
StdDev	7.57	5.99	7.19	6.06	15.94	8.76	30.94

Table 3: Number of instructions between breaks in the absence of any control flow prediction.

Program	Unique Names	Single Target	Static	Inf 1-Bit BTB	Inf 2-Bit BTB
CONGRESS	18.5	29.5	73.7	76.7	88.8
DOC	56.2	76.7	93.2	92.2	96.5
GROFF-3	6.6	31.2	86.4	79.0	95.3
IDL	†	99.9	99.9	99.9	99.9
IDRAW	34.9	83.5	94.6	95.6	98.0
MORPHER	70.3	92.0	96.7	96.6	97.7
RTSH	2.9	51.1	93.6	96.0	98.0
Mean	31.6	66.3	91.2	90.9	96.3

Table 4: Percentage of indirect function calls predicted using compile-time, profile-based and dynamic prediction. († could not be computed)

Program	Using GROFF-1	Using GROFF-2	Using GROFF-3	Using All Combined
GROFF-1	**87.4**	84.4	86.6	86.6
GROFF-2	82.5	**86.4**	85.6	86.4
GROFF-3	79.6	85.6	**86.4**	86.3

Table 5: Percentage of I-calls predicted for GROFF when using static prediction with different input files.

Metric	CONGRESS	DOC	GROFF	IDL	IDRAW	MORPHER	RTSH
Break	9.02	8.36	8.42	9.04	9.86	11.17	9.12
Call	14.45	14.62	14.57	11.27	18.92	15.16	14.30

Table 6: Mean number of instructions above the I-call in the instruction stream before hitting a break or another function call. Both of these include the start of a procedure as break point.

	2Bit 2Brs		Unique Name		Single Target		Static ICalls	
Program	NIBB	%BP	NIBB	%BP	NIBB	%BP	NIBB	%BP
congress	67.7	88.0	69.7	88.3	70.9	88.5	76.2	89.3
doc	56.0	86.5	95.2	92.1	127.6	94.1	176.4	95.7
groff-3	74.0	89.7	75.8	90.0	83.2	90.9	106.7	92.9
idl	45.7	77.7	45.7	77.7	467.3	97.8	468.0	97.8
idraw	71.0	87.1	88.7	89.7	136.0	93.3	154.8	94.1
morpher	51.9	85.8	74.0	90.0	85.1	91.3	87.9	91.6
rtsh	71.6	80.2	72.6	80.4	95.0	85.1	130.5	89.1
HM / Avg	60.7	85.0	71.1	86.9	106.5	91.6	125.5	92.9

Table 7: Measurements of breaks that can be predicted using compile-time and static I-call prediction with 2-Bit branch prediction.

resources [6]. It is likely that our prediction architecture would benefit from a small *2-bit* prediction mechanism for indirect function calls.

The last prediction mechanism we considered was *branch target address registers* (BTAR's). We assumed architectures would implement a small number of BTAR's, and they would likely not be saved across procedure calls. Thus, there are two limits on using BTAR's to indicate intended branch targets. We assumed that the BTAR could be loaded anywhere in the previous basic block, providing a lower bound on the interval when the BTAR is loaded and the branch taken. Likewise, we assume a clever compiler might be able to load the BTAR immediately following the previous procedure call or return (because we assumed BTARs are not saved across function calls). Table 6 shows these two values (instructions since beginning of basic block and instructions since last call/return). In general, there are very few instructions in which to schedule the "prepare to jump" information before the first target instruction is needed.

For simple prediction of targets for indirect function calls, we found:

- Dynamic methods using the 2-bit branch target buffer were the most effective technique we considered. However, this style of prediction may be expensive to implement.
- By combining a simpler branch prediction technique with BTB's for indirect function calls, the resource demands become more realistic.
- Static profile-driven prediction was very accurate, and is used in the remainder of this paper.

4.3 Using Profiles to Eliminate Indirect Function Calls

Our prior measurements have shown the percentage of *I-calls* predicted using these different techniques. By comparison, Table 7 shows the percent of *total breaks* predicted. Using static prediction with prior profiles for I-calls accurately predicts half of the remaining breaks in control, doubling the number of instructions between breaks (assuming that the breaks are evenly distributed). Recall Figure 1. In this figure conditional branches and indirect function calls are predicted using the static profile-based technique just described. The second bar for each program indicates the additional delays incurred by breaks from indirect function calls and mispredicted conditional branches. While the third bar, eliminates delays from statically predicted indirect function calls. For architectures providing BTB's, the delay would be slightly smaller. Clearly, predicting branches is the foremost priority, but predicting indirect calls removes a substantial number of breaks.

The success of profile-based static prediction also indicates that many methods could be successfully compiled inline, even without compile-time type analysis. We can convert an indirect function call, e.g.,`object -> foo();` to a conditional procedure call with a run-time type check:

```
if (typeof(object) == A )
    object -> A::foo();
else
    object -> foo();
```

This transformation is useful for three reasons. First, once this code transformation has been performed, function call `A::foo()` can be inlined. Secondly, If there is a high likelyhood of calling `A::foo()`, this code sequence is less expensive on most RISC architectures, using on 4-5 instructions rather than 5-8. Lastly, if an architecture provides *branch prediction* but does not support prediction for indirect function calls, the transformed code can avoid many misprediction penalties. Existing branch prediction hardware may be able to improve on strictly profile-based prediction because it can accommodate bursts of calls to a secondary call target.

Although inlining functions is useful, it does not always reduce program execution time [10]. However, many indirect function calls in C++ tend to be very short, because programmers are more likely to employ proper data encapsulation techniques. We believe automatic inlining will be more useful for C++ than C. Further, on most architectures, the converted indirect-function call is more efficient if there is a high likelyhood of calling the most common function (`A::foo()` above).

We constructed the following cost models for handling I-calls and used this to optimize I-calls in more detail. Assume the cost of a direct method call, C_{dmc}, is 2 instructions. This comes from an extra instruction needed to compute the object pointer which is passed to the call instruction. The cost of an indirect method call, C_{imc}, is 7 instructions, and is because of the extra instructions needed to compute the pointer addresses and future branch target. The cost of an "if," C_{if}, as shown in the previous example, is 3 instructions, including an indirect load for the object pointer, a load of a constant and the comparison. The penalty for mispredicting a conditional branch or an indirect function call, C_{miss}, is 10 instruction times. This is because we assume mispredicted breaks can cause a 10 cycle pipeline delay. From this we get the cost for indirect method calls with no prediction to be

$$C_{nopredict} = C_{imc} + C_{miss} \, .$$

Since the indirect call is not predicted, it is considered to be mispredicted. By comparison, with the static profile-based prediction mechanism as discussed in the previous section, the cost becomes

$$C_{predict}(P) = C_{imc} + (1 - P)C_{miss} \, .$$

if there is a P% probability of accurately predicting the call target for a call site. The cost for converting an indirect method call to an 'if,' as done above, would be

$$C_{use-if}(P) = C_{if} + PC_{dmc} + (1 - P)(C_{miss} + C_{nopredict}) \, ,$$

however, we can use the existing profile information to compute 'Q', the percentage of the *second most* likely call target being selected. Note that Q has to be less than or equal to min(P,(1-P)), else it would be the most likely target selected. It is interesting to note that Q might be a high percentage of the remaining I-calls. For example, we may have P = 40%, with the next most likely branch occurring Q = 35% of the time. This means that $35/(100 - 40)$, or 58% of the remaining 60% of I-calls are correctly predicted. Thus, the cost for converting an indirect function call to an 'if' construct is actually

$$\begin{aligned} C_{use-if}(P, Q) &= C_{if} + PC_{dmc} + \\ &\quad (1 - P)(C_{miss} + C_{predict}(Q/(1 - P))) \, . \end{aligned}$$

Figure 2 shows the costs for $C_{nopredict}$ (the horizontal line), $C_{predict}$ (the lower line) and the boundaries for $C_{use-if}(P, Q)$ for $Q = min(P, (1 - P))$ the best case, and $Q = 0.0$ the worst case. In the worst case, our "if conversion" is the same as $C_{use-if}(P)$. This is a hypothetical case when there are so many second most likely targets that Q is approximately equal to zero percent. In the graph the best case if conversion, $C_{use-if}(P, Q)$ for $Q = min(P, (1 - P))$, is achieved when an I-call site has only 2 targets. This can only happen when $P >= 0.5$. So when $P < 0.5$, at best Q can only equal P, and the remaining $1-(P+Q)$ I-call targets cannot be predicted. This is the reason the line in the graph for the best case if conversion changes slope at $P = 0.5$. From the graph, one can see that it is always a benefit to do static I-call prediction. Given the values for C_{if}, C_{dmc}, C_{miss} and C_{imc} that we used, when $P = 0.6$, depending on how accurate one can predict the second most likely target Q, it can be better to do the if conversion on the I-call rather than only predicting the most likely target. From the graph, one can see that, $C_{use-if}(P)$ eventually intersects $C_{predict}(P)$, where it is always beneficial to do the if conversion on an architecture that provides static prediction. With our architecture assumptions, this occurs at $P = 0.86$. The lines $C_{nopredict}$ and $C_{use-if}(P)$ (worst case) also eventually intersect, where it is always beneficial to do the if conversion on an architecture that does not provide static prediction. With our architecture assumptions, this occurs at $P = 0.52$. Thus, doing the if conversion on an architecture that does not provide static prediction gives the user, in a sense, the benefit of static prediction. As mentioned, the architecture we've considered is similar to the Digital AXP 21064. Other architectures, including the Intel Pentium, also issue two instructions per clock, and some newly announced architectures, such as the IBM RIOS-II issue up to *eight* instructions per cycle. On these architectures, the advantage of "if-conversion" occurs with much lower probabilities for P.

In general, prediction information can greatly reduce the penalty for indirect function calls. As noticed from the graph it is always beneficial to predict the destination for an I-call. Accurate profile-based measurements expose other optimizations when the accuracy of predicting the most frequently called function exceeds 80-90%. Table 4 shows this occurs for many of the programs we measured. This transformation also provides opportunity for inlining the body of the function, allowing the compiler to customize the parameters to the function, avoid register spills and the like.

The right-most bar for each program shown in Figure 1 shows the effect of applying this transformation where appropriate, based on our model, for each call site in the programs. By comparison, the second bar from the right shows the benefit of using prediction and unique name elimination, without using the if conversion. Although the advantage is small, similar cost/benefit analysis can be used to determine the advantage of additional function inlining. Note, there will be a greater advantage in using the if conversion when the architecture does not support static prediction.

5 Conclusions and Future Work

As object-oriented programs become more common, there will be an increasing need to optimize indirect function calls. This will become even more important as processor pipeline depths increase and superscalar instruction issue and speculative execution become more common. Existing branch prediction mechanisms accurately prediction $95\% - 97\%$ of conditional branches. Because branch prediction is so successful, accurate prediction of the remaining breaks

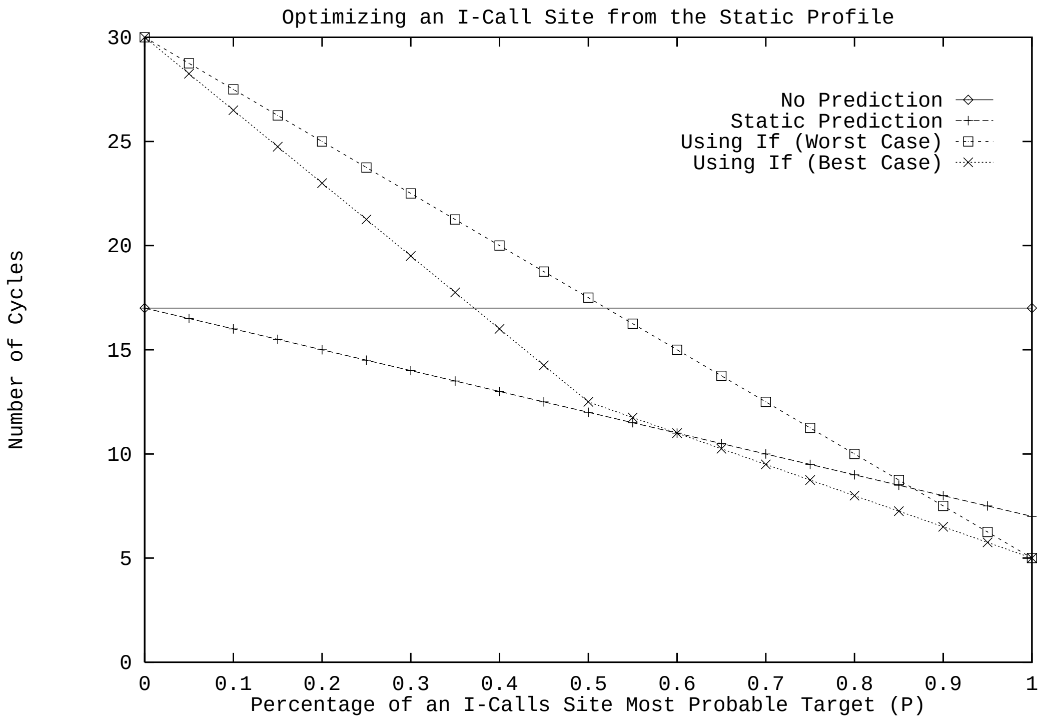

Figure 2: Cost, in instructions and additional delay, for different indirect function call methods

in control-flow becomes increasingly important as processors begin to issue more instructions concurrently. Eliminating the misprediction penalty for indirect function calls in C++ programs can remove 10% of the remaining breaks in control in a C++ program.

We found that static profile-based prediction mechanisms worked well for the collection of existing C++ programs we examined. We saw additional improvements by combining compiler optimization techniques (unique name elimination and 'if conversion') with static indirect call prediction. The information from profile-based prediction is also useful for other code transformations, such as inlining and better register scheduling. Our results show that we get an average of 10% improvement in the number of instructions executed for a program by using our I-call prediction/optimization techniques.

We recommend that compilers for highly pipelined, speculative execution architectures : use profile-based static prediction methods to optimize C++ programs, use link-time information to remove indirect function calls, and customize call-sites using 'if conversion' based on profile information. Furthermore, we hope the architecture and benchmarking community expands benchmark suites to include modern programming languages such as C++, Modula-3 and the like, because these languages exercise different architectural features than C or Fortran programs.

Acknowledgements

We would like to thank Ben Zorn, John Feehrer and the reviewers for comments on the paper. We'd also like to thank James Larus for developing QPT and helping us solve the various problems we encountered applying it to such large programs. This work was funded in part by NSF grant No. ASC-9217394, and is part of a continued effort to make languages such as C++ suitable for scientific computing.

References

[1] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The tera computer system. In *Proc. 17th Annual Symposium on Computer Architecture, Computer Architecture News*, pages 1–6, Amsterdam, Netherlands, June 1990. ACM, ACM.

[2] T. Ball and J. R. Larus. Branch prediction for free. In *1993 SIGPLAN Confernce on Programming Language Design and Implementation*. ACM, June 1993.

[3] T. Ball and J.R. Larus. Optimally profiling and tracing programs. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 59–70, Albuquerque, NM, January 1992.

[4] Michael Burke. An interval-based approach to exhcaustive and incremental interprocedural analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[5] Brad Calder and Dirk Grunwald. Branch alignment. Technical Report (In Preperation), Univ. of Colorado-Boulder, 1993.

[6] Brad Calder and Dirk Grunwald. Fast & accurate instruction fetch and branch prediction. CU-CS xxx, Univ. of Colorado, October 1993. (In preperation).

[7] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Exploiting behavioral differences between C and C++ programs. Technical Report (In Preperation), Univ. of Colorado-Boulder, 1993.

[8] Craig Chambers and David Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, Portland, OR, June 1989.

[9] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 150–164, White Plains, NY, June 1990.

[10] Jack W. Davidson and Anne M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering*, 18(2):89–102, February 1992.

[11] Jack W. Davidson and D.B. Whalley. Reducing the cost of branches by using registers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 182–191, Los Alamitos, CA, May 1990.

[12] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, Boston, Mass., October 1992. ACM.

[13] Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.

[14] Urs Hölzle, Craig Chambers, and David Unger. Optimizating dynamically-typed object-orientred languages with polymorphic inlines caches. In *ECCOP '91 Proc.* Springer-Verlag, July 1991.

[15] David R. Kaeli and Philip G. Emma. Branch history table prdiction of moving target branches due to subroutine returns. In *18th Annual International Symposium of Computer Architecture*, pages 34–42. ACM, May 1991.

[16] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *19th Annual International Symposium of Computer Architecture*, pages 46–57, Gold Coast, Australia, May 1992. ACM.

[17] Johnny K. F. Lee and Alan Jay Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, ??(??):6–22, January 1984.

[18] David J. Lilja. Reducing the branch penalty in pipelined processors. *IEEE Computer*, pages 47–55, July 1988.

[19] M. S. Lam M. D. Smith, M. Horowitz. Efficient superscalar performance through boosting. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 248–259, Boston, Mass., October 1992. ACM.

[20] Scott McFarling and John Hennessy. Reducing the cost of branches. In *13th Annual International Symposium of Computer Architecture*, pages 396–403. ACM, 1986.

[21] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Boston, Mass., October 1992. ACM.

[22] Hemant D. Pande and Barbera G. Ryder. Static type determination for C++. Technical Report LCSR-TR-197, Rutgers Univ., February 1993.

[23] Chris Perleberg and Alan Jay Smith. Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412, April 1993.

[24] Barbera G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979.

[25] J. E. Smith. A study of branch prediction strategies. In *8th Annual International Symposium of Computer Architecture*. ACM, 1981.

[26] D. W. Wall. Limits of instruction-level parallelism. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, Boston, Mass., 1991. ACM.

[27] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch predictions. In *19th Annual International Symposium of Computer Architecture*, pages 124–134, Gold Coast, Australia, May 1992. ACM.

[28] Tse-Yu Yeh and Yale N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *19th Annual International Symposium on Microarchitecture*, pages 129–139, Portland, Or, December 1992. ACM.

[29] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *20th Annual International Symposium of Computer Architecture*, pages 257–266, San Diego, CA, May 1993. ACM.