Quantifying Behavioral Differences Between C and C++ Programs (revised)

Brad Calder, Dirk Grunwald, and Benjamin Zorn

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

{calder,grunwald,zorn}@cs.colorado.edu

January 18, 1995

Abstract

Improving the performance of C programs has been a topic of great interest for many years. Both hardware technology and compiler optimization research has been applied in an effort to make C programs execute faster. In many application domains, the C++ language is replacing C as the programming language of choice. In this paper, we measure the empirical behavior of a group of significant C and C++ programs and attempt to identify and quantify behavioral differences between them. Our goal is to investigate whether optimization technology that has been successful for C programs will also be successful in C++ programs. We furthermore identify behavioral characteristics of C++ programs that suggest optimizations that should be applied in those programs. Our results show that C++ programs exhibit behavior that is significantly different than C programs. These results should be of interest to compiler writers and architecture designers who are designing systems to execute object-oriented programs.

1 Introduction

The design of computer architecture is typically driven by the needs of various programs and programming languages. A significant amount of research, both in compiler optimization and in architecture design, has been conducted with the specific goal of improving the performance of existing conventional programs [1, 2]. Early studies of program behavior [3, 4, 5, 6] have guided architectural design, and the importance of measurement and simulation has permeated architectural design philosophy [7]. In particular, the IBM 801, Berkeley, and Stanford RISC projects were all guided by studies of C and FORTRAN programs [8]. More recent studies have used the SPEC program suite. This set of programs, widely used to benchmark new hardware platforms and compiler implementations, consists of a mixture of C and FORTRAN programs [9, 10].

More recently, object-oriented programming, and specifically the language C++, has become widely used and is replacing procedural languages such as C in a number of application areas including user-interfaces, data structure libraries, scientific computing [11], and operating systems [12]. While the C++ language is a superset of C, additional features provided in the language support a programming style that is very different from that of programming in C.

Before conducting the research reported in this paper, our hypothesis was that C++ programs behave quite differently from C programs and that these differences may have a significant impact on performance. In this paper we investigate the behavior of 12 C++ programs and 11 C programs, including the SPECint92 benchmark suite. While this small sample of programs clearly does not capture the behavior of all C and C++ programs, the sample of programs measured is representative of a variety of common tasks implemented in these languages. Furthermore, our measurements show that the C++ programs we use represent a broad variety of C++ programming styles, ranging from mostly procedural to highly object-oriented.

In this paper, we seek to quantify behavioral differences between programs and languages and investigate how existing hardware and software optimizations that support C programs will also support C++ programs. Hardware designers and optimizing compiler writers can use information about how programs behave to exploit that behavior effectively. For example, register windows were included in the RISC architecture because it was observed that C programs tended to use a small set of stack frames for long periods of time. Such quantitative measurements of program behavior are of great value to compiler writers and hardware designers. We measure the dynamic execution of the programs, including function size, basic block size, instructions between conditional branches, call stack depth, use of indirect function calls, use of memory operations, and measurements of cache locality. To highlight differences between C and C++ programs, we break down our measurements into sub-categories, such as comparing behavior in methods versus non-methods.

Because, in statistical terms, our program sample size is relatively small, we cannot make definitive claims about behavioral differences based on the results presented here. Nevertheless, our results are compelling, and we hope they will encourage others to attempt to confirm or deny our claims.

Our results indicate that, based on the sample of programs that we studied, C and C++ programs can differ significantly in some aspects of their behavior. Differences between the languages are hightened when we classify C++ programs as "Object Oriented" and "Non-Object Oriented," as we do in Section 4.11. The differences suggest whether or not hardware and software optimizations that have been effective for C programs will also be effective for C++ programs. We also conclude that new optimizations, not even considered as necessary in C programs, are appropriate for C++ programs. Hardware designers and compiler writers may use our results to construct a new generation of systems that execute C++ programs more efficiently. We have conducted some of this research in related publications [13, 14, 15], but a detailed discussion is beyond the scope of this paper.

Section 2 describes related work in the area of program behavior measurement. Section 3 describes the tools we used to collect our measurements and the programs that we measured. Section 4 includes the basic data that we gathered as well as our interpretation of it, while Section 5 discusses the implications of our results. Finally, we summarize our conclusions in Section 6.

2 Background

In this section, we describe related work investigating the empirical behavior of programs. This work falls roughly into two categories: measurements of different aspects of program behavior and measurements of instruction set usage on particular architectures.

2.1 Program Behavior Measurement

There have been a large numbers of studies of various aspects of the behavior of different kinds of programs. Knuth measured both static and dynamic behavior of a large collection of Fortran programs [6]. Among other things, he concluded that programmers had poor intuition about what parts of their programs were the most time-consuming, and that execution profiles would significantly help programmers improve the performance of their programs.

Much of the work in the area of program behavior measurement prior to 1984 is summarized in Weicker's paper describing the Dhrystone benchmark [16]. Weicker used these results to create a small benchmark program, Dhrystone, that was intended to simulate the average systems program behavior reported in previous work. In 1988, Weicker updated the Dhrystone benchmark, creating version 2.0, which is the version of Dhrystone that we measure.

Since 1984, measurements of a number of aspects of program behavior have appeared. The SPEC benchmark suite [9, 10], used in recent years to compare the performance of new computer architectures, has been investigated both in terms of instruction set usage [17, 18] and cache locality [19, 20]. We mention these measurements because our measurements include the SPECint92 programs, and in some cases can be compared directly with these previous results.

While other work in this area has concentrated on the behavior of programs written in a single language [3, 5], our focus is on comparing the relative behavior of programs written in C and C++, two closely related languages. In particular, we are interested in measuring aspects of behavior that might be exploited either with better hardware or with aggressive software optimizations.

2.2 Instruction Set Architecture Measurement

Another area of related work is the measurement of instruction set architectures. This practice is commonly used by architecture designers to understand how the features of the hardware are being used. Clark and Levi report on instruction set use in the VAX-11/780 [21] and conclude that different programs use different parts of the large VAX instruction set. Weicek investigates how six compilers use the VAX-11 instruction set [22]. Similarly, Sweet reports on the static instruction set usage of the Mesa instruction set [23], while McDaniel described the dynamic instruction set usage in Mesa [24] programs. All of these analyses were conducted to give architecture designers insight into how to improve the next-generation architecture. This approach to architecture design has become so familiar that the method can now be found described in the popular textbook, "Computer Architecture: A Quantitative Approach" [7].

More recently, published measurements of this kind have concentrated on the SPEC benchmark programs (e.g., [17]). Because the SPEC benchmarks are widely used to compare system performance of workstations, compiler writers and architects study these and similar programs in detail. Modifying compilers or architectures using information from the SPEC suite will lead to good SPEC performance, but may not improve C++ programs. For example, prior to the introduction of the SPEC program suite, many manufacturers used the original Dhrystone program to compare system performance. Numerous compiler optimizations, primarily optimizing string operations that occur frequently in the Dhrystone program, were implemented; these optimizations greatly benefited Dhrystone performance, but did not dramatically improve programs with different behavior.

Our measurements include aspects of instruction set usage that indicate what differences hardware designers will see between C and C++ programs. Furthermore, the C++ programs we have collected could serve as an initial set of C++ benchmarks in the same way that the SPECint92 and SPECfp92 benchmarks are currently used.

3 Methods

In this section, we describe the methods used to collect the data presented in the next section. Specifically, we describe the programs measured, the tools used to collect the measurements, and the format of the data presented.

3.1 The Programs

We have measured 12 C++ programs, 11 C programs, and the DHRYSTONE benchmark. The C programs include all of those present in the SPECint92 benchmark suite. This large program suite precluded investigating additional programs, such as the C and FORTRAN programs in the SPECfp92 benchmark suite; other studies provide such comparisons [17, 25]. The function of these programs and the input datasets we used are described in Tables 1 and 2. We realize that empirical studies of programs and programming languages are fraught with problems—how can we analyze enough programs to produce meaningful results yet still manage the great quantity of generated data? Thus, throughout this paper, we encourage the reader to keep in mind that our results are specific to the set of programs we chose to measure.

We sought large programs, both in terms of source code size and in terms of executable size; where possible, we selected programs that were in widespread use and familiar to a broad audience of users. We sought programs measured in other studies; the SPECint92 programs met this requirement. We sought programs written in both C and C++ that provide similar functionality. For example, we measure DITROFF, a troff processor written in C, and GROFF, a troff processor written in C++. In this particular case, the input file provided to both programs was exactly the same. Other C and C++ programs are also paired in the same way—XFIG (C) and IDRAW (C++) are picture drawing programs; XTEX (C) and DOC (C++) are document previewers (DOC was used in this way, although it also has editor capabilities); and GCC (C) and CFRONT (C++) are translators. We also included programs written using an object-oriented style in

BURG	A program that generates a fast tree parser using BURS technology (version 1.0). It is commonly used to construct optimal instruction selectors for use in compiler code generation. The input used was a grammar that describes the VAX instruction architecture.
DITROFF	C version of the "ditroff" text formatter. The input used was the same collection of manual pages given to GROFF.
TEX	A widely used text-formatting program, version 3.141. The input used was "dvips.tex," a forty-five page manual describing a program used to format TEX output.
XFIG	An interactive structured graphics editor (version 2.1.8). The input involved loading a previously drawn figure and modifying it.
XTEX	An interactive DVI previewer (version 2.18.5). The input was an interactive session in which a 10-page document was viewed forwards and backwards.
COMPRESS	A file compression program, version 4.0, that uses adaptive Lempel-Ziv coding. The test input required compressing a one million byte file.
EQNTOTT	A translator from a logic formula to a truth table, version 9. The input was the file int_pri_3.eqn.
ESPRESSO	A logic optimization program, version 2.3, that minimizes boolean functions. The input file is an example provided with the release code (cps.in).
GCC	A benchmark version of the GNU C Compiler, version 1.35. The measurements presented show only the execution of the "cc1" phase of the compiler. The input used was a preprocessed 4832-line file (1stmt.i).
LI	A Lisp interpreter that is an adaptation of XLISP 1.6 written by David Michael Betz. The input measured was a solution to the N-queens problem where N=8.
SC	A spreadsheet program, version 6.1. The input involved cursor movement, data entries, file handling, and some computation.
DHRYSTONE	Version 2.0 of a small synthetic benchmark program intended to mimic the observed behavior of systems programs. The program does not require input.

Table 1: General information about the C programs.

BISIM	A mixed mode switch-level and gate-level simulator for MOS and Bipolar circuits. The input used was gate-level simulation of a 64-bit CMOS RISC microprocessor.
CFRONT	The AT&T C++ to C conversion program, version 3.0.2. The input used was the file `groff.c`, provided as part of the GNU `troff` implementation, after being preprocessed with `cpp`.
DELTABLUE	A version of the "delta-blue" constraint solution system written in C++. We used the example program that comes with the DELTABLUE system.
DOC	Another interactive text formatter, based on the InterViews 3.1 library. The input involved interactively browsing and modifying a 10-page document.
GROFF	Groff Version 1.9 — A version of the "ditroff" text formatter. The input used was a collection of manual pages.
IDL	Sample backend for the Interface Definition Language system distributed by the Object Management Group. Input was a sample IDL specification for an early release of the Fresco graphics library.
IDRAW	Interactive structured graphics editor, based on the InterViews 2.6 library. The input involved drawing and editing a simple picture.
IXX	Part of the Fresco graphics library, distributed with X11R6. The "ixx" tool converts an IDL specification to C++ code, much as the IDL application does—however, it was written by different authors and is structured differently. We used input files from the Fresco release.
LIC	Part of the Stanford University Intermediate Format (SUIF) compiler system. It is a **l***inear* **i***nequality* **c***alculator*. We used the largest distributed example as an input.
MMAKE	A Makefile generation program. The input used was a directory with many files to be compiled and linked together.
MPSDEMO	An example distributed with the OSE-V4.0b3 object library. The program is an experiment in translating an Object Z specification into C++ code. The Object Z specification was that of a mobile phone system.
PORKY	Part of the Stanford University Intermediate Format (SUIF) compiler system. It performs a variety of compiler optimizations. We used it to perform constant folding, constant propagation, reduction detection and scalarization for a large C program.

Table 2: General information about the C++ programs.

the C language (XTEX). Finally, we sought C++ programs representing a range of object-oriented programming styles. Because we are familiar with the origin and history of development of many of the C++ programs we measured, we have insight into the programming style used in them.

The C++ programs measured represent a broad range of language use. In particular the programs are gathered from seven independent sources including DEC Western Research Lab (BISIM, MMAKE), Mark Linton's InterViews group (IDRAW, DOC, and IXX), Sun Microsystems (IDL), Stanford University (PORKY and LIC), AT&T (CFRONT), and the GNU project (GROFF). The C++ programs vary widely in their use of member functions and virtual functions, but in general make little use of templates or exceptions, relatively recent features in C++. The programs were all compiled with optimization enabled, therefore optimizations such as member-function inlining were enabled.

Information about the program executions we measured is provided in Table 3. Here and throughout the paper we show results from a single input to each program. We have gathered data from multiple inputs to the same program for some of the programs to see how much variation in behavior different inputs will cause, but those results are not reported here. In general, we saw little variation in the behaviors we measured based on program input set. We have made the raw data for these additional runs available on the Internet (see Section 6).

With the information presented in Table 3, some important characteristics of the test programs become apparent. One important observation is that several of the C++ programs, and most notably CFRONT, make few indirect function calls. From this behavior we immediately conclude that few or no virtual functions are defined in these programs, and thus the programs make little use of dynamic dispatch, an important characteristic of the object-oriented paradigm. Other measurements of CFRONT, presented later, confirm this observation. One explanation of this behavior is that CFRONT was used to bootstrap a C++ translator, and as such, it was to the developers' advantage to use only a subset of the full set of C++ features in the translator.

Of the other C++ programs, we should note that DOC and IDRAW are both interactive X-windows programs implemented using the InterViews class library [26]. InterViews is a large class library that has evolved and changed with the evolution of C++ itself. We note that Mark Linton, the designer of InterViews, has been a very active user of C++ for many years and has been instrumental in shaping the design of the language. He is also very familiar with the features provided and the InterViews library reflects this knowledge. IDRAW is an early InterViews program, implemented with version 2.6 of the library. In InterViews 2.6, graphical objects were larger, encompassing more functionality in each object. In the later InterViews 3.1, the larger graphical objects were replaced by small "glyphs"—these simple objects are composed to form more complex objects. For example, rather than provide a single "button" object with multiple parameters to define the button, the user creates a drawable object and "wraps" it with an object that handles input events. In this library, software reuse is encouraged by constructing small objects and composing them. Thus, the DOC program, which was written using InterViews 3.1, has considerable "software reuse" since each primitive object can be composed for a variety of functions.

Program Name	Static Insns	Statistics Collected During Program Execution: Total Insns	Total Func Calls	Insn/ Call	Total I-Calls	Insn/ I-Call
BURG	26,936	390,772,349	6,342,378	61.6	8,753	44644.4
DITROFF	27,241	38,893,571	663,454	58.6	6,920	5620.5
TEX	101,428	147,811,789	853,193	173.2	0	—
XFIG	383,767	33,203,506	536,004	61.9	33,312	996.7
XTEX	293,311	23,797,633	207,047	114.9	6,227	3821.7
COMPRESS	13,144	92,629,716	251,423	368.4	0	—
EQNTOTT	19,172	1,810,540,472	4,680,514	386.8	3,215,048	563.1
ESPRESSO	60,674	513,008,232	2,094,635	244.9	84,751	6053.1
GCC	186,066	143,737,904	1,490,292	96.4	80,809	1778.7
LI	33,235	1,354,926,022	31,857,867	42.5	919,965	1472.8
SC	59,291	917,754,841	12,903,351	71.1	13,785	66576.3
BISIM	250,342	2,674,963,554	91,920,565	29.1	638,542	4189.2
CFRONT	225,064	16,529,573	174,260	94.9	12	—
DELTABLUE	20,784	86,475,952	3,298,440	26.2	2,213,640	39.1
DOC	445,817	459,765,082	10,178,613	45.2	5,123,163	89.7
GROFF	121,191	56,926,360	1,126,050	50.6	220,495	258.2
IDL	79,381	21,138,427	873,288	24.2	497,021	42.5
IDRAW	631,667	58,232,402	1,560,135	37.3	479,279	121.5
IXX	111,808	51,703,946	1,032,525	50.1	73,759	701.0
LIC	384,058	4,267,588	86,833	49.1	549	7773.4
MMAKE	41,803	9,483,858	80,555	117.7	11,205	846.4
MPSDEMO	91,795	919,445	25,637	35.9	4,222	217.8
PORKY	216,678	163,702,710	6,722,596	24.4	924,808	177.0
DHRYSTONE	19,310	608,057,060	18,000,726	33.8	0	0.0
C Mean	109,478	497,006,912	5,625,468	152.8	397,233	14614.1
C++ Mean	218,365	300,342,408	9,756,624	48.7	848,891	1314.2

Table 3: Information about Program Size and Executions. Static Insns shows how many instructions each program contains. Total Insns shows how many instructions were executed during each program run. Total Func Calls shows how many function calls each program executed, including indirect function calls. Insn/Call is the number of instructions executed per call (Total Insns divided by Total Func Calls). Total I-Calls shows how many indirect function calls each program executed and Insn/I-Call shows how many instructions per indirect call were executed. Note that programs with less than 100 total I-calls are indicated with a dash in the Insn/I-Call column and not included in the means. All programs were executed on a DEC 3000-400 workstation using an Alpha AXP21064 processor. The C programs were compiled with the DEC `cc` command with the `-O` option. The C++ programs were compiled with the DEC `cxx` C++ compiler, again using the `-O` option.

Interestingly, the table also shows that EQNTOTT, a C program, performs a very large number of indirect function calls. This behavior is explained by the fact that the main computation of EQNTOTT is sorting (about 95% of the execution time is spent in the library routine `qsort`). The function used to compare two values in the `qsort` routine is passed as an argument to `qsort` and called indirectly.

3.2 Tools and Metrics

In the next section, we present results including average function size, average basic block size, percentage of stores and loads, and others. Many of the measurements presented are *dynamic*, or collected from the execution of the programs. We used the ATOM tool [27] to collect program information. ATOM instruments the binary executables of programs to measure the behavior of programs executing on the DEC Alpha architecture. In earlier versions of this work, we used the the QPT trace generator [28, 29], which allowed us to measure programs on the MIPS instruction architecture.

With ATOM or QPT, we are able to identify all the function calls, basic block transitions, instruction fetches, data loads and stores, and other operations that occur during program execution. We are also able to determine when indirect function calls occur and the target function of the indirect call. We identify and distinguish when C++ functions, C++ member functions (referred to by the more widely used term "methods" in the remainder of this paper), and C functions are being invoked. Note that we also use the terms "function" and "procedure" interchangeably in this paper, even though in some languages these terms refer to semantically different constructs. A C++ function is a function compiled by the C++ compiler, while a C function is compiled by a conventional C compiler. C++ member functions (or methods) are functions associated with objects. This information allows us to classify behavior such as basic block size by the type of function in which the basic block occurs. We break down our measurements in this way throughout this paper.

It is important to note that we do not separate the impact of the particular architecture and compiler chosen for these studies. Thus, we are not only measuring the behavior of the particular programs, but also aspects of the hardware on which they execute (the 64-bit Alpha architecture) and the compiler with which they were compiled (DEC C and C++). For the most part, the results presented are characteristic of modern RISC instruction set architectures (except for the memory operations, described in Section 4.7; and the I-cache measurements, described in Section 4.9.1). Likewise, the compiler used was quite characteristic of modern commercial optimizing compilers. As described in §4.10, we also compared the execution of one program, GROFF, using both the DEC C++ and GNU G++ compilers. This comparison demonstrated little difference between the two compilers.

To measure cache performance, we have constructed a direct-mapped cache simulator and compared its output to that of Tycho, a widely used, all-associativity cache simulator [25, 30] to determine its correctness. Due to the length of this study, we chose to simulate and give results for only direct-mapped caches.

Program	C	C++	Leaf	Non-Leaf	Meth	Non-Meth	Indirect	Direct	All
BURG	100.0	—	72.3	27.7	—	100.0	0.1	99.9	100.0
DITROFF	100.0	—	14.7	85.3	—	100.0	1.0	99.0	100.0
TEX	100.0	—	20.0	80.0	—	100.0	0.0	100.0	100.0
XFIG	100.0	—	35.5	64.5	—	100.0	6.2	93.8	100.0
XTEX	100.0	—	50.6	49.4	—	100.0	3.0	97.0	100.0
COMPRESS	100.0	—	0.1	99.9	—	100.0	0.0	100.0	100.0
EQNTOTT	100.0	—	85.3	14.7	—	100.0	68.7	31.3	100.0
ESPRESSO	100.0	—	75.0	25.0	—	100.0	4.0	96.0	100.0
GCC	100.0	—	28.9	71.1	—	100.0	5.4	94.6	100.0
LI	100.0	—	13.4	86.6	—	100.0	2.9	97.1	100.0
SC	100.0	—	29.1	70.9	—	100.0	0.1	99.9	100.0
BISIM	11.8	88.2	27.5	72.5	57.1	42.9	0.7	99.3	100.0
CFRONT	26.1	73.9	35.7	64.3	41.7	58.3	0.0	100.0	100.0
DELTABLUE	15.5	84.5	24.8	75.2	77.7	22.3	67.1	32.9	100.0
DOC	20.6	79.4	28.6	71.4	77.7	22.3	50.3	49.7	100.0
GROFF	32.5	67.5	31.7	68.3	55.0	45.0	19.6	80.4	100.0
IDL	14.8	85.2	69.9	30.1	82.5	17.5	56.9	43.1	100.0
IDRAW	22.6	77.4	36.0	64.0	71.6	28.4	30.7	69.3	100.0
IXX	69.3	30.7	33.3	66.7	28.1	71.9	7.1	92.9	100.0
LIC	22.8	77.2	42.5	57.5	62.9	37.1	0.6	99.4	100.0
MMAKE	46.2	53.8	41.7	58.3	50.0	50.0	13.9	86.1	100.0
MPSDEMO	14.4	85.6	24.3	75.7	77.9	22.1	16.5	83.5	100.0
PORKY	9.5	90.5	63.8	36.2	84.5	15.5	13.8	86.2	100.0
DHRYSTONE	100.0	—	77.8	22.2	—	100.0	0.0	100.0	100.0
C Mean	100.0	—	38.6	61.4	—	100.0	8.3	91.7	100.0
C++ Mean	25.5	74.5	38.3	61.7	63.9	36.1	23.1	76.9	100.0

Table 4: Percentage of Invocations (Computed Dynamically)

3.3 Explanation of the Data Presented

In the next section, results of measurements of the programs are presented in a uniform way. Because the tables are quite complex, we describe their format in some detail here. To understand this format, consider Table 4, which shows how the function calls in the programs can be classified.

Each table contains one row per program measured, with the C programs appearing first and the C++ programs under them. The SPECint92 programs are separated from the other C programs. At the bottom of the tables, measurements of the DHRYSTONE benchmark, the mean of the C program measurements, and the mean of the C++ program measurements are presented in separate rows. The means presented in these rows are the simple arithmetic means of the numbers in the column, without any weighting. The means were not weighted by program execution time because the execution times of the programs vary greatly, and if weighting was done, programs like BISIM would dominate the weighted mean. Also, we decided to present only the arithmetic mean, although other kinds of more robust means, such as the harmonic mean, or a mean that discards high and low values, could have been chosen. We feel that the arithmetic mean conveys the appropriate information; furthermore, it is widely used and understood. We are also making the raw data for these tables publicly available on the Internet so that researchers interested in other statistical characterizations can make them.

Each row of the table is divided into ten columns. The first column, is the program name. The next eight columns break down the data into four pairs of two alternatives. The final column presents the overall value for the program without a breakdown. For example, Table 4 shows what kinds of functions are being called in the test programs. The four pairs of columns in the table represent the following comparisons:

Columns 2–3 A comparison of behavior in functions and methods defined in C++ versus functions defined in C. A "C Function" is a function compiled by the C compiler. A "C++ Function" is a function compiled by the C++ compiler that is not associated with a specific class; a "C++ Method" is a function associated with a specific class. C functions are often called by C++ functions because C++ programs use C libraries. We distinguish C++ methods and functions from C functions by looking at the function name. This comparison is not meaningful in the C programs, which obviously do not invoke any C++ functions. From the table, we see that approximately 74.5% of all function invocations in C++ programs are made to C++ methods and functions, while the other 25.5% are made to C functions.

Columns 4–5 A comparison of behavior in leaf (Leaf) versus non-leaf (Non-Leaf) functions. From the table, we see that in C++ programs, an average of 38.3% of calls are made to leaf functions, whereas in C, an average of 38.6% of calls are made to leaf functions.

Columns 6–7 A comparison of behavior in methods (Meth) versus non-methods (Non-Meth). Note that the non-methods column includes all non-method functions, and in particular, C functions. Again, since C programs do not contain methods, this distinction is not meaningful in those programs. From the table, we see that in C++ programs, an average of 63.9% of all function calls are made to methods. Furthermore, in some programs, such as IDL, this percentage is much higher (i.e., 82.5%). Note that IXX and CFRONT also perform significantly fewer method calls than the average for C++ programs.

Columns 8–9 A comparison of behavior in functions called indirectly versus directly. In general, C programs do not perform many indirect function calls, so these columns are mostly of interest in the C++ programs. From the table, we see that on average 23.1% of all function calls in C++ programs are indirect, whereas in C only 8.3% of all calls are indirect. Furthermore, we see that the C average is heavily influenced by the EQNTOTT benchmark, which we have already mentioned. Without EQNTOTT, the C mean becomes only 2.3%. We also see that BISIM, CFRONT, and LIC are unusual C++ programs, performing few indirect function calls.

The DHRYSTONE benchmark program is included in our results to show how well it emulates the behavior of either the C or C++ programs that we measured.

4 Results

Throughout this section we have intentionally resisted the urge to only present the summary information (i.e., C and C++ means) because it is important for the reader to be able to consider the behavior of individual programs as this information can be valuable as well. For example, DELTABLUE, DOC, IDL, and IDRAW are the C++ programs written in the most "object-oriented" style based on the large fraction of method calls and indirect function calls that each performs (see Table 4). Readers interested specifically in object-oriented uses of C++ may be interested in drawing conclusions based on the behavior of these programs separately (or even the DOC application alone).

Program	C Functions Number	C Functions %	C++ Functions Number	C++ Functions %	C++ Methods Number	C++ Methods %	All Functions Number
BURG	305	100.0	—	—	—	—	305
DITROFF	409	100.0	—	—	—	—	409
TEX	831	100.0	—	—	—	—	831
XFIG	4149	100.0	—	—	—	—	4149
XTEX	3301	100.0	—	—	—	—	3301
COMPRESS	148	100.0	—	—	—	—	148
EQNTOTT	211	100.0	—	—	—	—	211
ESPRESSO	550	100.0	—	—	—	—	550
GCC	1650	100.0	—	—	—	—	1650
LI	550	100.0	—	—	—	—	550
SC	511	100.0	—	—	—	—	511
BISIM	1140	28.9	582	14.7	2225	56.4	3947
CFRONT	312	31.8	297	30.3	371	37.9	980
DELTABLUE	221	67.4	15	4.6	92	28.0	328
DOC	2355	34.5	228	3.3	4244	62.2	6827
GROFF	575	32.8	278	15.8	902	51.4	1755
IDL	401	27.5	326	22.4	731	50.1	1458
IDRAW	2946	27.7	196	1.8	7503	70.5	10645
IXX	788	49.8	44	2.8	749	47.4	1581
LIC	2077	39.0	297	5.6	2958	55.5	5332
MMAKE	268	51.1	43	8.2	213	40.6	524
MPSDEMO	857	49.6	102	5.9	769	44.5	1728
PORKY	1720	46.4	322	8.7	1661	44.9	3703
DHRYSTONE	175	100.0	—	—	—	—	175
C Mean	1146.8	100.0	—	—	—	—	1146.8
C++ Mean	1138.3	40.5	227.5	10.3	1868.2	49.1	3234.0

Table 5: Number of C Functions, C++ Functions, and C++ Methods Defined in the Programs (Computed Statically).

4.1 Static Function Number and Size

In this section, we investigate the static properties of the procedures defined in the programs, both in terms of type and size. Knowing what kinds of routines are being defined helps us better understand how the features of the languages are being used, while knowing the static sizes of these routines helps us understand if optimizations such as inlining will be effective.

Table 5 shows the number of different kinds of routines defined in the programs, categorizing them as C functions, C++ functions, or C++ methods. From the table, we see that in many C++ programs, C++ functions do not account for a large percentage of the routines defined. Surprisingly, C functions account for quite a large percentage of routines defined in the C++ programs, most likely because C++ programs rely heavily on predefined C libraries.

Table 6 shows the mean and median number of static instructions for C functions, C++ functions and C++ methods. The numbers were calculated by taking the number of static instructions in each group (C functions, C++ functions and C++ methods) and dividing this by the number of static functions in each group in the program.

Program	C Functions		C++ Functions		C++ Methods		All Functions	
	Mean	Median	Mean	Median	Mean	Median	Mean	Median
BURG	88.3	49	—	—	—	—	88.3	49
DITROFF	66.6	38	—	—	—	—	66.6	38
TEX	122.1	49	—	—	—	—	122.1	49
XFIG	92.5	48	—	—	—	—	92.5	48
XTEX	88.9	49	—	—	—	—	88.9	49
COMPRESS	88.8	31	—	—	—	—	88.8	31
EQNTOTT	90.9	43	—	—	—	—	90.9	43
ESPRESSO	110.3	61	—	—	—	—	110.3	61
GCC	112.8	36	—	—	—	—	112.8	36
LI	60.4	31	—	—	—	—	60.4	31
SC	116.0	52	—	—	—	—	116.0	52
BISIM	35.4	15	75.1	35	74.8	18	63.4	18
CFRONT	79.3	21	282.8	109	313.6	73	229.7	65
DELTABLUE	69.1	24	150.0	42	35.5	25	63.4	25
DOC	72.6	24	82.3	34	60.4	34	65.3	32
GROFF	62.0	19	147.0	72	49.5	24	69.1	25
IDL	43.2	16	67.5	24	54.8	24	54.4	19
IDRAW	66.6	21	82.9	37	55.9	31	59.3	28
IXX	73.5	28	75.0	72	67.5	32	70.7	30
LIC	27.5	19	267.1	85	83.7	24	72.0	18
MMAKE	77.3	20	53.1	18	88.3	39	79.8	28
MPSDEMO	37.0	17	65.9	20	69.4	40	53.1	22
PORKY	26.0	20	278.0	109	49.6	5	58.5	18
DHRYSTONE	110.3	31	—	—	—	—	110.3	31
C Mean	94.3	44.3	—	—	—	—	94.3	44.3
C++ Mean	55.8	20.3	135.5	54.8	83.6	30.8	78.2	27.3

Table 6: Mean and Median Number of Instructions Per C Function, C++ Function, and C++ Method (Computed Statically).

The table shows that C++ programs on average have fewer instructions per function than the C programs (an average of 78.2 versus 94.3). When examining the median, the difference between the C++ and C programs is even more pronounced (27.3 versus 44.3), which implies that the C++ programs might be able to benefit more from procedure inlining than C programs. Also from the table, we see that methods in C++ programs tend to have fewer instructions than C++ procedures.

It is interesting to note in Table 6 for the C++ programs that the mean and median for the C++ functions (135.4 and 54.7) is a lot higher than the C++ methods mean and median (83.6 and 30.8) which is again higher than the C functions mean and median (55.2 and 20.2). A reason for the low mean and median for the C functions is that a lot of the C functions included in the C++ programs are small library routines. As we saw from the previous table, the majority of routines defined in C++ programs are either C functions or C++ methods, and as such, the size of these kinds of routines has the largest effect on the overall average size.

In combining Table 5 with Table 6 we can see which types of functions constitute the majority of a C++ program's code. In particular, we can multiply the frequency of occurrence of the different function types by the mean size of functions of those types. On average we see that 53% of the code consists of C++ methods, 29.1% consists of C library routines and the remaining 17.9% of the code consists of C++ functions.

4.2 Dynamic Function Size

In this section, we investigate the average number of instructions executed in each function (or method). We report a dynamic measure of function size, where the number of instructions executed for each function is counted each time it is called. These numbers were calculated by dividing the total number of instructions executed in each kind of function by the total number of calls to those kinds of functions (e.g., total instructions executed inside methods divided by total calls to methods). Function size is important because small (or more precisely, shorter-lived) functions have proportionally greater fixed function call overhead (i.e., saving registers, setting up arguments, etc.) and so will benefit more from optimizations like inlining. Also, as we will see, function size may have a significant effect on instruction cache performance.

Table 7 contains these data. The table shows the size of leaf functions versus non-leaves, methods versus non-methods, etc. The most significant result to notice in the table is that, dynamically, C++ functions and methods are much shorter than C functions (an average of 48.7 instructions versus 152.8 instructions). Note that the "All" column exactly matches the Insn/Call column in Table 3, as it should.

From the table, we also see that C++ methods are likely to be smaller than C++ non-methods (i.e., C++ functions not associated with a class and functions compiled by the C compiler). Likewise, C and C++ leaf functions are likely to be smaller than non-leaves. Note that the leaf value for COMPRESS is an outlier; when it is not included, the average for the C leaf functions is only 94.5. In C++, we also see that indirectly-called functions are approximately the same size as directly called functions.

Program	C	C++	Leaf	Non-Leaf	Meth	Non-Meth	Indirect	Direct	All
BURG	61.6	—	30.6	142.8	—	61.6	328.9	61.2	61.6
DITROFF	58.6	—	72.3	56.3	—	58.6	40.0	58.8	58.6
TEX	173.2	—	44.3	205.4	—	173.2	0.0	173.2	173.2
XFIG	61.9	—	38.6	74.8	—	61.9	75.7	61.0	61.9
XTEX	114.9	—	93.9	136.5	—	114.9	100.7	115.4	114.9
COMPRESS	368.4	—	1360.2	367.5	—	368.4	0.0	368.4	368.4
EQNTOTT	386.8	—	402.8	294.2	—	386.8	488.6	163.5	386.8
ESPRESSO	244.9	—	151.3	526.5	—	244.9	58.7	252.8	244.9
GCC	96.4	—	30.1	123.5	—	96.4	22.6	100.7	96.4
LI	42.5	—	31.9	44.2	—	42.5	22.0	43.1	42.5
SC	71.1	—	49.4	80.1	—	71.1	35.9	71.2	71.1
BISIM	22.3	30.0	11.3	35.8	34.8	21.6	299.8	27.2	29.1
CFRONT	49.5	110.9	36.9	127.1	73.4	110.2	23.0	94.9	94.9
DELTABLUE	40.3	23.6	19.2	28.5	23.9	34.1	15.3	48.5	26.2
DOC	51.3	43.6	23.5	53.8	43.0	52.9	48.4	41.9	45.2
GROFF	50.2	50.7	26.9	61.6	49.4	52.0	26.1	56.5	50.6
IDL	26.2	23.9	10.0	57.3	20.0	43.8	26.7	20.9	24.2
IDRAW	54.2	32.4	21.2	46.4	31.8	51.4	34.7	38.5	37.3
IXX	50.6	48.9	21.6	64.3	47.2	51.2	38.8	50.9	50.1
LIC	45.4	50.3	21.8	69.4	54.1	40.8	13.1	49.4	49.1
MMAKE	163.3	78.6	153.8	92.0	81.9	153.6	16.3	134.1	117.7
MPSDEMO	23.5	37.9	10.9	43.9	38.0	28.5	33.4	36.3	35.9
PORKY	53.4	21.3	8.2	52.8	15.2	74.3	10.0	26.6	24.4
DHRYSTONE	33.8	—	21.6	76.5	—	33.8	0.0	33.8	33.8
C Mean	152.8	—	209.6	186.5	—	152.8	106.6	133.6	152.8
C++ Mean	52.5	46.0	30.4	61.1	42.7	59.5	48.8	52.1	48.7

Table 7: Mean Number of Instructions per Invocation (Computed Dynamically)

Program	C	C++	Leaf	Non-Leaf	Meth	Non-Meth	Indirect	Direct	All
BURG	5.5	—	6.8	4.9	—	5.5	7.8	5.5	5.5
DITROFF	5.0	—	6.8	4.7	—	5.0	4.9	5.0	5.0
TEX	8.6	—	10.4	8.5	—	8.6	0.0	8.6	8.6
XFIG	5.2	—	4.8	5.3	—	5.2	6.5	5.1	5.2
XTEX	6.3	—	7.3	5.8	—	6.3	5.0	6.4	6.3
COMPRESS	5.7	—	18.4	5.7	—	5.7	0.0	5.7	5.7
EQNTOTT	8.5	—	9.1	5.4	—	8.5	9.2	5.6	8.5
ESPRESSO	5.1	—	5.0	5.1	—	5.1	5.3	5.1	5.1
GCC	5.6	—	5.2	5.7	—	5.6	7.2	5.6	5.6
LI	5.2	—	2.9	5.7	—	5.2	4.8	5.2	5.2
SC	4.0	—	3.5	4.2	—	4.0	7.2	4.0	4.0
BISIM	4.4	6.5	6.1	6.2	7.2	4.8	5.7	6.3	6.2
CFRONT	5.4	6.7	5.9	6.5	5.9	6.8	6.4	6.4	6.4
DELTABLUE	5.2	4.9	4.3	5.1	4.8	5.3	5.0	4.9	5.0
DOC	4.2	4.9	4.2	4.8	4.8	4.3	4.5	4.9	4.7
GROFF	4.4	5.4	4.6	5.2	5.6	4.6	4.5	5.1	5.1
IDL	4.9	4.5	3.3	5.3	4.0	6.3	4.1	5.7	4.5
IDRAW	4.4	6.0	3.8	6.0	5.9	4.8	5.9	5.2	5.4
IXX	4.9	5.9	4.9	5.2	5.8	5.0	6.6	5.1	5.2
LIC	4.5	5.9	4.6	5.8	6.0	4.8	3.8	5.5	5.5
MMAKE	5.8	7.7	6.2	6.6	7.9	5.8	11.6	6.3	6.4
MPSDEMO	4.3	6.0	5.3	5.8	6.0	5.0	4.7	6.1	5.8
PORKY	5.0	4.2	3.0	5.0	4.1	4.6	4.3	4.4	4.4
DHRYSTONE	6.5	—	6.6	6.4	—	6.5	0.0	6.5	6.5
C Mean	5.9	—	7.3	5.5	—	5.9	5.3	5.6	5.9
C++ Mean	4.8	5.7	4.7	5.6	5.7	5.2	5.6	5.5	5.4

Table 8: Mean Number of Instructions per Basic Block (Computed Dynamically)

One explanation for the smaller functions in C++ lies in the object-oriented approach of implementing program functionality in methods, where each method performs a relatively small and specific function for the objects in the class. Decomposing a large program using this approach appears to result in programs with invocations of many small functions.

In Table 4 (from the previous section) we also see that for the C++ programs, 25.5% of their procedure invocations are for C functions, 63.4% of their procedure invocations are C++ methods, and 11.1% ($100 - 25.5 - 63.4$) are for C++ functions. Taking into account the average dynamic function size for C functions, C++ functions and C++ methods, we see that the C++ programs will spend 56% of their time in C++ methods, 27.5% of their time in C library functions and the remaining 16.5% of their time in C++ functions.

4.3 Dynamic Basic Block Size

Table 8 shows the mean number of instructions executed per basic block in the test programs. The numbers were calculated by dividing the total number of instructions executed in the functions of the different types by the total number of basic blocks entered in those functions. By basic block, we mean any single-entry, single-exit sequence of

Program	% Insn that are Breaks	Break Type Categories (Percentage of Breaks)						Cond. Br. (% Tak.)	Insn Between Cond Branches	
		%CB	**%UB**	**%IC**	**%PC**	**%Ret**	**%Oth**		Mean	Std Dev
BURG	17.1	74.1	6.9	0.0	9.5	9.5	0.0	68.8	7.9	6.4
DITROFF	17.5	76.3	4.2	0.1	9.7	9.8	0.0	58.1	7.5	8.9
TEX	10.0	75.9	10.7	0.0	5.8	5.8	1.9	57.5	13.2	14.3
XFIG	17.5	73.6	7.7	0.6	8.6	9.2	0.3	54.8	7.8	8.7
XTEX	14.1	78.2	8.5	0.2	6.0	6.2	1.0	53.3	9.1	9.9
COMPRESS	13.9	88.5	7.6	0.0	2.0	2.0	0.0	68.3	8.1	7.2
EQNTOTT	11.5	93.5	2.1	1.5	0.7	2.2	0.0	90.3	9.3	6.0
ESPRESSO	17.1	93.2	1.9	0.1	2.3	2.4	0.1	61.9	6.3	4.9
GCC	16.0	78.9	7.4	0.4	6.1	6.5	0.8	59.4	7.9	8.5
LI	17.7	63.9	8.7	0.4	12.9	13.2	0.9	49.3	8.9	9.5
SC	22.3	83.5	3.9	0.0	6.3	6.3	0.0	64.3	5.4	6.1
BISIM	15.3	34.0	21.1	0.2	22.3	22.4	0.0	63.9	19.2	22.1
CFRONT	13.4	76.0	7.2	0.0	7.9	7.9	1.0	53.1	9.8	8.9
DELTABLUE	17.6	54.4	2.1	14.6	7.1	21.7	0.0	56.9	10.5	8.5
DOC	19.1	67.4	9.4	5.8	5.8	11.6	0.1	54.6	7.8	8.8
GROFF	17.5	66.2	10.4	2.2	9.1	11.3	0.8	49.2	8.6	9.4
IDL	19.6	50.0	7.5	12.0	9.1	21.1	0.3	46.7	10.2	8.4
IDRAW	17.0	61.4	7.1	4.8	10.9	15.7	0.0	59.0	9.6	14.5
IXX	17.0	66.0	10.4	0.8	10.9	11.7	0.1	56.3	8.9	10.7
LIC	16.5	66.9	8.4	0.1	12.3	12.3	0.0	54.0	9.1	10.2
MMAKE	14.4	84.6	3.6	0.8	5.1	5.9	0.0	83.4	8.2	8.0
MPSDEMO	14.9	56.9	5.8	3.1	15.6	18.7	0.0	60.8	11.8	12.9
PORKY	19.8	55.3	3.0	2.9	17.9	20.8	0.1	60.5	9.1	8.4
DHRYSTONE	13.8	50.0	6.0	0.0	21.4	21.4	1.2	28.6	14.5	15.6
C Mean	15.9	80.0	6.3	0.3	6.3	6.6	0.5	62.4	8.3	8.2
C++ Mean	16.8	61.6	8.0	3.9	11.2	15.1	0.2	58.2	10.2	10.9

Table 9: Percentage of Breaks for each Program and Categorization of Breaks by Break Type. The "% Insn that are Breaks" gives the percentage of all executed instructions that resulted in a break in control flow. The break types are: conditional branches (**CB**), unconditional branches (**UB**), indirect procedure calls (**IC**), procedure calls (**PC**), procedure returns (**Ret**), and other breaks (**Oth**, e.g., signals, switch statements, etc.)

straight-line instructions. Larger basic blocks offer more opportunity for architecture-specific optimizations, such as instruction scheduling.

In this case, we see that there is a small difference in the basic block size between C and C++ programs, with C programs having slightly larger basic blocks (5.9 instructions versus 5.5 instructions, respectively). The table indicates that basic blocks in methods may be slightly larger than those in non-methods (0.5 instructions difference or approximately 10% on average), and that basic blocks in C++ leaf functions and methods are slightly smaller than basic blocks in non-leaf C++ functions and methods. Again, the COMPRESS application skews the C leaf statistics—if it is not included the leaf and non-leaf means for C are similar. The C functions in C++ programs tend to have small basic blocks (4.8 instructions on average), giving them a size similar to the leaf C++ methods and functions.

4.4 Breaks in Control Flow

In this section we investigate the frequency of different kinds of breaks in program control flow. Results in this section are related to the previous results showing the size of both basic blocks and functions. In the previous table we showed that basic blocks are in both languages are from 5.5 to 5.9 instructions long. In this section, we show the dynamic percentage of instructions that are break instructions, which differs from the basic block size because some basic blocks end with a fall through to the next basic block.

Modern pipelined architectures rely on a predictable sequence of instructions; each type of break in control can be predicted using different mechanisms. Mispredicting the direction of a conditional branch or the destination of an indirect call or return can stall the processor for many cycles on modern architectures. Conditional branch prediction has been studied by a number of researchers; Lilja [31], McFarling and Hennessy [32] and Smith [33] present good surveys. Kaeli and Emma [34] showed that a hardware *return stack* effectively predicted the destination of procedure returns. In related work, we [13] have shown that indirect functions can be effectively predicted for C++ programs.

The columns of Table 9 decompose the breaks in program execution in the programs into six classes: conditional branches (**CB**), unconditional branches (**UB**), indirect procedure calls (**IC**), procedure calls not including indirect calls (**PC**), procedure returns (**Ret**), and "other" breaks (**Oth**). Table 9 describes the types of branches, or breaks in control flow, during the program execution. An "indirect branch" is a branch where the destination is not explicitly stored in the program body; the branch destination may be loaded from a table or computed from an expression. The column labeled "indirect calls" (**IC**) are indirect branches that result in a procedure call—these are used in C++ to provide "dynamic dispatch" to implement inheritance. The column marked "Other indirect branches" (**Oth**) includes all other indirect branches and jumps. These branches are used to implement `switch` statements, signals, exceptions and other control transfers. Also shown in this table is the "% of Taken" conditional branches.

Table 9 indicates what kind of break instructions are being executed. First, the table shows that C++ programs have a slightly higher percentage of break instructions than C programs (16.8% versus 15.9%) More importantly, however, the table shows that C++ programs execute many fewer conditional branch instructions (61.6% versus 80.0%) and more procedure calls, indirect procedure calls, and return instructions. As a result, breaks in control flow in C++ have very different characteristics from those in C. For example, because there are many more return instructions (approximately 2–3 times as many), these breaks are more predictable. On the other hand, because C++ programs have more indirect call breaks, these breaks require methods other than traditional conditional branch prediction mechanisms for accurate prediction.

The last two columns in Table 9 illustrate this difference in another way. There we show the mean and standard deviation of instructions executed between conditional branches in the programs, including the branch instruction itself. From these columns we see that on average, C++ programs tend to execute approximately 2 more instructions per conditional branch than corresponding C programs.

The results in Table 9 indicate that handling indirect calls, procedure calls, and returns properly will be important for C++ programs. Because indirect calls are quite frequent in C++, we and others are examining methods to reduce

Program	Quantile												Static Count
	10	20	30	40	50	60	70	80	90	95	99	100	
BURG	1	3	5	9	19	33	58	95	135	162	268	859	1,766
DITROFF	3	11	19	28	38	50	64	91	132	201	359	867	1,974
TEX	3	7	15	26	39	58	89	139	259	416	788	2,369	6,050
XFIG	8	31	74	138	230	356	538	814	1,441	2,060	3,352	7,476	25,224
XTEX	2	8	15	22	36	63	104	225	644	1,187	2,647	6,325	21,597
COMPRESS	1	2	2	3	4	5	6	8	12	14	16	230	1,124
EQNTOTT	1	1	1	2	2	2	2	3	14	42	72	466	1,536
ESPRESSO	4	10	19	30	44	63	88	121	163	221	470	1,737	4,568
GCC	13	38	77	143	245	405	641	991	1,612	2,309	3,724	7,639	16,294
LI	2	4	7	11	16	22	29	38	52	80	128	557	2,428
SC	2	3	4	6	9	16	30	47	76	135	353	1,465	4,478
BISIM	1	3	5	7	10	17	35	82	175	311	781	4,855	9,124
CFRONT	6	15	29	44	69	108	188	364	833	1,404	2,894	5,644	17,565
DELTABLUE	1	2	3	5	9	16	29	55	96	137	173	421	1,639
DOC	30	63	96	129	161	194	227	260	292	309	322	8,148	26,028
GROFF	7	21	42	75	118	165	226	313	457	622	1,095	2,944	8,038
IDL	12	24	35	47	58	70	81	93	104	110	114	1,182	3,839
IDRAW	8	16	23	31	39	46	54	62	69	73	77	6,500	31,333
IXX	2	4	6	8	10	11	13	15	17	18	18	1,099	6,697
LIC	17	33	49	65	82	98	114	130	146	154	161	2,311	18,897
MMAKE	1	1	2	2	2	4	15	43	134	233	520	1,082	3,036
MPSDEMO	3	8	19	33	50	78	127	197	366	571	981	1,487	4,954
PORKY	1	2	2	3	7	20	38	56	122	238	557	2,553	8,808
DHRYSTONE	2	3	5	8	13	17	21	25	29	31	33	458	1,796
C Mean	3	10	21	38	62	97	149	233	412	620	1,107	2,726	7,912
C++ Mean	7	16	25	37	51	68	95	139	234	348	641	3,185	11,663

Table 10: Quantiles of Conditional Branch Coverage. The table shows the number of static conditional branch sites, present in the program text, that account for the Quantile percentage of all the dynamically executed conditional branches for a given program. The table also indicates the total number of static conditional branches present.

this overhead [13, 35]. The C++ language was designed to be very efficient, introducing additional costs only when specific features (such as dynamic dispatch) are used. For example, careful design has produced a numerical library that can be used from C++ that is as efficient as a related FORTRAN library [11], and efficient operating systems have been written in C++ [12]. In many cases, C++ programmers eschew 'expensive' features, such as dynamic dispatch to achieve this efficiency. Optimizations that eliminate or reduce these costs will simplify software design in C++.

4.5 Branch and Indirect Call Quantiles

To better understand the behavior of programs, it is very helpful to know how many of the conditional branches statically present in a program are actually significant during the program's execution. To better characterize "significance" we have measured how many of the static conditional branches in the programs account for different quantiles of the dynamic execution of branches. Table 10 contains these results. To understand this table, consider a specific example, the CFRONT program, where the table indicates that there are 17,565 conditional branches statically present.

The table also shows that of these 17,565 static branch sites, only 69 branches account for 50% (the 50th quantile) of all dynamic conditional branch execution. To account for 95% of all conditional branch execution, 1,404 of the 17,565 branches are required. Finally, for CFRONT, the table indicates that of the 17,565 branches present, only 5,644 of these branches were ever executed given the input we used. Thus, the table indicates what kind of program coverage was provided by the inputs used. Not all branches will be executed during each program execution because many branches are only encountered during error conditions, or may reside in unreachable or unused code.

One important result indicated by this table is that a relatively small fraction of the conditional branches present in each program's text actually contributes significantly to the dynamic execution of the programs. For both C and C++ the mean of the 95th quantiles requires less than 10% of the total branches present and the 50th quantile requires less than 1% of all branches.

An implication of this result is that for many programs, branch prediction techniques need not predict all the programs branches effectively, but instead can be very effective if only the small fraction of important branches are predicted correctly.

We have computed a similar measure of the indirect call behavior in the programs, and these results are presented in Table 11. This table shows the number of indirect call sites (static locations where indirect calls are performed) that account for different quantiles of the total number of indirect calls the programs performed. Results similar to those in the previous table are apparent. What is also clear in Table 11 is that there are many more "significant" indirect calls in C++ programs than in C programs.

4.6 Call Stack Depth

We have measured the mean and standard deviation of the call stack depth of the programs, as indicated in Table 12. The mean call stack depth was calculated by sampling the call stack depth each time a call was performed. Based on the mean values, it would appear that there there is little variation in the mean stack depth between C and C++ programs. Looked more carefully at the data, however, we see that the mean values are strongly influenced by outliers. If we look instead at the the median call stack depth, we find that C programs tend to have a smaller call stack than C++ programs (9.9 versus 12.1).

To understand these results, it is important to look more carefully at the individual programs. Of the C programs, LI and BURG are outliers. Both of these programs make heavy use of recursion: LI is a Lisp interpreter solving the 8-queens problem; BURG uses recursion to perform bottom-up matching for a code generator. We also see that the C programs that are X applications (XTEX and XFIG) also have call stacks deeper than the median.

By comparison, the C++ programs with a deep call stack and large variance in depth, IDRAW and DOC, derive this behavior for a combination of reasons. Like XTEX and XFIG, they are X applications, written with the InterViews library. In that library, specific functionality is implemented using *functional composition*. For example, rather than pass a parameter indicating that a margin should appear around an object, the InterViews library provides a `Margin`

Program	Quantile												Static
	10	20	30	40	50	60	70	80	90	95	99	100	Count
BURG	1	1	1	2	2	2	3	3	3	4	4	3	51
DITROFF	1	1	1	1	2	2	2	2	3	3	3	3	37
TEX	1	1	1	1	1	1	1	1	1	1	1	0	125
XFIG	1	2	2	3	5	8	12	17	29	40	63	118	918
XTEX	1	3	5	7	9	12	16	22	35	45	66	95	916
COMPRESS	1	1	1	1	1	1	1	1	1	1	1	0	20
EQNTOTT	1	1	1	1	1	1	2	2	2	2	4	10	32
ESPRESSO	1	1	1	2	2	2	3	3	6	8	10	9	40
GCC	1	1	2	2	3	3	4	6	9	14	27	49	145
LI	1	1	1	1	1	1	1	1	2	2	2	2	23
SC	1	1	1	1	1	1	1	1	1	1	1	0	24
BISIM	1	1	1	1	2	2	3	3	4	4	5	30	388
CFRONT	1	1	1	1	3	4	5	6	7	8	8	8	197
DELTABLUE	1	1	1	1	2	2	2	2	2	2	4	15	62
DOC	1	1	1	1	4	11	19	33	55	112	378	2,111	6,200
GROFF	2	3	5	6	8	9	13	20	40	57	98	174	767
IDL	1	2	2	3	3	4	4	5	5	15	70	531	1,381
IDRAW	3	10	20	34	51	69	95	148	244	354	726	1,829	9,628
IXX	2	4	6	8	12	16	23	33	49	60	113	205	1,097
LIC	1	2	2	3	4	5	6	8	15	23	32	38	1,283
MMAKE	1	1	1	1	2	2	3	3	5	5	5	8	158
MPSDEMO	1	4	6	9	13	20	30	50	81	114	190	232	920
PORKY	1	2	3	4	8	14	23	39	71	104	200	407	1,431
DHRYSTONE	1	1	1	1	1	1	1	1	1	1	1	0	88
C Mean	1	1	1	2	2	3	4	5	8	11	16	26	211
C++ Mean	1	2	4	6	9	13	18	29	48	71	152	465	1,959

Table 11: Quantiles of Indirect Call Coverage. The table shows the number of static indirect call sites present in the program text that must be included to account for the Quantile percent of all executed indirect function calls. The table also shows the total number of static indirect call sites present.

Program	Call Stack Depth Mean	Call Stack Depth Std Dev
BURG	10.5	30.84
DITROFF	7.1	2.45
TEX	7.9	2.71
XFIG	11.6	4.47
XTEX	14.2	4.27
COMPRESS	4.0	0.07
EQNTOTT	6.5	1.39
ESPRESSO	11.5	4.67
GCC	9.9	2.44
LI	42.0	14.50
SC	6.8	1.41
BISIM	11.6	1.44
CFRONT	11.3	8.11
DELTABLUE	7.5	1.85
DOC	23.2	17.20
GROFF	10.0	2.95
IDL	12.1	2.42
IDRAW	18.1	8.57
IXX	13.0	3.28
LIC	13.5	2.96
MMAKE	10.8	3.63
MPSDEMO	8.2	2.51
PORKY	14.2	5.22
DHRYSTONE	3.5	0.69
C Mean	12.0	6.29
C++ Mean	12.8	5.01

Table 12: Mean and Standard Deviation of Call Stack Depth

object. The inner object is “wrapped” by the `Margin` object, incurring an additional procedure call. Both IDRAW and DOC use this programming style, and most such calls are indirect function calls. There are fewer opportunities for inline function expansion in C++ programs where indirect function calls are widely used.

Our results indicate that the application domain and the programming techniques used have a significant influence on call stack depth and variance. Further, we show that C programs often require a smaller call stack than C++ programs.

4.7 Memory Operations

We measured and compared the frequency of memory operations in C and C++ programs, and the results are provided in Table 13 (loads), Table 14 (loads) and Table 15 (stores). The results were computed by dividing the number of load and store instructions performed in each program by the total number of instructions executed. Modern computer architectures are sensitive to the number of memory operations; hardware mechanisms such as caches seek to mask some of these problems.

Table 13 shows the fraction of load instructions for the programs executing on the Alpha architecture. Initially, we measured the behavior a group of programs on the MIPS architecture, where we found a significant difference in the load behavior of C and C++ programs. Thus, we were initially surprised at the results in Table 13, which indicates that on the Alpha, there are only minor differences in the load behavior between C and C++ programs. A comparison of our results from the MIPS architecture and the Alpha architecture are given in Table 14; we could only compare the programs that executed on both the MIPS and Alpha processors.

Table 14 indicates that on the MIPS, the C programs measured executed approximately 4% more load instructions than the C++ programs. This difference is not as large on the Alpha architecture. In particular, the same C programs on the Alpha architecture execute more load instructions than they do on the MIPS, while the C++ programs execute more loads on the Alpha, but not as many more as the C programs. To explain this behavior, we must consider the structure of the Alpha architecture and procedure calling convention, which is a true 64-bit architecture. All pointers and addresses (including instruction addresses) occupy 64 bits. Furthermore, instructions occupy only 32 bits. In general, the target address of a call instruction on the Alpha cannot fit into the call instruction itself [1]. Instead, the address must be loaded from memory. Thus every procedure call on the Alpha in any language, including C, requires an indirect function call, causing a significant number of extra load instructions. The MIPS instruction set architecture, however, is only 32 bits, and the target address of a call instruction fits entirely in the instruction itself, preventing the need for an additional load. It is possible to transform most programs to eliminate this indirection, and this is performed by later versions of the DEC OSF/1 linker.

[1] Note that through link-time optimization, calls can be converted into local jumps if the target of the jump is known to be close enough. Our measurements indicate that this optimization is not heavily used in the version of the Alpha compiler and linker we measured.

Program	C	C++	Leaf	Non-Leaf	Meth	Non-Meth	Indirect	Direct	All
BURG	21.7	—	12.9	26.7	—	21.7	45.5	21.5	21.7
DITROFF	30.3	—	18.6	32.9	—	30.3	28.7	30.3	30.3
TEX	30.7	—	19.6	31.3	—	30.7	0.0	30.7	30.7
XFIG	23.5	—	15.6	25.8	—	23.5	26.4	23.3	23.5
XTEX	23.2	—	16.1	28.2	—	23.2	26.1	23.1	23.2
COMPRESS	26.4	—	0.1	26.5	—	26.4	0.0	26.4	26.4
EQNTOTT	12.8	—	11.8	20.2	—	12.8	11.6	20.3	12.8
ESPRESSO	21.6	—	20.1	22.9	—	21.6	25.9	21.5	21.6
GCC	23.9	—	16.7	24.6	—	23.9	27.2	23.8	23.9
LI	28.1	—	44.1	26.3	—	28.1	20.4	28.2	28.1
SC	21.2	—	15.3	22.8	—	21.2	41.6	21.2	21.2
BISIM	18.5	28.3	27.4	27.4	29.8	22.1	34.6	26.8	27.4
CFRONT	14.7	30.1	14.3	30.2	25.1	29.4	25.0	28.0	28.0
DELTABLUE	20.9	32.8	24.5	31.2	33.9	20.2	34.8	26.9	30.0
DOC	21.0	23.5	19.3	23.6	23.3	21.8	23.6	22.1	22.9
GROFF	19.7	26.9	15.9	26.3	27.5	21.3	25.5	24.5	24.6
IDL	12.1	27.7	21.5	26.7	28.2	19.0	27.7	21.1	25.2
IDRAW	20.9	25.8	18.6	25.6	26.5	20.6	27.0	23.0	24.2
IXX	22.8	24.6	16.3	24.5	23.6	23.2	24.2	23.3	23.3
LIC	17.0	27.3	18.3	26.8	27.7	19.5	25.0	25.2	25.2
MMAKE	16.8	29.0	15.6	27.8	29.1	16.9	15.2	21.3	21.1
MPSDEMO	19.5	27.5	19.6	27.3	27.4	23.7	27.6	26.6	26.7
PORKY	20.3	29.4	30.3	26.8	28.9	25.9	20.3	28.0	27.5
DHRYSTONE	20.4	—	14.2	26.5	—	20.4	0.0	20.4	20.4
C Mean	23.9	—	17.3	26.2	—	23.9	23.1	24.6	23.9
C++ Mean	18.7	27.7	20.1	27.0	27.6	22.0	25.9	24.7	25.5

Table 13: Percent of Load Instructions Dynamically Executed (Alpha Architecture).

Program	Load Percentage	
	MIPS	Alpha
COMPRESS	17.3	26.4
EQNTOTT	14.6	12.8
ESPRESSO	17.9	21.6
GCC	18.7	23.9
LI	21.8	28.1
SC	19.2	21.2
CFRONT	20.3	28.0
DOC	23.4	22.9
GROFF	21.6	24.6
IDRAW	22.1	24.2
C Mean	18.2	22.3
C++ Mean	21.8	24.9

Table 14: Comparison of Load Percentage on Alpha and MIPS architectures for selected programs. Note that because the DOC and IDRAW programs are interactive, the input datasets on the MIPS and Alpha architectures differed. In the rest of the programs, the same input datasets were used on both the MIPS and Alpha architectures.

Program	C	C++	Leaf	Non-Leaf	Meth	Non-Meth	Indirect	Direct	All
BURG	17.3	—	34.3	7.7	—	17.3	1.6	17.4	17.3
DITROFF	8.3	—	8.3	8.3	—	8.3	10.4	8.3	8.3
TEX	10.1	—	15.1	9.8	—	10.1	0.0	10.1	10.1
XFIG	10.9	—	8.0	11.7	—	10.9	12.7	10.7	10.9
XTEX	10.0	—	8.3	11.2	—	10.0	9.3	10.0	10.0
COMPRESS	9.5	—	83.5	9.2	—	9.5	0.0	9.5	9.5
EQNTOTT	1.3	—	0.0	11.4	—	1.3	0.0	9.8	1.3
ESPRESSO	5.1	—	6.5	3.9	—	5.1	0.0	5.1	5.1
GCC	11.7	—	9.6	12.0	—	11.7	10.3	11.8	11.7
LI	14.7	—	0.0	16.3	—	14.7	12.1	14.7	14.7
SC	9.1	—	1.2	11.1	—	9.1	13.9	9.1	9.1
BISIM	9.3	17.1	20.9	15.8	16.6	16.0	5.5	17.2	16.4
CFRONT	14.3	13.6	13.0	13.8	9.9	15.5	14.9	13.7	13.7
DELTABLUE	11.2	11.0	7.3	11.8	11.0	11.2	12.9	9.8	11.0
DOC	9.3	9.3	13.3	8.6	9.2	9.4	6.8	12.2	9.3
GROFF	8.1	12.8	5.7	12.4	13.4	8.9	11.6	11.3	11.3
IDL	2.6	6.3	1.9	7.3	5.9	5.4	5.7	5.7	5.7
IDRAW	11.6	13.2	8.0	13.9	13.1	12.0	12.6	12.7	12.7
IXX	12.2	11.6	5.6	13.0	11.1	12.3	8.5	12.2	12.0
LIC	12.6	13.7	6.2	15.2	14.0	12.4	10.0	13.5	13.5
MMAKE	8.8	18.9	8.0	17.7	19.1	8.9	22.1	12.2	12.4
MPSDEMO	8.0	12.2	8.2	12.1	12.4	8.7	11.5	11.8	11.8
PORKY	12.0	13.3	20.7	10.9	17.6	7.8	12.9	13.0	13.0
DHRYSTONE	10.7	—	9.6	11.8	—	10.7	0.0	10.7	10.7
C Mean	9.8	—	15.9	10.2	—	9.8	6.4	10.6	9.8
C++ Mean	10.0	12.7	9.9	12.7	12.8	10.7	11.2	12.1	11.9

Table 15: Percent of Store Instructions Dynamically Executed (Alpha Architecture).

Other trends are also apparent in Table 13. For example, leaf functions tend to have significantly fewer loads than non-leaf functions, at least partly because they do not perform any loads required in calling procedures. The C code executed in C++ programs tends to require fewer loads, probably because many of the C functions called are also leaf functions (e.g., library functions such as strcmp, etc). Methods appear to require more loads than non-methods and indirectly called functions tend to perform slightly more loads than directly called functions, at least in C++.

In Table 15, we see the fraction of all instructions that are store instructions in the programs. Here we see that the C++ programs almost uniformly perform more stores that the C programs. Overall, C++ programs appear to execute approximately 2% more store instructions than C programs. We also see there tend to be fewer stores in leaf than in non-leaf functions (again, COMPRESS leaf measurements represent an outlier) and more stores in methods than in non-methods.

Differences in store percentages cannot be attributed to the use of indirect function calls, which only require additional loads. Another significant contribution to the high rate of memory operations in C++ is probably due to register saves and restores across function calls. Recall that C++ programs execute many more calls and returns than C programs, resulting in additional register saves and restores.

Program	Number of Allocs	Number of Frees	Mean Alloc. Size (bytes)	Instructions per Object	Instructions per Byte
BURG	23098	2895	843.4	16918	20
DITROFF	0	0	—	—	—
TEX	60	32	1727.1	—	—
XFIG	7260	4070	193.6	4573	23
XTEX	2944	1131	358.9	8083	22
COMPRESS	1	0	16.0	—	—
EQNTOTT	85	0	23981.6	—	—
ESPRESSO	190386	190077	122.5	2694	22
GCC	1043	903	1353.4	137811	101
LI	27	0	3407.5	—	—
SC	6985	2419	52.0	131389	2524
BISIM	56931	12	97.6	46986	481
CFRONT	4492	2034	200.1	3679	18
DELTABLUE	120106	101178	39.2	719	18
DOC	81574	59181	206.5	5636	27
GROFF	24855	10867	57.2	2290	40
IDL	10219	446	47.7	2068	43
IDRAW	36349	22481	659.0	1602	2
IXX	22442	616	31.6	2303	72
LIC	5391	4551	47.4	791	16
MMAKE	1459	46	443.7	6500	14
MPSDEMO	170	137	185.3	5408	29
PORKY	110044	107256	32.6	1487	45
DHRYSTONE	2	0	56.0	—	—
C Mean	21080	18320	3205.6	50245	452
C++ Mean	39502	25733	170.6	6622	67

Table 16: Information on Allocation and Deallocation of memory. Mean Alloc. Size shows the mean size of the objects allocated in bytes. Instructions per Object and Instructions per Byte indicates the average number of instructions executed per object and per byte allocated in the program, respectively. Programs that allocated less than 100 objects are omitted from the Instr. per Object and Instr. per Byte columns and also from the means of these columns.

4.8 Dynamic Storage Allocation

In this section, we investigate the dynamic storage allocation performed by each of the test programs. Table 16 indicates the number of allocations and deallocations, the mean of the size of objects allocated, and the frequency of dynamic storage allocation in the test programs. In previous work [36], we showed that memory allocation is a time consuming operation that is easy to optimize, resulting in 5–15% performance improvements.

The table shows the absolute number of allocations (calls to `malloc`) and deallocations (calls to `free`) performed by each test program, as well as the average number of instructions executed per object and byte allocated; these metrics are useful when modeling memory allocation algorithms [37]. The table also shows the mean object size for each of the test programs. The mean object size is computed by measuring the total bytes allocated by the program and dividing by the number of allocations. Note that some of the programs may exhibit additional dynamic storage allocation behavior that is not indicated in this table. For example, GCC uses an internally defined allocation mechanism called obstacks to allocate and free dynamic memory. Similarly, LI allocates large chunks of memory and manages these chunks with an internal garbage collection algorithm. In these measurements, calls to realloc are treated as calls to free followed immediately by calls to malloc.

In the table, we see striking differences between C and C++ programs. First, it is very clear that many C programs allocate very few objects. Of the eleven C programs, only six allocated more than 100 total objects. Even in the remaining C programs that did allocate some dynamic storage, the frequency of storage allocation is far less than that of the C++ programs. Looking more closely, we see that some of the C programs allocate little or no dynamic storage (e.g., COMPRESS, DITROFF), while others allocate a small number of large objects and nothing else (e.g., TEX, LI). All the C++ programs, on the other hand, allocate a significant number of dynamic objects and furthermore allocate them at a high rate (on average, a byte of dynamic storage every 67 instructions).

While the frequency of allocation is clearly higher in C++, the relative size of allocated objects is apparently much smaller. From the table, we see that the C mean is also heavily influenced by programs such as LI, EQNTOTT, and TEX, that allocate a few very large objects and not many small objects. In the C++ programs, on the other hand, many objects are allocated and even if some large objects are allocated, these large objects usually do not dominate the mean object size as they often do in C programs. We say usually, because the IDRAW program has the highest average object size of the C++ programs. In IDRAW, a large number of 4000 byte objects are allocated, and these objects do raise the mean object size significantly.

An important conclusion to draw from this data is that efficient and correct dynamic storage allocation is quite important in C++ programs. Also, because correct deallocation of objects is often hard (e.g., note that many programs execute far fewer frees than allocations), C++ programs will benefit from forms of automatic storage management. For example, the InterViews library (used by DOC and IDRAW) provides automatic reference counting for all InterViews objects. Conservative garbage collection algorithms have also been shown to be effective for C and C++ programs [38, 39].

One interesting question that arises is why there is so much more heap allocation in C++ programs. There are undoubtedly many reasons, but we summarize some possibilities here[2]. Perhaps the most important reason is that object-oriented languages stress the creation of reusable components, which often return heap-allocated objects as a result. For example, the XTEX program displays a document by reading the page contents into a single large buffer and iterating through that page. By comparison, DOC allocataes a "glyph" for each type of character to be displayed, and additional structures, such as "hboxes" to contain words. While this kind of interface results in higher memory-usage, it reduces the complexity of the interface because the lifetimes of the objects created are not constrained. Trade-offs between flexibility and memory efficiency have changed dramatically since C was first being used (and many of the existing C libraries were written). Another important reason for differences may be historical. C programs were originally constrained to execute in a very small address space and stack allocation was very important in that

[2]We would like to thank David Ungar, Eliot Moss, John Ellis, and Mario Wolczko for their thoughts on this matter.

Program	4K	8K	16K	32K	64K	128K
BURG	3.40	2.00	0.66	0.04	0.03	0.00
DITROFF	5.55	3.18	1.78	0.58	0.11	0.00
TEX	4.84	2.85	2.17	1.00	0.34	0.17
XFIG	7.84	5.60	3.54	2.06	1.22	0.61
XTEX	3.46	2.19	0.85	0.58	0.38	0.20
COMPRESS	0.00	0.00	0.00	0.00	0.00	0.00
EQNTOTT	0.39	0.29	0.04	0.00	0.00	0.00
ESPRESSO	0.58	0.40	0.25	0.16	0.14	0.14
GCC	6.48	4.48	2.83	1.71	0.97	0.67
LI	3.83	3.33	0.49	0.06	0.00	0.00
SC	2.01	1.18	0.41	0.34	0.09	0.02
BISIM	3.69	1.12	0.62	0.41	0.25	0.16
CFRONT	9.41	7.24	4.82	2.63	1.48	0.80
DELTABLUE	2.42	1.57	1.11	0.42	0.00	0.00
DOC	3.98	2.60	1.59	0.88	0.57	0.20
GROFF	8.22	5.33	3.09	1.68	0.81	0.26
IDL	2.80	2.17	0.84	0.67	0.47	0.16
IDRAW	8.67	6.76	4.63	2.86	1.34	0.84
IXX	8.91	5.82	3.31	1.04	0.36	0.01
LIC	5.47	4.10	3.02	1.73	0.74	0.67
MMAKE	2.94	2.09	0.73	0.33	0.13	0.06
MPSDEMO	8.73	6.51	4.43	3.17	2.13	1.12
PORKY	4.75	2.51	1.43	0.66	0.45	0.19
DHRYSTONE	1.32	0.00	0.00	0.00	0.00	0.00
C Mean	3.49	2.32	1.18	0.59	0.30	0.16
C++ Mean	5.83	3.98	2.47	1.37	0.73	0.37

Table 17: Miss rates (%) for direct-mapped instruction cache

environment. C++ is a newer language and machines no longer have the tight memory constraints that machines in the 1970's had. Furthermore, C++ programmers include C programmers, but also include Lisp, Smalltalk, and other programmers of languages in which heap-allocation is very common. Another important difference between C and C++ is the support the languages provide for heap-allocation. Whereas in C, you are provided with a simple library interface, C++ supports allocation and deallocation with both syntactic and semantic conveniences such as constructors, destructors, new, and delete.

4.9 Cache Performance

In this section we investigate the instruction and data cache miss rates of the different test programs. In all cases, we discuss the miss rates in direct-mapped caches ranging from 4 kilobytes to 128 kilobytes in size.

4.9.1 Instruction Cache

The instruction (I) cache miss rate is a measure of the locality of reference of instruction fetches in a program. Table 17 shows the instruction cache miss rates of the test programs in caches of different sizes. The tables show that the I-cache miss rate of C programs is usually lower than that of C++ programs for caches of all sizes. Based on the C and C++ means in the table, C++ programs require I-caches that are approximately two times larger than C programs to achieve similar miss rates. Because the text size of the executable is likely to affect the cache miss rate, one might surmise that the increase in I-cache miss rate is correlated with the size of the executable. Of the four C programs with more than 100,000 instructions (TEX, XTEX, XFIG, and GCC), three have miss rates substantially higher than the C average. Likewise, of the C++ programs with executables smaller than 100,000 instructions (CFRONT, DELTABLUE, MMAKE, and MPSDEMO), three of the four have miss rates below the C++ average. Furthermore, because the MPSDEMO program executed so few total instructions (only 919,000), it is likely that part of its high miss rate can be attributed to cold-start misses.

Another possible cause of the poor miss rates in C++ programs we measured is the increased use of functional composition to perform complex tasks; we have seen indications of this behavior in other data we have collected. Instead of executing large monolithic functions to perform a task, as is often the case in C programs, C++ programs tend to perform many calls to small functions. Thus, C++ programs benefit less from the spatial locality of larger cache blocks, and suffer more from function call overhead. Note that this occurs even for programs performing similar functions. For example, Table 17 shows that GROFF has a higher miss rate than DITROFF, even though Table 3 shows that GROFF executes a comparable number of instructions to accomplish the same task.

The smaller dynamic function size of C++ programs, shown in Table 4.2, is probably another cause of the instruction cache miss rates. Instruction cache sizes range from 4Kb to 2Mb. Within a particular function, it is unlikely that instruction references conflict for cache locations. By comparison, programs executing a small number of instructions in each function, such as C++, *may* suffer more from instruction cache conflicts; for example, two mutually recursive functions may be aligned to the same memory cache addresses and constantly displace each other from the cache. C programs execute more instructions per function invocation, meaning more work is done within a particular function. If function calls can increase cache misses, the behavior of C functions can lead to fewer conflicts.

Table 4.2 also shows that C++ can possibly benefit more than C programs from basic block reordering and procedure layout algorithms to eliminate cache conflicts. These algorithms [14, 40, 41] have been shown to efficiently reduce the number of instruction cache misses and improve branch prediction.

4.9.2 Data Cache

The data (D) cache miss rate is a measure of the locality of reference of a program's access to data in the stack, static data segment, and heap. Table 18 shows the data cache miss rates of the test programs in caches of different sizes. In the data cache miss rates, we see much less differentiation between the C and C++ programs. In caches of all sizes, the differences appear negligible. But, while the averages are quite similar, there are notable differences between the C and C++ programs. First, it is important to note that two C programs, COMPRESS and SC, are significant outliers that strongly influence the mean, especially in the cases of larger caches. If those programs are not included in the C mean,

Program	4K	8K	16K	32K	64K	128K
BURG	15.38	10.90	7.71	6.23	5.05	3.88
DITROFF	12.93	6.81	3.07	0.43	0.16	0.03
TEX	13.12	8.09	4.42	2.29	1.51	1.03
XFIG	14.41	9.87	6.22	3.69	2.36	1.37
XTEX	7.92	4.61	3.03	1.87	1.14	0.74
COMPRESS	18.18	15.22	14.02	9.98	8.73	7.11
EQNTOTT	7.64	5.35	4.82	3.94	3.49	2.82
ESPRESSO	8.45	6.37	3.46	1.90	0.92	0.42
GCC	11.80	7.58	4.48	2.26	1.24	0.70
LI	11.87	8.25	5.00	2.24	0.93	0.44
SC	22.24	16.80	14.53	12.57	10.43	8.19
BISIM	11.62	3.47	2.90	2.46	2.20	2.08
CFRONT	19.18	12.01	7.47	4.68	2.67	1.53
DELTABLUE	25.07	22.49	19.24	18.32	17.32	15.15
DOC	9.69	6.21	4.25	2.41	1.67	0.99
GROFF	14.83	9.38	5.80	2.45	1.32	0.73
IDL	14.17	10.91	7.54	4.31	2.77	1.37
IDRAW	18.59	13.37	9.68	6.70	4.58	2.82
IXX	11.91	6.90	3.23	1.92	0.72	0.56
LIC	9.63	5.91	3.74	2.47	2.03	1.97
MMAKE	4.48	3.04	2.05	1.35	0.83	0.40
MPSDEMO	12.16	7.43	4.13	2.31	1.33	1.06
PORKY	16.39	9.25	6.19	3.67	2.65	1.09
DHRYSTONE	13.76	12.70	0.00	0.00	0.00	0.00
C Mean	13.09	9.08	6.43	4.31	3.27	2.43
C++ Mean	13.98	9.20	6.35	4.42	3.34	2.48

Table 18: Miss rates (%) for direct-mapped data cache

the C mean miss rate in a 128-kilobyte cache drops to 1.27%. Likewise, the DELTABLUE C++ program is a significant outlier, especially in the larger caches; if it is not included in the average, the C++ Mean drops to 1.32%. From the table, we see that the data cache miss rates in both C and C++ programs can be considerable, even for relatively large caches. The miss rate in C++ programs tends to be slightly higher, but not substantially so. This result is somewhat surprising given the large number of heap-allocated data structures the in C++ programs that we saw in Table 16.

4.10 Comparison Between Compilers

To determine the impact of the compiler we used in this study, we compiled one program, GROFF, using both the DEC C++ compiler and version 2.6.0 of the GNU G++ compiler. In general, the comparison is most notable for the similarities between the programs resulting from the two compilers; most differences are explained by the different runtime systems and standard libraries provided with each compiler. Rather than present the information for both programs, we only comment on the notable differences.

First, the program compiled by G++ had slightly more methods (18.3% versus 15.8%); this is an artifact of the different runtime system and implementations of the standard libraries provided with each compiler. Furthermore, methods were executed more frequently (57% versus 55%). On average, the G++ compiler produced slightly more compact C++ functions; however, this is again due to the runtime system. The DEC C++ program had 575 C functions, while the G++ program had 340 C functions. Much of the functionality provided by the C functions was implemented in C++; most of these are small functions, and skew the mean values.

The G++ compiler generated slightly less efficient code; overall, the G++ program executed 2.5% more instructions and called 8% more functions. The G++ program generated significantly fewer conditional branches ($5,831$ versus $8,038$), although both programs executed roughly the same number of branches. Again, most of the differences arise from the different runtime libraries. All other metrics were only slightly different between the two programs. We also compared DITROFF using the DEC C compiler and the version 2.6.0 of the GNU C compiler. Again, there were few notable differences, other than the GNU C compiler was slightly less efficient.

4.11 Summary of Results

Table 19 summarizes the results presented in this section allowing a direct comparison between the dynamic behavior of the C, C++, and DHRYSTONE programs. An additional column ("OO" C++) reflects means based only only four of the C++ programs: DELTABLUE, IDL, DOC, and IDRAW. We selected this subset because these programs exhibited the greatest use of indirect function calls of the C++ programs measured. In particular, in these programs, more than 30% of all procedure calls were performed indirectly. We call them "OO" C++ programs because they exhibit much greater use of both dynamic dispatch and method use than the other C++ programs measured; both of these behaviors are characteristic of programs written in an object-oriented style.

Metric	C Mean	C++ Mean	"OO" C++ Mean	DHRYSTONE	Table
Percentage of Calls that are Indirect	8.3%	23.1%	51.3%	0%	4
Percentage of Calls that are to Methods	—	63.9%	77.4%	—	4
Mean Instructions per Function (static)	94.3	78.2	60.6	110.3	6
Mean Instructions per Function Call (dynamic)	152.8	48.7	33.2	33.8	7
Mean Instructions per Basic Block (dynamic)	5.9	5.4	4.9	6.5	8
Mean Instructions per Conditional Branch (dynamic)	8.3	10.2	9.5	14.5	9
Median Mean Call Stack Depth	9.9	12.1	18.1	3.5	12
Percent Instructions that are Loads (dynamic)	23.9%	25.5%	25.6%	20.4%	13
Percent Instructions that are Stores (dynamic)	9.8%	11.9%	9.7%	10.7%	15
Mean Allocation Size (bytes)	3206	171	238	56	16
Instructions / Byte Allocated	452	67	22	—	16
I-Cache Miss Rate (%) (16K, direct-mapped)	1.18	2.47	2.04	0.00	17
D-Cache Miss Rate (%) (128K, direct-mapped)	2.43	2.48	5.08	0.00	18

Table 19: Summary of Results in Other Tables Comparing the C Mean, C++ Mean and DHRYSTONE Values. The "OO" C++ column reflects means based only on the DELTABLUE, IDL, DOC, and IDRAW C++ programs. These programs exhibited the greatest use of indirect function calls ($>$ 30% of all calls were indirect in these programs). The Table Number indicates where in this paper more complete information about the metric can be found.

In considering the "OO" C++ results, it is important to keep in mind that the sample size (4 programs) is very small. Based on this sample, we see that in many cases the differences between C and C++ are exaggerated when only the "OO" C++ programs are considered. In particular, the trends toward more smaller functions (both statically and dynamically) are even more definite in the "OO" C++ programs. Basic blocks are also smaller in the "OO" C++ programs. "OO" C++ programs also allocated objects very frequently (a byte every 22 instructions on average), and the mean object size was quite large (238 bytes, heavily influenced by the large objects allocated in both IDRAW and DOC).

Interestingly, some of the C/C++ differences are not as significant, when only the "OO" C++ programs are considered. For example, the instructions per conditional branch is lower in the "OO' C++ programs than it is in the overall C++ mean. The percentage of store instructions is also closer to the percentage observed in C programs. Because these results are based on such a small set of programs, we view them as at best tantalizing. We hope to investigate a larger set of "OO" C++ programs (or that others will) in the near future.

Table 19 also provides a clear picture of how DHRYSTONE captures the behavior of C or C++ programs. We conclude that it fails to capture the behavior of the test programs in most of the metrics we have measured. In particular, in DHRYSTONE: the mean number of instructions per invocation is smaller than the C and C++ means; the mean number of instructions per basic block is larger than the C and C++ means; the mean number of instructions per conditional branch is much higher than C and C++; the mean call stack depth is far smaller than in the C or C++ programs; the variance in the call stack depth is far smaller than in the C or C++ programs; fewer loads and stores are executed than in the C or C++ programs; almost no dynamic data is allocated, which is close in behavior to some C programs but not all; and the data and instruction cache miss rates are much lower than in actual programs.

Based on our measurements, it appears that DHRYSTONE benchmark fails to capture many important aspects of program behavior such as fraction of memory operations, call stack depth, and function size.

5 Implications

Throughout the presentation of our measurements in the previous section, we have discussed some of the reasons behind the observed behavioral differences in C and C++ programs. In this section, we synthesize this material to illustrate the implications of these differences for compiler writers and architecture designers.

In §4.1, we noted that the average static function size for C++ programs was smaller than the static C function size. In §4.2, we noted that more instructions were executed in functions in the C programs we measured than in the C++ programs. Section §4.3 showed that the C++ programs we measured had basic blocks that were comparable in size to our C programs. In §4.4 we noted that C++ programs had fewer conditional branches, but had significantly more procedure calls, some of which were indirect. In §4.5, we noted that a small number of the total branch sites and indirect call sites in programs account for a large fraction of the total dynamically executed branches and calls in both C and C++ programs. In §3.1 and §4.9.1, we noted that C++ programs had larger load images. In short, C++ programs tend to have more, shorter procedures that are often reached via indirect function calls.

This combination of characteristics poses several challenges and opportunities for compiler writers and architects. First, procedure inlining may be more promising, but more difficult to accomplish, due to the larger number of indirect function calls and the difficultly of interprocedural data flow analysis in C++ [35]. Other studies [42, 43, 44] have shown that procedure inlining is problematic; it does not always improve program performance. However, one attribute of procedure inlining is that it removes procedure calling overhead; this overhead is obviously a larger percentage of small procedures. Thus, automated procedure inlining decisions may benefit C++ programs more than C programs. For the programs we studied, the static C++ procedure size was smaller than the static C procedure size. Therefore, inlining C++ procedures may result in less code expansion and there may be more promising candidates for inlining. The most relevant work in this area has been done for the Self language [45, 46]. In related work, we found considerable opportunity for similar optimizations for C++ programs [13]. In particular, we found that profile-directed multi-version procedure inlining may perform very well. Here, dynamic type checks and inlined procedures would expand the most frequently executed methods; some of these runtime checks may be eliminated by compile time type analysis [35]. Procedure inlining reduces calling overhead and should also reduce the number of load and store operations.

In §4.4 we see that C++ programs actually execute slightly more branches than C programs. The C++ programs execute significantly fewer conditional branches, but at the same time execute more procedure calls, indirect procedure calls, and returns. These results imply that different branch prediction architectures are needed for C and C++ programs in order to achieve a high prediction accuracy for the different languages. The implications of these differences are considered in other publications [13, 14, 15], and a detailed discussion is beyond the scope of this paper. As described in §4.4, direct procedure calls and returns are easy to predict, while indirect procedure calls and conditional branches are more difficult to predict. The C++ programs tend to execute fewer of the branches that are hard to predict, suggesting that the C++ programs are easier to predict than C programs. In part, this arises from the programming style; C programmers tend to pass myriad parameters and specialize behavior using those parameters. By comparison, object-oriented C++ programmers tend to use subroutine calls to handle common behavior, and those procedure calls are more predictable. As mentioned, other work [13] attempts to reduce the number of indirect function calls by conditionally inlining indirect function calls. However, the benefits of substituting conditional branches for indirect function calls is very dependent on the underlying architecture. Most current architectures have support for conditional branch prediction, but not for indirect function calls.

Despite the presence of fewer conditional branches, C++ programs may not benefit more than C programs from architectures offering instruction level parallelism [47]. These architectures schedule several instructions concurrently. In VLIW architectures, the compiler performs the scheduling [48, 47], while in superscalar architectures, the compiler and architecture cooperate to schedule parallel instructions [49]. To take advantage of VLIW or superscalar techniques in C++ programs, compilers must be able to schedule instructions, registers and other resources across basic blocks and even procedure calls. Since C++ programs have many small procedures more interprocedural analysis will be needed to be taken advantage of for wide-issue architectures. Considerable effort will be needed to extract instruction level parallelism from either language. In C, the problem can been addressed by trace scheduling, an intra-procedural technique, while in C++, it must also be addressed by inter-procedural control flow analysis and procedure inlining since there are many more procedure calls than in C programs.

In §4.8, we found that C++ programs are more likely than C programs to dynamically allocate many small objects on the heap. This behavior implies that improvements to memory allocation, such as customizing the memory allocator for the application [36], will be more effective for C++ programs. The negligible difference in data cache performance shown in §4.9.2 implies that specific C++ optimizations for data cache locality may not be necessary. By comparison, optimizations for instruction caches [40, 41, 50] and possibly virtual memory systems [51, 52, 53, 54] will be more important for C++ programs than for C programs.

Our data also indicates that link-time optimizations, such as those proposed by Wall [55] and others will become more important. Object-oriented languages, such as C++, allow programmers to extend the class hierarchy without affecting the functionality of previously compiled procedures. This means that a programmer could use class 'X' and compile several modules using the interface of class 'X'. Some time later, class 'Y' could be declared as a subclass of 'X'. The original programs can operate on instances of class 'X' or class 'Y'; yet, optimizations permitted by the use of class 'Y' will not be detected. In general, these optimizations can not be detected until program link time, when all code bodies associated with a program are assembled. Only then is the complete class hierarchy visible to the compiler, and only then can specific optimizations be considered. For example, all subclasses of class 'X' may inherit a particular method (say, "X::foo"). Even though there is only one method, "X::foo", that may be called, in the absence of information about the full class hierarchy, the compiler must encode calls to that method using dynamic dispatch. However, a direct function call (or inlined function expansion) would be possible, since any call to "foo" must call "X::foo". In related work [13], we found that 31% of all indirect function calls in similar C++ programs could be eliminated with this simple link-time optimization. Clearly, these optimizations do not address all programs or systems. Analyzing dynamically loaded programs or programs imported from shared libraries that may change from execution-to-execution, is problematic.

In summary, object-oriented languages, even those employing a modicum of object-oriented features such as C++, will force program optimizers to perform more optimizations after the total program has been made available, typically at link time. A greater amount of link-time analysis allows C++ programs, and possibly other statically typed object-oriented languages, to execute efficiently on architectures designed for "conventional" languages such as C and FORTRAN. In general, these optimizations also benefit programs written in traditional languages, such as C or FORTRAN; however, their incremental benefit is less apparent in such languages. By comparison, these optimizations will be essential for object-oriented languages. Not surprisingly, highly-optimizing compilers for object-oriented languages typically perform those optimizations when the full program is available [45, 46].

Ideally, these optimizations will reduce the propensity for programmers using this emerging technology to "micro-optimize" their existing applications. For example, prior to the development of efficient register scheduling algorithms, a number of computer languages, such as C, provided "hints" to the compiler to indicate what variables should be stored in registers [60]; often, programmer intuition is incorrect. Similarly, the current design of languages such as C++ has been influenced by the extant compiler infrastructure; the dearth of systems using effective link-time optimizers led the language designers to employ a series of "crutches" that complicate the language design and the software engineering process. For example, a robust linker and compiler system would have avoided the need to distinguish between "virtual" and non-virtual functions. Likewise, the ability to inline functions would avoid the need to define inline functions in header files.

6 Summary

In this paper we have examined the dynamic behavior of 12 large C++ programs and 11 large C programs, including the programs in the SPECint92 benchmark suite. We have made the data gathered for this paper publicly available via anonymous FTP in the directory `ftp.cs.colorado.edu:pub/cs/misc/C++Data`. It is our hope that others use this data to support additional related research.

The goal of our measurements is to characterize the behavior of C and C++ programs and understand the ways in which the behavior differs between the languages. An empirical characterization of the behavior of programs written in these languages is very useful for a number of reasons. First, hardware designers and compiler optimizer writers can use this information to guide the design of new hardware and software that makes these programs perform better. Second, by understanding the differences in the ways that C and C++ programs behave, we can better guess at how optimizations known to be effective in C will be effective in C++. Finally, programmers can make use of these measurements to get a better understanding of how complex, "object-oriented" C++ programs are structured. To our knowledge, no one has yet measured the frequency of indirect procedure call usage or method usage in large C++ programs. Our results indicate that in an "object-oriented" C++ program, up to 80% of all procedure calls are method calls and up to 67% of all procedures calls are made indirectly.

Our results are closely tied to the set of programs we chose to measure; we have attempted to choose a representative sample of both C and C++ programs. With the caveat that the results are dependent on the programs we measured, we have observed notable differences between the measured C and C++ programs. The complete results are contained in the many tables throughout the paper, a summary of which appears in §4.11. Here we mention the most striking differences in behavior observed:

- C++ programs often perform an order-of-magnitude more indirect procedure calls;
- C procedures execute three times more instructions that C++ procedures (including methods), on average;
- C and C++ programs have basic blocks of approximately the same size; C programs execute more conditional branches than C++ programs and C++ programs execute more procedure calls, indirect procedure calls, and returns than C programs;
- C++ programs issue more loads and stores than C programs,
- C++ programs allocate far more objects from the heap; and
- C++ programs have worse instruction cache locality;.

We measured the performance of the DHRYSTONE benchmark and found it to be very different in many ways than either the C or C++ programs.

We also considered the implications of our measurements and discussed how existing architectural features, such as register windows, or compiler optimizations, such as interprocedural analysis, are likely to work in C++. We noted that several new approaches to optimizations or architectures could be successful in improving C++ program performance, including profile-directed inlining, better interprocedural register allocation techniques (to minimize register save/restore traffic), increased support for indirect call prediction, and support for automatic dynamic storage allocation (or garbage collection).

We view the results presented in this paper as a first step toward empirically understanding the behavior of C and C++ programs. It is clear that further studies are needed, both to determine if our results hold for a larger sample of programs and also for a range of instruction set architectures and compilers. We plan to continue these studies and report our results as we gather them. The current generation of instrumentation tools, such as QPT and ATOM, make the collection and analysis of such behavioral information far easier than it has been in the past.

7 Acknowledgements

We owe a great debt to James Larus for supporting our work by creating and maintaining QPT; this study would not have been possible at this time without the tools he developed. We would likewise like to thank Amitabh Srivastata and Alan Eustace for allowing us to use ATOM and providing support for it. We would like to thank Alex Wolf for numerous discussions on software metrics. Similarly, we thank Mark Linton for assistance with the InterViews applications and comments concerning their history. We would also like to thank Hans Boehm, Kinson Ho, James Larus, Doug Lea, Urs Hölzle, and Barbara Ryder for their comments on drafts of this paper. We thank the developers of the Internet, who had the foresight to create a tool uniquely qualified to make significant amounts of shared software available for our measurement. Finally, we thank the anonymous referees for their numerous insightful comments.

Brad Calder was supported by an ARPA Fellowship in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland. This work was funded in part by NSF grant No. ASC-9217394, NSF grant No. CCR-9404669, ARPA contract ARMY DABT63-94-C-0029 and and Digital Equipment Corporation External Research Grant Number 1580.

References

[1] David R. Ditzel and H. R. McLellan. Register allocation for free: The C stack machine. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 48–56, Palo Alto, CA, March 1982.

[2] Andrew Tannenbaum. Implications of structured programming for machine architecture. *Communications of the ACM*, 21(3):237–246, March 1978.

[3] G. Alexander and D. Wortman. Static and dynamic characteristics of XPL programs. *IEEE Computer*, 8(11):41–46, November 1975.

[4] John Cocke and Peter Markstein. Measurement of program improvement algorithms. In *Information Processing 80*, pages 221–228. IFIP, North-Holland Publishers, 1980.

[5] J. Elshoff. A numerical profile of commerical PL/1 programs. *Software – Practice and Experience*, 6:505–525, 1976.

[6] Donald E. Knuth. An empirical study of FORTRAN programs. *Software, Practice and Experience*, 1:105–133, 1971.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, Inc., 1990.

[8] Manolis G. H. Katevenis. *Reduced Instruction Set Computer Architecture for VLSI*. ACM Doctoral Dissertation Award Series. MIT Press, 1985.

[9] Standard Performance Evaluation Corporation, Fairfax, VA. *SPEC CFP92 Technical Manual*, release v1.1 edition, 1992.

[10] Standard Performance Evaluation Corporation, Fairfax, VA. *SPEC CINT92 Technical Manual*, release v1.1 edition, 1992.

[11] Jack Dongarra, Roldan Pozo, and David Walker. LAPACK++: A design overview of object-oriented extensions for high performance linear algebra. In *Proceedings Supercomputing '93*, pages 162–171. IEEE Press, November 1993.

[12] R. H. Campbell, V. Russo, and G. M. Johnston. "The Design of a Multiprocessor Operating System". In *Proc. USENIX C++ Workshop*, pages 109–125, Santa Fe, New Mexico, November 1987.

[13] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *1994 ACM Symposium on Principles of Programming Languages*, pages 397–408, January 1994.

[14] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. In *6th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[15] Brad Calder and Dirk Grunwald. Fast and accurate instruction fetch and branch prediction. In *21st Annual International Symposium of Computer Architecture*, pages 2–11, April 1994.

[16] Reinhold P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, October 1984.

[17] R.F. Cmelik, S.I. Kong, D.R. Ditzel, and E.J. Kelly. An analysis of MIPS and SPARC instruction set utilization on the SPEC benchmarks. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 290–302, Santa Clara, CA, April 1991.

[18] C.B. Hall and K. O'Brien. Performance characteristics of architectural features of the IBM RISC System/6000. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 303–309, Santa Clara, CA, April 1991.

[19] Dionisios N. Pnevmatikatos and Mark D. Hill. Cache performance of the integer SPEC benchmarks on a RISC. *Computer Architecture News*, 18(2):53–69, June 1990.

[20] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith. Cache performance of the SPEC92 benchmark suite. *IEEE Micro*, 13(4):17–27, August 1993.

[21] Douglas Clark and Henry M. Levy. Measurements and analysis of instruction set use in the VAX-11/780. In *The Ninth Annual Symposium on Computer Architecture*, pages 9–17, Austin, TX, April 1982.

[22] Cheryl A. Wiecek. A case study of VAX-11 instruction set usage for compiler execution. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 177–184, Palo Alto, CA, 1982.

[23] Richard E. Sweet and Jr. James G. Sandman. Static analysis of the Mesa instruction set. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 158–166, Palo Alto, CA, March 1982.

[24] Gene McDaniel. An analysis of a Mesa instruction set using dynamic instruction frequencies. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 167–176, Palo Alto, CA, March 1982.

[25] Mark D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California at Berkeley, Berkeley, CA, November 1987. Also appears as tech report UCB/CSD 87/381.

[26] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, 22(2):8–22, February 1989.

[27] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, Orlando, FL, June 1994.

[28] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software—Practice and Experience*, 24(7), 1994.

[29] James R. Larus. Efficient program tracing. *IEEE Computer*, 26(5):52–61, May 1993.

[30] Mark D. Hill and Alan Jay Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, C-38(12):1612–1630, December 1989.

[31] David J. Lilja. Reducing the branch penalty in pipelined processors. *IEEE Computer*, 21(7):47–55, July 1988.

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

[33] J. E. Smith. A study of branch prediction strategies. In *8th Annual International Symposium of Computer Architecture*, pages 135–147. ACM, 1981.

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

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

[36] Dirk Grunwald and Benjamin Zorn. CUSTOMALLOC: Efficient synthesized memory allocators. *Software—Practice and Experience*, 23(8):851–869, August 1993.

[37] Benjamin Zorn and Dirk Grunwald. Evaluating models of memory allocation. *ACM Transactions on Modelling of Computer Systems*, 4(1):107–117, January 1994.

[38] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. *Software – Practice and Experience*, 24(6):527–542, June 1994.

[39] Benjamin Zorn. The measured cost of conservative garbage collection. *Software—Practice and Experience*, 23(7):733–756, July 1993.

[40] Wen mei W. Hwu and Pohua P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *16th Annual Annual International Symposium on Computer Architecture, SIGARCH Newsletter*, pages 242–251. ACM, ACM, 1989.

[41] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27. ACM, ACM, June 1990.

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

[43] K. D. Cooper, M. W. Hall, and L. Torczon. Unexpected side effects of inline substitution: a case study. *Letters on Programming Languages and Systems*, 1(1):22–32, March 1992.

[44] 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.

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

[46] Urs Hölzle, Craig Chambers, and David Unger. Optimizating dynamically-typed object-oriented languages with polymorphic inlined caches. In *ECCOP '91 Proc.*, pages 21–38. Springer-Verlag, July 1991.

[47] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview and perspective. *Journal of Supercomputing*, 7:9–50, 1993.

[48] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[49] M. S. Lam M. D. Smith, M. Horowitz. Efficient superscalar performance through boosting. In *Proceedings of the 5th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 248–259, Boston, Mass., October 1992. ACM.

[50] Scott McFarling. Program optimization for instruction caches. In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 183–191. ACM, 1988.

[51] W. A. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. On the performance enhancement of paging systems through program analysis and transformation. *IEEE Transactions on Computers*, C-30(5):341–356, May 1981.

[52] Domenico Ferrari. Improving locality by critical working sets. *Communications of the ACM*, 17(11):614–620, 1974.

[53] S. J. Hartley. Compile-time program restructing in multiprogrammed virtual memory systems. *IEEE Transactions on Software Engineering*, 14(11):1640–1644, 1988.

[54] D. Hatfield and J. Gerald. Program restructing for virtual memory. *IBM Systems Journal*, 10(3):168–192, 1971.

[55] David W. Wall. Register windows vs. register allocation. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 67–78, Atlanta, GA, June 1988.

[56] P.A. Steenkiste and J.L. Hennessy. A simple interprocedural register allocation algorithm and its effectiveness for Lisp. *ACM Transactions on Programming Languages and Systems*, 11(1):1–32, January 1989.

[57] David W. Wall and Michael L. Powell. The Mahler experience: Using an intermediate language as the machine description. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 100–104. ACM, October 1987.

[58] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimizations at link-time. *Journal of Programming Languages*, pages 1–18, March 1993. (Also available as DEC-WRL TR-92-6).

[59] Carl A. Waldspurger and William E. Weihl. Register relocation: Flexible contexts for multithreading. In *20th Annual Annual International Symposium on Computer Architecture, SIGARCH Newsletter*, pages 120–129. ACM, ACM Press, May 1993.

[60] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Software Series. Prentice-Hall. Inc., Englewood Cliffs, NJ, 1978.