Quantifying Behavioral Differences Between C and C++ Programs

Brad Calder, Dirk Grunwald,
and Benjamin Zorn

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

CU-CS-698-94 January 1994

University of Colorado at Boulder

Technical Report CU-CS-698-94
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

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

Quantifying Behavioral Differences Between C and C++ Programs

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 1994

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 determine 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 [13, 49]. Early studies of program behavior [2, 10, 15, 29] guided the architectural design, and the importance of measurement and simulation has permeated architectural design philosophy [22]. In particular, the Berkeley and Stanford RISC were guided by studies of C and FORTRAN programs [27]. 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 [45, 46].

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 [14] and even operating systems [6]. 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 C. Before conducting the research reported in this paper, our hypothesis was that C++ programs would exhibit behaviors measurably different than C programs; this paper confirms that

hypothesis. In this paper, we seek to quantify these differences and identify how existing hardware and software optimizations that support C will also support C++.

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 have measured the behavior of ten large C++ programs and ten large C programs, including the programs provided in the SPECint92 benchmark suite. We measured 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 subcategories, such as comparing behavior in methods versus non-methods.

Our results indicate that the C and C++ programs we measured behave significantly differently in almost every aspect of behavior. Furthermore, the differences indicate 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 can use our results to construct a new generation of systems that execute C++ programs significantly more efficiently. 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[29]. 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 programmings 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[53]. 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 [45, 46], used in recent years to compare the performance of new computer architectures, has been investigated both in terms of instruction set usage[9, 19] and cache locality [39] (both of these

studies used what is now called the SPECmark89 suite). We mention these measurements because our measurements include the SPECint92 programs, and 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[2, 15], 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 [8] 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 [54]. Similarly, Sweet reports on the static instruction set usage of the Mesa instruction set [48] and McDaniel reports the dynamic instruction set usage in Mesa [34]. 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" [22].

More recently, published measurements of this kind have concentrated on the SPEC benchmark programs (e.g., [9]). 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 Application Programs

We have measured ten large C++ programs, ten large C programs, and the DHRYSTONE2 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 FORTRAN programs in the SPECfp92 benchmark suite; other studies provide such comparisons [9, 24]. The function of these programs and the input datasets we used are described in Tables 1 and 2.

CFRONT	The AT&T C++ to C conversion program, version 2.0. The input used was the file `groff.c`, provided as part of the GNU `troff` implementation, after being preprocessed with `cpp`.
CONGRESS	Interpreter for a PROLOG-like language. The input was one of the examples distributed with CONGRESS for configuration management.
DOC	Interactive text formatter, based on the InterViews 3.1 library. The input involved interactively browsing a 10-page document; as such, none of the editing capabilities of the program were used.
DUMP	Dump scans a Persi/Reprise [41] representation of a C++ program, converting it into a textual form.
GROFF	Groff Version 1.7 — A version of the "ditroff" text formatter. The first input was a collection of manual pages and the second was a 10-page paper. The inputs used were the same given to DITROFF.
IDRAW	Interactive structured graphics editor, based on the InterViews 2.6 library. The input involved drawing and editing a simple figure.
MORPHER	Structured graphics "morphing" demonstration, based on the InterViews 3.1 library. The input was a morphed "running man" example distributed with the program.
RT	An advanced ray tracing program from Indiana University, featuring extensive use of C++ templates.
RTSH	Ray Tracing Shell – An interactive ray tracing environment, with a TCL/TK user interface and a C++ graphics library. The input was a small ray traced image distributed with the program.
IDL	Sample backend for the Interface Definition Language system distributed by the Object Management Group. Input was a sample IDL specification for the Fresco graphics library.

Table 1: General information about the C++ programs.

DITROFF	C version of the "ditroff" text formatter. The first input was a collection of manual pages and the second was a 10-page paper. The inputs used were the same given to GROFF.
XDVI	Interactive DVI file previewer (patchlevel 11). The input was an interactive session in which a 10-page document was viewed forwards and backwards.
XFIG	Another interactive structured graphics editor (version 2.1.1). The input involved drawing and editing a simple figure.
XTEX	Another interactive DVI previewer (version 2.18.5). The input was an interactive session in which a 10-page document (the same used for XDVI) was viewed forwards and backwards.
026.COMPRESS	A file compression program, version 4.0, that uses adaptive Lempel-Ziv coding. The test input required compressing a one million byte file.
023.EQNTOTT	A translator from a logic formula to a truth table, version 9. The input was the file int_pri_3.eqn.
008.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).
022.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=6.
072.SC	A spreadsheet program, version 6.1. The input involved cursor movement, data entries, file handling, and some computation.
085.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).
DHRYSTONE2	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 2: General information about the C programs.

Program	Size (KBytes)	Total Instr	Total Func Calls	Total I-Calls	Compiled with
CFRONT	518	11,495,850	156,139	7	ATT C++
CONGRESS	2184	152,658,254	5,250,042	342,265	GNU C++
DOC	2469	406,673,840	9,476,005	5,310,058	GNU C++
DUMP	956	107,690,479	2,378,279	199,576	GNU C++
GROFF-1	1184	46,415,030	935,541	205,479	GNU C++
GROFF-2	1184	63,637,380	1,393,762	304,940	GNU C++
IDRAW	4250	184,930,642	4,307,713	1,490,459	GNU C++
MORPHER	2432	52,131,590	1,077,765	425,071	GNU C++
RT	1595	192,734,302	5,333,576	730,647	ATT C++
RTSH	3422	822,050,947	16,506,024	5,547,702	GNU C++
DITROFF-1	1487	52,686,831	676,656	8,491	GNU CC
DITROFF-2	1487	110,076,123	1,471,485	77	GNU CC
XDVI	1009	17,401,146	74,770	1,421	GNU CC
XFIG	1549	22,600,322	136,711	6,181	GNU CC
XTEX	1236	21,898,393	102,930	3,939	GNU CC
026.COMPRESS	67	94,345,259	251,333	2	GNU CC
008.ESPRESSO	295	604,811,607	1,903,209	84,755	GNU CC
023.EQNTOTT	102	1,426,572,207	4,012,030	3,215,051	GNU CC
022.LI	181	279,475,999	6,542,081	180,133	GNU CC
072.SC	281	1,017,786,477	10,634,338	282,938	GNU CC
085.GCC	974	137,821,007	1,355,551	80,812	GNU CC
DHRYSTONE2	56	6,362,520	172,020	3	GNU CC
IDL	1416	151,418,823	5,763,166	2,991,272	DEC C++

Table 3: Information about Program Executions. Total I-Calls indicates the number of indirect function calls executed in the program. Programs compiled with GNU C++ and GNU CC were compiled with version 2.3.3 of the compiler. IDL was compiled with version 1.2 of the DEC C++ compiler. In all cases, optimization was enabled (-O). All measurements were conducted on a MIPS-based DECstation 5000/240 workstation.

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 about which other measurement studies had been conducted; 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 files provided to both programs were exactly the same. Other C and C++ programs are also paired in the same way— XFIG (C) and IDRAW (C++) are picture drawing programs, XDVI (C), XTEX (C), and DOC (C++) are document previewers (DOC was used in this way, although it also has editor capabilities), and 085.GCC (C) and CFRONT (C++) are translators. We also included programs written using an object-oriented style in 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.

Information about the program executions we measured is provided in Table 3. Note that in the cases of DITROFF and GROFF, information about two runs with different inputs is presented both here and throughout the paper. We include these results to give an indication of the sensitivity of our measurements to different input sets.

With the information presented in Table 3, some important characteristics of the test programs already become apparent. The most important observation is that CFRONT, while written in C++, makes only 7 indirect function calls. From this behavior we immediately conclude that no virtual methods are defined

in CFRONT, and thus the program makes no use of dynamic dispatch, an important characteristic of the object-oriented paradigm. This conclusion leads us to believe that CFRONT does not take advantage of many of the benefits of the object-oriented paradigm, and thus is closer in behavior to C programs than to C++ programs. 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. As a result, when presenting summaries of the C++ programs, we include two means (or medians): one that includes CFRONT and one that does not.

Of the other C++ programs, we should note that DOC, IDRAW, and MORPHER are all interactive X-windows applications implemented using the InterViews class library [32]. 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, programs such as DOC and MORPHER have considerable "software reuse" since each primitive object can be composed for a variety of functions.

Interestingly, the table also shows that 023.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 023.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. Measurements presented are of the dynamic execution of the programs and were collected using a modified version of QPT, a program operation trace generator [30]. The instrumented programs were executed on a DECstation 5000/240 workstation with 112 megabytes of memory.

With 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. With modifications, 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++ methods, and C functions are being invoked. A C++ function is a function compiled by the C++ compiler, while a C function is compiled by a conventional C compiler. C++ 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.

The cache performance measurements we report were obtained using a modified version of the Tycho all-associativity cache simulator [23, 24]. Our modification involved making the stand-alone program into a callable library so that it could be linked with the instrumented executable. Tycho allows us to measure the performance of caches with different associativities simultaneously; we simulated direct-mapped caches with 32-byte cache lines.

Program	C	C++	Term	N-Term	Meth	N-Meth	Indirect	Direct	All
CFRONT	20.9	79.1	47.5	52.5	44.4	55.6	0.0	100.0	100.0
CONGRESS	29.4	70.6	55.2	44.8	61.1	38.9	6.5	93.5	100.0
DOC	5.6	94.4	75.0	25.0	94.0	6.0	56.0	44.0	100.0
DUMP	28.0	72.0	53.8	46.3	46.3	53.7	8.4	91.6	100.0
GROFF-1	23.1	76.9	61.5	38.5	63.5	36.5	22.0	78.0	100.0
GROFF-2	15.7	84.3	61.2	38.8	68.6	31.4	23.3	76.7	100.0
IDRAW	13.2	86.8	57.6	42.4	80.6	19.4	34.6	65.4	100.0
MORPHER	16.6	83.4	63.1	36.9	80.4	19.6	39.4	60.6	100.0
RT	3.8	96.2	80.4	19.6	79.3	20.6	13.7	86.3	100.0
RTSH	31.3	68.7	63.3	36.7	68.7	31.3	33.6	66.4	100.0
DITROFF-1	100.0		50.1	49.9		100.0	1.2	98.8	100.0
DITROFF-2	100.0		59.5	40.5		100.0	0.0	100.0	100.0
XDVI	100.0		74.6	25.4		100.0	1.9	98.1	100.0
XFIG	100.0		66.3	33.6		100.0	4.5	95.5	100.0
XTEX	100.0		69.9	30.1		100.0	3.8	96.2	100.0
026.COMPRESS	100.0		99.9	0.1		100.0		100.0	100.0
008.ESPRESSO	100.0		83.6	16.4		100.0	4.5	95.5	100.0
023.EQNTOTT	100.0		91.6	8.4		100.0	80.1	19.9	100.0
022.LI	100.0		50.7	49.3		100.0	2.7	97.3	100.0
072.SC	100.0		46.6	53.4		100.0	2.3	97.7	100.0
085.GCC	100.0		61.8	38.2		100.0	6.0	94.0	100.0
DHRYSTONE2	100.0		76.6	23.4		100.0		100.0	100.0
IDL	100.0		75.6	24.4		100.0	51.9	48.1	100.0
C++ Mean	18.7	81.3	61.9	38.1	69.0	31.0	23.9	76.1	100.0
C Mean	100.0	0.0	70.0	30.0	0.0	100.0	10.6	89.4	100.0
C++ w/o cfront	18.4	81.6	63.7	36.3	72.0	28.0	26.9	73.1	100.0

Table 4: Percentage of Invocations

Because our measurements are of dynamic program execution and we do not perform a static analysis of the program, we are unable to determine which functions in a program are *leaf* functions (i.e., do not contain any function calls). Instead, we are only able to determine which functions do not make any function calls for a particular input dataset, and we call these functions *terminal* functions to distinguish them from leaf functions. Clearly the leaf functions are a subset of the terminal functions.

3.3 Explanation of the Data Presented

In the next section, results of measurements of the application 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 applications 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 and numbered. At the bottom of the tables, measurements of the DHRYSTONE2 benchmark, the mean of the C++ program measurements, and the mean of the C program measurements are presented in separate rows. A row at the bottom shows the C++ mean without including the CFRONT application. While rows appear for two separate inputs of GROFF and DITROFF, only the average of the two inputs is included in the final C and C++ means.

Each row of the table is divided into nine columns (after the first column, which is the program name). The first eight of these 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. C functions are often called by C++ functions because C++ programs often 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 C++ functions. From the table, we see that approximately 80% of all function invocations in C++ programs are made to C++ methods and functions, while the other 20% are made to C functions.

Columns 4–5 A comparison of behavior in terminal (Term) versus non-terminal (N-Term) functions. From the table, we see that in C++ programs, an average of 62% of calls are made to terminal functions, whereas in C, an average of 70% of calls are made to terminal functions.

Columns 6–7 A comparison of behavior in methods (Meth) versus non-methods (N-Meth). 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 69% of all function calls are made to methods. Furthermore, in some programs, such as DOC, this percentage is much higher (i.e., 94%). CFRONT also performs significantly fewer method calls (44%) than the average for C++ programs.

Columns 8–9 A comparison of behavior in functions invoked indirectly versus directly. For the most part, 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 24% of all function calls in C++ programs are indirect, whereas in C only 10.6% of all calls are indirect. Furthermore, we see that the C average is heavily influence by the 023.EQNTOTT benchmark, which we have already mentioned. We also see that CFRONT is an unusual C++ program, performing essentially no indirect function calls.

The DHRYSTONE2 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. Note that another C++ program, IDL, appears at the bottom of each table. We include IDL there because we are currently unable to distinguish method calls, C++ function calls and C function calls in IDL (due to its being compiled by the DEC C++ compiler). As a result, the C versus C++ columns and Method versus Non-Method columns are not correct for IDL. For this reason, we also do not include IDL in the C++ Mean row. Nevertheless, all other measures of execution in IDL are correct and we include it to provide data from an additional C++ program.

To provide more insight into the specific behavior of the C++ programs, we include Table 5, which presents the fraction of method calls (top line of three), C++ function calls (second line of three), and C function calls (third line of three) in each of the C++ applications. Note that the table indicates the percentage of calls from a particular type of function. For example, in CFRONT the table shows that of all calls from methods (versus non-methods), 65.8% of these calls were made to other C++ methods. The table once again illustrates that DOC, IDRAW, and MORPHER are the most "method-intensive" C++ programs, while CFRONT is the least.

4 Results

Throughout this section we have intentionally resisted the urge to only present the summary information (i.e., C and C++ means) because we believe it is important for the reader to be able to consider the behavior

Program	C	C++	Term	N-Term	Meth	N-Meth	Indirect	Direct	All
CFRONT	3.3	48.8		44.4	65.8	33.4		44.4	44.4
	0.5	38.3		34.7	22.9	40.7		34.7	34.7
	96.2	12.9		20.9	11.3	25.8	100.0	20.9	20.9
CONGRESS		69.2		61.1	66.8	43.0	87.1	58.5	61.1
		10.8		9.6	11.6	3.2	1.7	10.4	9.6
	100.0	20.0		29.4	21.6	53.8	11.3	31.2	29.4
DOC	0.8	96.4		94.0	96.5	44.6	98.2	83.2	94.0
		0.5		0.5	0.4	1.3	0.4	0.5	0.5
	99.2	3.1		5.6	3.1	54.2	1.4	16.3	5.6
DUMP	0.0	54.7		46.3	55.2	26.4	63.5	43.7	46.3
		30.5		25.8	35.5	4.1	18.0	26.9	25.8
	100.0	14.8		28.0	9.3	69.5	18.5	29.4	28.0
GROFF-1	0.1	70.4		63.5	72.4	51.9	81.7	58.9	63.5
		14.9		13.4	9.1	19.0	6.1	15.2	13.4
	99.9	14.7		23.1	18.5	29.1	12.2	25.9	23.1
GROFF-2	0.1	72.5		68.6	73.7	61.4	83.3	64.7	68.6
		16.6		15.7	9.9	23.8	8.7	17.6	15.7
	99.9	10.9		15.7	16.4	14.7	8.1	17.7	15.7
IDRAW	0.0	86.7		80.6	88.4	17.3	85.6	75.7	80.6
	4.3	6.3		6.2	4.7	18.7	6.5	5.9	6.2
	95.6	6.9		13.2	6.9	64.0	7.9	18.4	13.2
MORPHER	0.0	88.7		80.4	87.7	47.9	87.5	75.3	80.4
	0.1	3.3		3.0	3.4	1.2	6.1	0.8	3.0
	99.9	8.0		16.6	8.9	50.9	6.4	23.9	16.6
RT	96.5	75.3		79.3	74.7	94.7	91.6	78.0	79.3
	2.5	20.2		16.8	20.8	3.5	3.8	18.3	16.8
	1.1	4.5		3.8	4.5	1.8	4.6	3.8	3.8
RTSH		74.2		68.7	74.2		78.2	63.2	68.7
						0.0		0.0	
	100.0	25.8		31.3	25.8	100.0	21.8	36.8	31.3

Table 5: Percent of Method Calls, C++ Function Calls and C Function Calls

Program	C	C++	Term	N-Term	Meth	N-Meth	Indirect	Direct	All
CFRONT	72.3	74.0	47.5	97.3	66.2	79.5	37.0	73.6	73.6
CONGRESS	23.3	31.5	23.6	35.8	33.6	21.9	20.5	29.7	29.1
DOC	71.8	41.2	30.5	80.1	40.5	80.8	49.5	34.6	42.9
DUMP	99.5	24.2	43.0	48.0	25.1	62.7	30.7	46.6	45.3
GROFF-1	54.1	48.3	39.2	66.3	44.5	58.5	27.9	55.7	49.6
GROFF-2	40.9	44.6	34.5	59.1	39.9	53.0	27.5	49.0	44.0
IDRAW	67.0	39.3	36.2	52.1	33.4	82.7	30.1	49.7	42.9
MORPHER	115.4	35.0	41.9	59.5	33.3	110.0	36.0	56.4	48.4
RT	294.0	25.9	17.0	114.7	23.9	83.2	23.9	38.1	36.1
RTSH	26.5	60.4	40.8	65.3	60.4	26.5	25.7	62.0	49.8
DITROFF-1	74.1		61.7	86.6		74.1	49.6	74.4	74.1
DITROFF-2	74.6		53.6	105.4		74.6	46.8	74.6	74.6
XDVI	232.4		259.6	152.3		232.4	176.2	233.5	232.4
XFIG	165.2		183.3	129.4		165.2	59.8	170.2	165.2
XTEX	212.6		243.6	140.7		212.6	96.8	217.2	212.6
026.COMPRESS	375.3		102.4	245087.0		375.3	35.3	375.3	375.3
008.ESPRESSO	317.8		187.5	984.0		317.8	62.6	329.7	317.8
023.EQNTOTT	355.6		366.9	232.7		355.6	397.3	187.1	355.6
022.LI	42.0		41.4	42.7		42.0	17.5	42.7	42.0
072.SC	97.8		69.5	122.4		97.8	18.7	99.6	97.8
085.GCC	101.7		44.8	193.6		101.7	21.1	106.8	101.7
DHRYSTONE2	37.0		27.5	68.1		37.0	46.0	37.0	37.0
IDL	26.3	10.2	12.7	68.5		26.3	37.7	13.9	26.3
C++ Mean	90.8	42.0	35.3	68.4	39.8	67.0	31.2	49.2	46.1
C Mean	197.5	0.0	155.7	24718.1	0.0	197.5	93.4	183.7	197.5
C++ w/o cfront	93.1	38.0	33.7	64.8	36.5	65.5	30.5	46.2	42.7

Table 6: Mean Number of Instructions per Invocation

of individual programs as this information can be valuable as well. For example, we believe that the DOC program represents the C++ application written in the most "object-oriented" style, and so readers may be interested in drawing conclusions based on the behavior of that program alone.

4.1 Dynamic Function Size

In this section, we investigate the average size in instructions of the functions (or methods) called by each program. Note that we report a dynamic measure of function size, where the number of instructions executed for each function is counted each time it is called. Function size is important because small 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 probably has a significant effect on instruction cache performance.

Table 6 contains these data. The table shows the size of terminal functions versus non-terminals, 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 46.1 instructions versus 197.5 instructions). In fact, when the C functions in C++ programs are not counted (the C++ column of the C++ Mean), we see that the average size of C++ functions and methods is only 42.0 instructions, almost five times smaller than the average C function. Because it is always useful to consider only the C++ functions and methods and not count the C functions that are part of the C++ program, we will repeatedly refer the the C++ column of the C++ Mean row as the *exclusive C++ mean.*

From the table, we also see that C++ methods are likely to be smaller than C++ non-methods (39.8 versus 67.0 instructions). Likewise, C and C++ terminal functions are likely to be smaller than non-terminals.

Program	C	C++	Term	N-Term	Meth	N-Meth	Indirect	Direct	All
CFRONT	5.2	4.8	4.2	5.3	4.3	5.4	5.7	4.9	4.9
CONGRESS	4.2	5.9	4.7	6.0	5.9	4.3	5.8	5.3	5.3
DOC	4.0	6.3	5.1	7.3	6.2	4.4	6.5	5.2	6.0
DUMP	4.3	8.3	4.5	6.4	8.7	4.6	9.8	5.1	5.2
GROFF-1	4.1	5.7	4.7	5.8	5.8	4.5	5.4	5.2	5.2
GROFF-2	3.8	5.7	4.8	6.1	5.9	4.7	5.3	5.4	5.4
IDRAW	4.4	8.0	5.4	9.1	7.1	6.4	8.7	6.4	6.8
MORPHER	3.7	8.8	4.4	8.6	8.4	4.1	10.9	4.7	5.7
RT	6.7	9.7	6.6	10.2	10.0	7.2	10.4	8.3	8.5
RTSH	4.7	12.6	10.1	9.6	12.6	4.7	11.0	9.6	9.9
DITROFF-1	5.9		6.5	5.6		5.9	6.8	5.9	5.9
DITROFF-2	6.4		7.1	6.0		6.4	5.8	6.4	6.4
XDVI	3.6		3.4	5.1		3.6	6.0	3.6	3.6
XFIG	3.8		3.5	5.1		3.8	5.7	3.8	3.8
XTEX	3.9		3.6	5.2		3.9	5.3	3.8	3.9
026.COMPRESS	5.9		7.2	5.5		5.9	5.0	5.9	5.9
008.ESPRESSO	6.1		5.7	6.4		6.1	5.5	6.1	6.1
023.EQNTOTT	3.0		2.9	4.9		3.0	2.9	4.5	3.0
022.LI	5.8		5.6	6.0		5.8	6.5	5.8	5.8
072.SC	5.2		4.2	5.8		5.2	6.2	5.1	5.2
085.GCC	5.6		5.6	5.7		5.6	7.0	5.6	5.6
DHRYSTONE2	4.5		3.5	7.0		4.5	4.9	4.5	4.5
IDL	6.2	5.5	4.1	9.1		6.2	7.0	4.8	6.2
C++ Mean	4.6	7.8	5.5	7.6	7.7	5.1	8.2	6.1	6.4
C Mean	4.9	0.0	4.9	5.5	0.0	4.9	5.6	5.1	4.9
C++ w/o cfront	4.5	8.1	5.7	7.9	8.1	5.0	8.6	6.3	6.6

Table 7: Mean Number of Instructions per Basic Block

Some specific programs are also worthy of note. 026.COMPRESS performs the entire file compression within a single non-terminal function, and as such the average instructions per non-terminal is much higher than the other programs. Also, CFRONT displays appears more similar to C programs than C++ programs, with a average of 73.6 instructions per invocation. The explanation for the smaller functions in C++ lies in the object-oriented approach of implementing program functionality in class methods, where each method performs a relatively small and specific function for the class. Decomposing a large program using this approach appears to result in programs with invocations of many small functions.

4.2 Basic Block Size

Table 7 shows the dynamic mean of the number of instructions per basic block in the test programs. Larger basic blocks offer more opportunity for architecture-specific optimizations, such as instruction scheduling. In this case, we see that C++ programs tend to have significantly more instructions per basic block than C programs. The difference between the C Mean (4.9 instructions) and the exclusive C++ mean (7.8 instructions) is striking. Without CFRONT, the exclusive C++ mean is even higher (8.1 instructions). The table also shows that basic blocks are more likely to be larger in methods (7.7 instructions) than in non-methods (5.1 instructions). The table shows that functions called indirectly are likely to have more instructions per basic than functions called directly (8.2 versus 6.1 instructions).

We believe the main reason for all of these observed behaviors is that dispatched methods often replace conditionals in object-oriented programs. Instead of testing conditionals to execute a sequence of instructions, as is often done in procedural programs, object-oriented programs perform dynamic dispatch to a method based on an object's type. Furthermore, once a method is entered, the execution context is

Program	% of Instructions which are breaks	Percentage of Breaks					Cond Branches	
		%CB	**%IJ**	**%UJ**	**%PC**	**%Ret**	**%FT**	**%T**
CFRONT	19.12	69.80	0.00	14.53	7.10	7.10	52.67	47.33
DOC	17.22	69.30	7.58	3.62	5.95	13.53	39.65	60.35
GROFF	18.81	65.13	2.81	9.73	9.26	12.08	47.17	52.83
IDL	18.76	41.57	10.53	17.73	9.76	20.29	50.89	49.11
IDRAW	15.58	64.90	5.17	5.02	9.78	14.95	42.93	57.07
MORPHER	17.66	71.84	4.62	4.66	7.09	11.71	43.77	56.23
RTSH	10.72	59.41	6.29	3.13	12.43	18.73	40.04	59.96
DITROFF	16.68	71.01	0.10	12.92	7.93	8.02	26.59	73.41
XDVI	25.67	89.41	0.03	7.16	1.64	1.68	39.92	60.08
XFIG	24.35	88.05	0.11	6.88	2.37	2.49	40.10	59.90
XTEX	24.26	89.55	0.07	6.49	1.86	1.94	37.80	62.20
026.COMPRESS	15.57	79.92	0.00	16.66	1.71	1.71	50.73	49.27
008.ESPRESSO	14.87	94.04	0.09	1.73	2.02	2.12	39.74	60.26
023.EQNTOTT	24.53	95.59	1.02	2.10	0.14	1.16	34.36	65.64
022.LI	17.57	65.32	0.39	6.86	13.15	13.35	53.60	46.40
072.SC	18.21	83.53	0.16	3.33	5.59	5.75	42.92	57.08
085.GCC	16.44	78.89	0.36	6.77	5.62	5.98	41.41	58.59
C++ Mean	16.84	63.13	5.29	8.34	8.77	14.05	45.30	54.70
C Mean	19.82	83.53	0.23	7.09	4.20	4.42	40.72	59.28

Table 8: Percentage of breaks for each program. Note that not all the C++ programs are shown. The other programs, DUMP, RT, and IDL have not been measured at this time.

completely determined, and, as a result, many methods contain straight-line code resulting in larger basic blocks.

4.3 Breaks in Program Execution

In this section we investigate the frequency of different kinds of breaks in program control flow. These results are related to the previous results showing the size of both basic blocks and functions. In Table 8, the first column shows the percentage of branch instructions that cause a break in the control flow graph. The last five columns decompose the number of branches into five classes: conditional branches (**CB**), indirect jumps (**IJ**), unconditional branches (**UB**), procedure calls (**PC**) and procedure returns (**Ret**). The table also shows what percentage of conditional branches fall through (**FT**) and what percentage of conditional branches are taken (**T**). C programs tend to have more breaks in control flow (19.8%) than C++ programs (16.9%). Also, the table shows that C++ programs execute 23 times more indirect jumps (5.29%) than C programs (0.23%), probably due entirely to virtual function calls in C++. The actual ratio is probably higher due to the fact that the 023.EQNTOTT C program executes an inordinate number of indirect function calls from the sort routine.

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 branch or the destination of an indirect call or return can stall the processor for 5-10 instruction cycles in modern architectures. Conditional branch prediction has been studied by a number of researchers; Lilja [31], McFarling [36] and Smith [42] present good surveys. Kaeli and Emma [26] showed that a hardware *return stack* effectively predicted the destination of procedure returns. In related work, we [5] have shown that indirect functions can be effectively predicted for C++ programs.

In Table 9, we first examine conditional branches because they are the most common type of branch. Table 9 shows the average number of instructions between conditional branch instructions in the test programs. From the table, we see the same relation between C and C++ that we saw when comparing basic

Program	C	C++	Term	N-Term	Meth	N-Meth	Indirect	Direct	All
CFRONT	8.0	7.2	6.4	7.9	6.3	8.3	8.7	7.3	7.3
CONGRESS	6.5	9.1	7.5	9.1	9.1	6.8	10.5	8.2	8.3
DOC	5.1	9.0	7.4	9.8	9.0	5.6	8.7	7.9	8.4
DUMP	6.8	34.6	7.9	13.3	43.0	7.8	43.0	9.4	9.8
GROFF-1	5.7	8.6	6.7	8.6	8.6	6.6	8.4	7.5	7.6
GROFF-2	5.3	8.8	7.1	9.2	8.9	6.9	8.2	8.0	8.0
IDRAW	5.7	12.2	7.3	14.7	10.7	8.7	16.1	8.8	9.9
MORPHER	4.5	15.3	5.8	13.9	14.6	5.0	22.3	6.2	7.9
RT	10.1	18.0	12.0	16.5	21.5	10.6	21.3	14.0	14.5
RTSH	7.4	20.2	17.5	14.1	20.2	7.4	22.5	14.8	15.7
DITROFF-1	8.4		9.6	7.8		8.4	11.7	8.4	8.4
DITROFF-2	9.3		11.2	8.2		9.3	12.0	9.3	9.3
XDVI	4.4		4.1	6.9		4.4	7.6	4.3	4.4
XFIG	4.7		4.2	6.9		4.7	7.1	4.6	4.7
XTEX	4.6		4.3	6.8		4.6	6.3	4.6	4.6
026.COMPRESS	8.0		10.4	7.4		8.0	7.6	8.0	8.0
008.ESPRESSO	7.1		7.0	7.3		7.1	6.8	7.2	7.1
023.EQNTOTT	4.2		4.1	7.8		4.2	4.0	7.0	4.2
022.LI	8.6		8.1	9.3		8.6	12.2	8.6	8.6
072.SC	6.5		5.7	6.9		6.5	18.7	6.5	6.5
085.GCC	7.5		8.2	7.2		7.5	12.6	7.4	7.5
DHRYSTONE2	6.0		4.5	11.0		6.0	7.7	6.0	6.0
IDL	12.8	1443.0	11.1	14.0		12.8	13.4	11.2	12.8
C++ Mean	6.6	14.9	8.7	12.0	15.9	7.4	17.9	9.4	9.9
C Mean	6.4	0.0	6.6	7.4	0.0	6.4	9.5	6.7	6.4
C++ w/o cfront	6.4	15.9	9.0	12.6	17.1	7.3	19.1	9.6	10.3

Table 9: Mean number of instructions per Conditional Branch

block sizes. The exclusive C++ mean (14.9 instructions) is more than twice as large as the C mean (6.4), again indicating that C++ programs tend to have fewer conditional branch (i.e., if-then-else) statements in them. Other trends also follow those seen in the basic block size results. Specifically, we see that methods tend to have many fewer conditional branches than non-methods and functions called indirectly tend to have fewer conditional branches than functions called directly. In fact, virtual functions (i.e. C++ methods called indirectly) have an average of 17.9 instructions per basic block and an average of only 31.3 instructions per invocation, indicated the presence of less than 1 conditional branch per function on average.

In Table 10, we only examine conditional branches and indirect jumps. The table shows the average number of instructions between these two types of unpredictable breaks in program control flow. We did not consider returns in these two tables because returns can be very accurately predicted using an a return stack [26], and pose few problems for modern architectures.

Many C++ programs use dynamic dispatch ("virtual functions") rather than conditional logic ("if"). Architecturally, this substitutes indirect function calls for conditional branches. We expect that some of the difference in the number of instructions between conditional branches in C and C++ programs would be reduced if indirect function calls are considered as well as conditional branches; from the table we see that this is indeed the case. The average number of instructions per break is reduced in the exclusive C++ mean from 14.9 instructions to 13.4, whereas the C mean remains the same at 6.4 instructions.

These results are confirmed by reconsidering Table 8. About 83% of the breaks in the C programs arise from conditional branches, but these only comprise 63% of the breaks in the C++ programs. This implies that handling indirect jumps and returns properly is increasingly important, since C++ is increasing in popularity. Because indirect jumps are so frequent in C++, researchers are examining methods to reduce

Program	C	C++	Term	N-Term	Meth	N-Meth	Indirect	Direct	All
CFRONT	8.0	7.2	6.4	1.0	6.3	8.3	8.7	7.3	7.3
CONGRESS	6.5	8.8	7.5	1.0	8.9	6.8	9.5	8.1	8.1
DOC	5.1	8.0	7.4	1.0	7.9	5.5	7.6	7.5	7.6
DUMP	6.8	29.6	7.9	1.0	37.4	7.7	35.3	9.3	9.7
GROFF-1	5.7	8.1	6.7	1.0	8.1	6.5	7.5	7.3	7.3
GROFF-2	5.3	8.4	7.1	1.0	8.4	6.8	7.4	7.8	7.7
IDRAW	5.6	10.9	7.3	1.0	9.5	8.6	12.3	8.4	9.1
MORPHER	4.5	12.7	5.8	1.0	12.5	4.9	17.2	6.0	7.4
RT	9.2	17.7	12.0	1.0	21.1	9.9	20.0	13.3	13.7
RTSH	7.4	17.4	17.5	1.0	17.4	7.4	15.4	14.0	14.2
DITROFF-1	8.4		9.6	1.0		8.4	11.7	8.4	8.4
DITROFF-2	9.3		11.2	1.0		9.3	12.0	9.3	9.3
XDVI	4.4		4.1	1.0		4.4	7.6	4.3	4.4
XFIG	4.7		4.2	1.0		4.7	7.0	4.6	4.7
XTEX	4.6		4.3	1.0		4.6	6.2	4.6	4.6
026.COMPRESS	8.0		10.4	1.0		8.0	7.6	8.0	8.0
008.ESPRESSO	7.1		7.0	1.0		7.1	6.8	7.1	7.1
023.EQNTOTT	4.1		4.1	1.0		4.1	4.0	6.1	4.1
022.LI	8.6		8.1	1.0		8.6	12.2	8.6	8.6
072.SC	6.5		5.7	1.0		6.5	18.7	6.4	6.5
085.GCC	7.4		8.2	1.0		7.4	12.6	7.4	7.4
DHRYSTONE2	6.0		4.5	1.0		6.0	7.7	6.0	6.0
IDL	10.2	1443.0	11.1	1.0		10.2	10.0	10.8	10.2
C++ Mean	6.5	13.4	8.7	1.0	14.4	7.3	14.8	9.0	9.4
C Mean	6.4	0.0	6.6	1.0	0.0	6.4	9.5	6.6	6.4
C++ w/o cfront	6.3	14.2	9.0	1.0	15.4	7.2	15.6	9.3	9.7

Table 10: Mean number of instructions per break in program (including indirect function calls and conditional branches)

this overhead [5, 38]. The C++ language was designed to be very efficient, introducing additional costs only when specific features (such as dynamic dispatch) were 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 [14], and efficient operating systems have been written in C++ [6]. In many cases, C++ programers eschew 'expensive' features, such as dynamic dispatch to achieve this efficiency. We feel optimizatoins that eliminate or reduce these costs will simplify software design in C++.

4.4 Call Stack Depth

We have measured the mean, median, and variance of the call stack depth of the applications as well. Here, we present the median and interquartile range (i.e., 75th percentile minus 25th percentile). Measurements of the mean, not presented here, show the same results; we included the median because our variance metric uses the interquartile range, which is easier to relate to the median.

Table 11 shows the median call stack depth while Table 12 shows the interquartile range. Both of these tables illustrate another striking difference between C and C++ programs. From the table of medians, we see that the median call stack depth of the C++ programs averages 15.8 functions while that of C programs is only 10.0 functions. The DOC and DUMP programs illustrate even more extreme cases of this behavior, both being twice as deep as the mean of the C programs. Of course, the depth of the call stack is very application and input dependent, and in the case of 022.LI, a Lisp interpreter solving the N-queens problem, recursive invocations create a very deeply nested call stack in a C program as well. However, 022.LI appears

Program	C	C++	Term	N-Term	Meth	N-Meth	Indirect	Direct	All
CFRONT	10	9	10	9	11	9	7	10	10
CONGRESS	19	18	19	18	18	19	19	19	19
DOC	18	20	18	29	20	18	10	30	20
DUMP	21	24	24	23	23	24	22	24	24
GROFF-1	14	10	11	10	10	13	9	11	11
GROFF-2	12	10	10	10	10	11	9	10	10
IDRAW	18	17	18	17	17	18	17	18	17
MORPHER	16	20	20	19	20	17	21	17	19
RT	7	8	8	6	8	8	7	8	8
RTSH	16	15	16	14	15	15	14	17	15
DITROFF-1	7		8	7		7	5	7	7
DITROFF-2	7		7	6		7	5	7	7
XDVI	14		15	14		14	11	14	14
XFIG	12		13	12		12	11	12	12
XTEX	15		15	15		15	13	15	15
026.COMPRESS	5		5	5		5	7	5	5
008.ESPRESSO	11		11	11		11	16	11	11
023.EQNTOTT	7		7	6		7	7	7	7
022.LI	41207		41158	41272		41207	41407	41205	41207
072.SC	8		8	7		8	10	8	8
085.GCC	11		11	10		11	12	11	11
DHRYSTONE2	4		4	5		4	6	4	4
IDL	14	6	14	14		14	14	14	14
C++ Mean	15.3	15.7	15.9	16.1	15.8	15.6	14.0	17.1	15.8
C Mean	10.0		10.3	9.6		10.0	10.2	10.0	10.0
C++ w/o cfront	16.0	16.5	16.7	17.0	16.4	16.4	14.9	17.9	16.6

Table 11: Median Call Stack Depth. 022.LI is not included in the mean because it is an extreme outlier.

to be quite uncharacteristic of the other C applications. We conclude that C++ program call stacks are likely to be much deeper than C program call stacks. This behavior can again be explained by the use of smaller functions composed with each other to implement a complex task and fewer opportunities for inline function expansion.

The difference in the variance of the call stack depth between C and C++ programs is even more striking. From Table 12 we see that the C++ interquartile range is 9.9 functions whereas the C interquartile range is only 2.3 functions. As mentioned earlier, one motivation for the use of register window architectures was that C programs displayed a relatively small variance in call stack depth. Our results confirm this observation of C programs, but suggest that C++ programs will not benefit as much from such hardware.

4.5 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) and in Table 14 (stores). Modern computer architectures are sensitive to the number of memory operations; hardware mechanisms such as caches seek to mask some of these problems.

The tables show the fraction of load and store instructions in the test programs. Again, we see significant differences between the C and C++ programs in both tables. In particular, C++ performs a greater percentage of loads (24.1% for the exclusive C++ mean versus 18.2% for the C mean) and a greater percentage of stores (13.4% for the exclusive C++ mean versus 7.9% for the C mean). Adding these numbers together, we see that C++ code in C++ programs perform greater than 10% more memory operations than C programs performing equivalent tasks (37.5% versus 26.1%).

Program	C	C++	Term	N-Term	Meth	N-Meth	Indirect	Direct	All
CFRONT	5	4	5	4	8	3	7	5	5
CONGRESS	8	9	9	8	9	8	12	8	8
DOC	20	32	29	26	32	21	21	25	32
DUMP	21	11	13	18	16	15	19	15	16
GROFF-1	5	5	5	5	5	5	5	5	5
GROFF-2	5	5	4	5	4	5	5	4	4
IDRAW	9	8	8	9	8	9	7	9	8
MORPHER	6	11	10	12	11	7	10	10	10
RT	1	1	0	1	1	0	2	1	1
RTSH	5	5	4	4	5	5	3	4	5
DITROFF-1	3	0	3	3	0	3	9	3	3
DITROFF-2	3	0	3	1	0	3	0	3	3
XDVI	2	0	2	3	0	2	3	2	2
XFIG	3	0	3	3	0	3	3	3	3
XTEX	5	0	5	6	0	5	7	5	5
026.COMPRESS	0	0	0	1	0	0	2	0	0
008.ESPRESSO	4	0	4	4	0	4	8	4	4
023.EQNTOTT	0	0	0	0	0	0	0	1	0
022.LI	41605	0	41639	41634	0	41605	41609	41607	41605
072.SC	1	0	1	1	0	1	3	1	1
085.GCC	3	0	3	2	0	3	2	3	3
DHRYSTONE2	1	0	1	1	0	1	0	1	1
IDL	4	1	3	3	0	4	4	4	4
C++ Mean	8.9	9.6	9.2	9.7	10.5	8.1	9.6	9.1	9.9
C Mean	2.3	0.0	2.3	2.4	0.0	2.3	3.6	2.4	2.3
C++ w/o cfront	9.4	10.3	9.7	10.4	10.8	8.8	9.9	9.6	10.6

Table 12: Variance (75% Quantile – 25% Quantile) Call Stack Depth. 022.LI is not included in the mean because it is an extreme outlier.

We also see from the tables a higher fraction of memory operations in methods rather than non-methods, and in functions called indirectly rather than directly. We feel that these results may be explained in several ways. The most significant contribution to the high rate of memory operations is probably due to register saves and restores across function calls. We have already seen that C++ has a much deeper call stack with more variance than C and these characteristics are likely to result in many more register saves and restores. This reason would also explain why the increased percentages of loads and stores observed in C++ are roughly equal (i.e., 5.9% more loads and 4.5% more stores). Further additional loads may occur due to the use of virtual functions, because functions called indirectly require additional memory references to load the function address from the object's dispatch table.

4.6 Dynamic Storage Allocation

In this section, we investigate the dynamic storage allocation performed by each of the test programs. Table 15 records the number allocations and deallocations as well as the distribution of the size of objects allocated. In previous work [18], 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 number of instructions between allocations or deallocations; this metric is commonly used when modeling memory allocation algorithms [56]. The table also shows the mean and median object size for each of the test programs.

Program	C	C++	Term	N-Term	Meth	N-Meth	Indirect	Direct	All
CFRONT	20.4	20.3	19.6	20.7	22.5	18.9	19.1	20.3	20.3
CONGRESS	16.9	20.7	20.2	19.5	21.0	17.0	18.9	19.9	19.8
DOC	17.7	24.0	23.3	23.6	24.0	18.9	25.1	20.4	23.4
DUMP	21.3	22.4	19.9	23.6	21.9	21.7	23.9	21.6	21.7
GROFF-1	19.1	22.5	21.1	22.1	23.2	19.6	22.5	21.5	21.6
GROFF-2	18.2	22.7	21.6	22.4	23.6	19.4	22.1	22.1	22.1
IDRAW	17.9	23.3	18.7	25.4	23.9	19.3	29.2	19.9	22.1
MORPHER	16.5	26.4	19.8	25.7	25.5	18.8	29.2	19.7	22.5
RT	21.1	23.4	18.3	25.4	25.5	19.6	26.4	22.3	22.7
RTSH	11.0	33.8	30.8	29.1	33.8	11.0	30.2	29.9	30.0
DITROFF-1	18.8		18.8	18.7		18.8	16.4	18.8	18.8
DITROFF-2	19.1		20.3	18.2		19.1	15.1	19.1	19.1
XDVI	17.3		17.9	14.5		17.3	19.1	17.3	17.3
XFIG	18.4		17.9	19.5		18.4	21.3	18.3	18.4
XTEX	17.5		17.6	17.4		17.5	23.0	17.4	17.5
026.COMPRESS	17.3		19.3	16.6		17.3	17.0	17.3	17.3
008.ESPRESSO	17.9		16.4	19.5		17.9	24.5	17.9	17.9
023.EQNTOTT	14.6		14.6	15.5		14.6	14.3	17.6	14.6
022.LI	21.8		21.8	21.9		21.8	14.9	21.9	21.8
072.SC	19.2		17.6	20.1		19.2	18.5	19.2	19.2
085.GCC	18.7		22.3	17.4		18.7	23.1	18.7	18.7
DHRYSTONE2	24.5		26.8	21.5		24.5	15.9	24.5	24.5
IDL	28.9	9.8	26.1	30.4		28.9	28.5	29.9	28.9
C++ Mean	17.9	24.1	21.3	23.9	24.6	18.3	24.9	21.8	22.7
C Mean	18.2	0.0	18.5	18.1	0.0	18.2	19.1	18.5	18.2
C++ w/o cfront	17.6	24.6	21.6	24.3	24.9	18.2	25.7	21.9	23.0

Table 13: Percent of Loads

Program	C	C++	Term	N-Term	Meth	N-Meth	Indirect	Direct	All
CFRONT	14.1	11.3	13.6	11.2	13.1	11.1	20.6	11.9	11.9
CONGRESS	11.4	12.5	10.1	14.1	12.9	10.7	11.0	12.3	12.3
DOC	9.4	10.9	12.6	8.6	10.9	9.3	9.4	13.2	10.7
DUMP	12.2	15.1	10.9	15.8	15.6	12.6	13.6	13.3	13.3
GROFF-1	10.2	11.2	11.0	10.9	11.7	9.9	10.8	11.0	10.9
GROFF-2	8.4	11.8	11.4	11.2	12.5	9.4	10.5	11.4	11.3
IDRAW	16.1	13.4	11.8	16.0	11.8	17.6	15.4	13.5	14.0
MORPHER	8.0	13.4	9.9	12.9	13.2	8.9	15.1	9.7	11.3
RT	10.8	16.2	6.3	19.5	20.4	8.0	21.0	13.9	14.5
RTSH	9.0	15.8	13.7	15.7	15.8	9.0	20.1	13.5	14.7
DITROFF-1	8.8		8.5	8.9		8.8	13.6	8.7	8.8
DITROFF-2	9.2		9.3	9.2		9.2	13.5	9.2	9.2
XDVI	7.3		7.2	7.8		7.3	11.1	7.3	7.3
XFIG	7.8		7.3	9.1		7.8	8.2	7.8	7.8
XTEX	8.1		8.1	7.8		8.1	7.0	8.1	8.1
026.COMPRESS	8.0		13.9	5.7		8.0	20.8	8.0	8.0
008.ESPRESSO	4.2		5.5	2.9		4.2		4.2	4.2
023.EQNTOTT	0.8		0.6	3.8		0.8		7.6	0.8
022.LI	13.0		12.4	13.6		13.0	14.3	13.0	13.0
072.SC	9.0		4.9	11.0		9.0	16.4	8.9	9.0
085.GCC	11.9		16.6	10.2		11.9	11.1	11.9	11.9
DHRYSTONE2	11.2		7.8	15.5		11.2	21.0	11.2	11.2
IDL	10.6	9.8	8.1	12.0		10.6	10.4	11.2	10.6
C++ Mean	11.2	13.4	11.1	13.9	14.0	10.8	15.2	12.5	12.6
C Mean	7.9	0.0	8.6	8.1	0.0	7.9	10.2	8.6	7.9
C++ w/o cfront	10.8	13.6	10.8	14.2	14.1	10.7	14.5	12.6	12.7

Table 14: Percent of Stores

Program	Number of		Alloc Size		Insns/Allocs	Insns/Frees
	Allocs	Frees	Mean	Median		
CFRONT	4461.0	1860.0	65.8	16.0	2577.5	6180.6
CONGRESS	316375.0	295317.0	45.6	28.0	482.5	516.9
DOC	71633.0	49646.0	197.9	48.0	5677.2	8191.5
DUMP	37159.0	26580.0	86.7	20.0	2898.1	4051.6
GROFF-1	22396.0	8439.0	46.8	24.0	2072.5	5500.1
GROFF-2	36298.0	21817.0	45.7	24.0	2312.4	3847.3
IDRAW	46033.0	33695.0	68.7	20.0	4017.3	5488.4
MORPHER	15026.0	11121.0	48.7	28.0	3469.4	4687.7
RT	6058.0	3790.0	192.1	20.0	31814.8	50853.4
RTSH	1218519.0	1404348.0	15.9	10.0	674.6	585.4
DITROFF-1	2.0	2.0	4110.0	28.0	26343415.5	26343415.5
DITROFF-2	2.0	2.0	4110.0	28.0	55038061.5	55038061.5
XDVI	840.0	224.0	591.8	16.0	20715.7	77683.7
XFIG	3447.0	1278.0	87.9	16.0	6556.5	17684.1
XTEX	2625.0	901.0	292.9	16.0	8342.2	24304.5
026.COMPRESS	3.0	2.0	5464.0	8192.0	47172628.0	47172628.0
008.ESPRESSO	190252.0	190250.0	79.9	28.0	3179.0	3179.0
023.EQNTOTT	86.0	1.0	21115.0	40.0	16588048.9	1426572207.0
022.LI	30.0	2.0	2021.7	8.0	9315866.6	139737999.5
072.SC	6906.0	2421.0	39.5	40.0	142428.7	406283.6
085.GCC	1046.0	903.0	1347.8	40.0	131760.0	152625.7
DHRYSTONE2	4.0	2.0	4120.0	48.0	1590630.0	3181260.0
IDL	37702.0	1716.0	22.3	12.0	4016.2	88239.4
C++ Mean	193845.7	204609.4	85.3	23.8	5978.2	9469.9
C Mean	20523.7	19598.4	3515.1	842.4	11408026.4	165485533.4
C++ w/o cfront	217518.7	229953.1	87.7	24.8	6403.3	9881.1

Table 15: Information on Allocation and Deallocation of memory. Alloc Size shows the mean and median size of the objects allocated in bytes.

In the table we see that some programs, such as RTSH, deallocate memory more often than they allocate it. As surprising as this may seem, some programs act this way due to the semantics of the Unix free operation, which specifically allows NULL pointers to be passed to it. Some applications are not careful about what values are passed to free, and sometimes pass NULL pointers unintentionally. These unintentional frees are harmless from the point of view of correct execution, although they do waste machine cycles.

In the table, we again see striking differences between C and C++ programs. The C programs appear to allocate far fewer objects than the C++ programs; the average number being an order of magnitude smaller than that of the C++ programs. Again, we see two extreme outliers in the C programs, 026.COMPRESS and DITROFF, that allocate only a handful of dynamic objects. These programs skew the mean of the instructions per allocation measure; however, even if these programs are excluded it is clear that the average instructions between allocations of the remaining programs is still significantly higher than that of the C++ programs.

While the frequency of allocation is clearly higher in C++, the relative size of allocated objects is not as clear. The C++ mean object size is significantly smaller than the C mean, but the C mean is also heavily influenced by programs such as 026.COMPRESS, 022.LI, and 023.EQNTOTT that allocate a few very large objects and not many small objects. Likewise, the mean of the C medians are heavily influenced by a few programs. Based on the specific programs, and not the mean, we see that the median object size in C++ programs tends to be smaller than that of the C programs that also allocate many objects (e.g., 008.ESPRESSO).

These results indicate that C++ programs allocate many more objects on the heap and those objects are often small objects, in the range of 16 to 28 bytes in size. C programs often allocate very few heap objects and the sizes of those objects can be quite large (e.g., 8192-byte objects used for buffered I/O).

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 the extra frees in the RTSH program), C++ programs will benefit from forms of automatic storage management. For example, the Interviews library (used by DOC, IDRAW, and MORPHER) provides automatic reference counting for all Interviews objects. Conservative garbage collection algorithms have also been shown to be effective for C and C++ programs [12, 55].

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[1]. 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. While this kind of interface results in higher memory-usage, it reduces the complexity of the interface because the lifetime 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 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.

[1]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
CFRONT	9.15	6.22	4.10	2.04	0.99	0.59
CONGRESS	6.37	4.16	2.01	1.27	0.68	0.09
DOC	3.69	2.82	1.85	1.49	1.24	0.16
DUMP	8.01	6.31	3.84	2.06	0.96	0.53
GROFF-1	6.84	4.50	2.10	1.04	0.70	0.39
GROFF-2	7.58	4.81	2.95	1.49	0.86	0.16
IDRAW	5.60	3.95	2.19	1.13	0.65	0.40
MORPHER	4.14	2.88	1.89	1.02	0.45	0.22
RT	2.74	2.17	1.17	0.42	0.18	0.05
RTSH	4.92	3.93	3.16	1.24	0.81	0.02
DITROFF-1	8.27	3.08	1.55	0.62	0.12	0.00
DITROFF-2	7.79	3.79	2.20	0.92	0.25	0.00
XDVI	1.10	0.70	0.20	0.15	0.10	0.04
XFIG	1.93	1.33	0.79	0.48	0.30	0.16
XTEX	1.34	0.76	0.44	0.31	0.20	0.10
026.COMPRESS	0.00	0.00	0.00	0.00	0.00	0.00
008.ESPRESSO	0.45	0.21	0.06	0.02	0.01	0.00
023.EQNTOTT	0.24	0.00	0.00	0.00	0.00	0.00
022.LI	3.14	1.22	0.91	0.51	0.00	0.00
072.SC	1.54	1.00	0.33	0.11	0.01	0.00
085.GCC	5.77	3.58	2.19	1.09	0.65	0.32
DHRYSTONE2	0.12	0.01	0.01	0.01	0.01	0.01
IDL	9.32	2.55	0.47	0.32	0.13	0.04
C++ Avg	5.76	4.12	2.53	1.33	0.75	0.26
C Avg	2.35	1.22	0.68	0.34	0.15	0.06
C++ w/o cfront	5.34	3.86	2.33	1.24	0.72	0.22

Table 16: Miss rates (%) for direct mapped instruction cache

4.7 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. Because we used Tycho, an all-associativity cache simulator, we also measured caches with multi-way associativity. The results of caches with greater associativity mirror those in direct-mapped cache and we do not present those results.

4.7.1 Instruction Cache

The instruction (I) cache miss rate is a measure of the locality of reference of instruction fetches in a program. Table 16 shows the instruction cache miss rates of the test programs in caches of different sizes. The tables clearly show that the I-cache miss rate of C programs is usually significantly 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 four 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. In fact, of the C programs with large executables (1 megabyte or greater: includes DITROFF, XDVI, XTEX, and XFIG), only one (DITROFF) has an I-cache miss rate near the average C++ miss rate. Srivastava [43] showed that C++ programs tend to contain a considerable number of unreachable functions. This is caused by the language semantics and software development environment. Each subclass that redefines a method must provide the body of that method in the final program, unless it can be determined that the method will never be called. This information can only be determined during program linking, and link-time optimizations are not commonly implemented.

Program	4K	8K	16K	32K	64K	128K
CFRONT	10.81	6.77	4.08	2.37	1.54	1.01
CONGRESS	6.53	4.14	2.82	1.48	0.99	0.67
DOC	5.70	3.93	2.62	1.55	1.18	0.75
DUMP	4.29	2.43	1.41	0.94	0.71	0.55
GROFF-1	6.84	4.50	2.10	1.04	0.70	0.39
GROFF-2	8.82	6.35	2.52	1.50	0.71	0.39
IDRAW	6.63	4.39	2.73	1.96	1.04	0.43
MORPHER	5.17	2.94	1.89	1.22	0.87	0.38
RT	3.40	2.27	1.34	0.83	0.59	0.49
RTSH	5.85	3.70	1.81	1.23	0.97	0.13
DITROFF-1	5.12	2.88	1.25	0.08	0.01	0.01
DITROFF-2	3.89	3.18	2.38	0.07	0.00	0.00
XDVI	2.13	1.15	0.82	0.66	0.42	0.26
XFIG	3.76	2.33	1.40	0.99	0.48	0.28
XTEX	3.34	2.09	1.42	0.93	0.42	0.28
026.COMPRESS	17.58	15.88	14.44	12.82	10.66	7.68
008.ESPRESSO	6.71	4.92	2.28	1.16	0.40	0.11
023.EQNTOTT	6.06	5.19	4.66	4.17	3.62	2.81
022.LI	4.97	3.01	1.88	1.31	0.91	0.34
072.SC	10.74	8.78	7.68	6.45	5.63	5.20
085.GCC	8.19	5.28	3.19	2.09	1.29	0.54
DHRYSTONE2	1.78	0.89	0.01	0.01	0.01	0.01
IDL	5.32	4.25	3.07	2.05	1.01	0.39
C++ Avg	6.24	4.00	2.33	1.43	0.95	0.53
C Avg	6.80	5.17	3.96	3.07	2.38	1.75
C++ w/o cfront	5.67	3.65	2.12	1.31	0.88	0.47

Table 17: Miss rates (%) for directed mapped data cache

Initially, we felt the differences in miss rate might be attributed to sampling error. Programs that execute for short periods of time suffer a disproportionate number of *cold start* cache misses. However, examining programs that execute for similar durations dispels this hypothesis. For example, comparing Table 3 and Table 16 shows that C programs that execute for a comparable number of instructions as C++ programs have lower miss rates (e.g., compare XDVI, XFIG and DITROFF-1 with CFRONT, GROFF-1 and MORPHER).

A likely cause of the poor miss rates in C++ 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 applications performing similar functions. For example, Table 16 shows that GROFF has a larger miss rate than DITROFF, even though Table 3 shows that GROFF executes fewer instructions to accomplish the same task.

In fact, we feel the smaller function size of C++ programs, shown in Table 4.1, causes much of the instruction cache miss rates. Instruction caches sizes range from 4Kb to 2Mb. Within a particular function, it is unlikely that instruction references conflict for cache locations. By comparison, programs containing many small functions, 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. Because C programs have larger functions, more work is done within a particular function, leading to fewer conflicts.

4.7.2 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 17 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 small caches, the differences are negligible, while in large caches C programs appear to have a higher miss rate (but see below). 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, 026.COMPRESS and 072.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, the C mean miss rate in a 128-kilobyte cache drops to 0.58%, which is close to the C++ mean (0.53%). Probably the most important difference between C and C++ programs illustrated by the table is that there is a much higher variance in the data cache miss rate of the C applications (ranging from 2.13% to 17.58% in a 4-kilobyte cache) than there is in the C++ applications, where the miss rates range from 3.40% to 10.81% in the same size cache.

The applications based on the InterViews library commonly use reference counting to simplify the reuse of specific object instances. For example, the DOC application is a document editor. The editor representation of the word "foo" is two instances of information concerning specific letters ('f' and 'o') structured as a linked list of three elements (forming 'f'-'o'-'o'). This sharing reduces the amount of data needed, possibly reducing the data cache miss rates. While other programs can employ the same mechanism, the object oriented languages simplify such bookkeeping. Based on the lack of a significant difference between the C and C++ means, it seems that the increased use of heap-allocated data in C++ appears not to result in a significant increase in the data cache miss rate.

4.8 Comments on the Dhrystone 2 Benchmark

In all of the previous tables, we include the results of measuring the DHRYSTONE2 systems programming benchmark. In the previous discussion we failed to mention how the DHRYSTONE2 results relate to the C and C++ programs that we measured and we summarize the relation here. Our conclusion is that DHRYSTONE2 fails to capture the behavior of the test programs in most of the metrics we have measured. In particular, in DHRYSTONE2: the mean number of instructions per invocation is much smaller than the C or C++ means (37 versus 197.5 versus 46.1); the mean number of instructions per basic block is smaller than the C or C++ means (4.5 versus 4.9 versus 6.1); the mean number of instructions per conditional branch (and program breaks) is close to that of C (6.0 versus 6.4); the median call stack depth is far smaller than in C or C++ (4 versus 10.0 versus 15.8); the variance in the call stack depth is far smaller than in C or C++ (1 versus 2.3 versus 9.9); more loads and stores are executed (35.7%) than in the C programs (26.1%) (although the DHRYSTONE2 mean is close to the C++ mean of 37.5%); 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 DHRYSTONE2 captures the conditional statement behavior of C programs reasonably well, but fails to capture other very important aspects of program behavior such as fraction of memory operations, call stack depth, and function size.

5 Implications

Throughout the presentation of the our measurements in the previous section, we have discussed what we believe are the reasons behind certain behavioral differences in C and C++ programs. In section, we

hope to synthesize this material to illustrate the implications of these differences for compiler writers and architecture designers.

In §4.1, we noted that more instructions were executed in functions in the C programs we measured than in the C++ programs. Section 4.2 showed that C++ programs had larger basic blocks than C programs. In §4.3 we noted that C++ programs had fewer conditional branches, but had significantly more indirect jumps. In §3.1 and §4.7.1, we noted that C++ programs had larger load images. In short, C++ programs tend to have shorter procedures that are often reached via indirect function calls, resulting in deeper and more variable call stacks.

This combination of characteristics poses several challenges and opportunities for compilers and architects. First, procedure inlining will 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++ [38]. Other studies [11] 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 will benefit C++ programs more than C programs, because inlining C++ procedures will result in less code expansion and there are more promising candidates for inlining. The most relevant work in this area has been done for the Self language [7, 25]. In related work we found considerable opportunity for similar optimizations for C++ programs [5]; 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 [38]. Procedure inlining reduces calling overhead and should also reduce the number of load and store operations.

These observations also imply that traditional intraprocedural optimizations (often called "global optimizations") will be less effective for C++ programs than for C programs because C++ subroutines contain fewer instructions. Traditional optimizations performed within a procedure, such as global register allocation, constant propagation and the like will be slightly less effective. However, these must be balanced against the slightly larger basic blocks found in C++ programs (§4.2). Larger basic blocks imply that instruction scheduling will be simpler and that the importance of conditional branch prediction will decrease, because the cost of mispredicted branches can be amortized over a larger number of instructions. In addition, in related work [4], we found that C++ programs tended to have more predictable conditional branches; that is, traditional branch prediction mechanisms are more effective for a similar set of C++ programs. In part, this occurs because C++ programs tend to have fewer branches, reducing the demands on extant resources. The benefits of substituting fewer branches for more indirect function calls is very dependent on the underlying architecture; in fact, other work [5] attempts to reduce the number of indirect functions, substituting conditional branches since existing architectures provide more architectural support for conditional branches.

Despite the presence of branches that are easier to predict and slightly larger basic blocks, we feel it is unlikely that C++ programs will benefit more than C programs from architectures offering instruction level parallelism [40]. These architectures schedule several instructions concurrently. In VLIW architectures, the compiler performs the scheduling [17, 40], while in superscalar architectures, the compiler and architecture cooperate to schedule parallel instructions [33]. To take advantage of VLIW or superscalar techniques in C++ programs, compilers must be able to analyze the target of dynamic method dispatches. Likewise, architectures must be able to resolve the likely target of a method dispatch as early as possible. In either case, the compiler or architecture require predictable control flow that occurs less frequently in C++ programs than C programs.

In §4.6, we found that C++ programs are more likely than C programs to dynamically allocate many small objects on the heap. This implies that improvements to memory allocation, such as customizing the memory allocator for the application [18], will be more effective for C++ programs. The negligible difference in data cache performance shown in §4.7.2 implies that specific C++ optimizations for data cache locality are not necessary. By comparison, optimizations for instruction caches [35, 37] and possibly virtual memory systems [1, 3, 16, 20, 21] will be more important for C++ programs than for C programs.

One of the most notable observations from the programs we instrumented is that C++ programs have deeper call stacks, with more variation in the call depth stack depth, than C programs. Procedure activation and calling conventions are at the core of many architectural optimizations; for example, the Berkeley RISC architecture proposed using rotating register windows, in part because that project relied on compiler implementations that, although dated, were in wide use at the time. Later research [51, 47] indicated that register windows were less advantageous when more sophisticated compile or link-time analysis could be performed.

Initially, we felt that register windows would benefit C++ programs more than C programs, because the complexity of interprocedural analysis in the presence of indirect function calls would reduce the effectiveness of link-time optimizations [52, 44]. However, register window underflow and overflow are the bane of register window implementations. When there are no more windows to allocate, existing windows must be flushed; likewise, windows may need to be loaded from memory. Overflows and underflows occur more frequently as the variance of the call stack depth increases; the actual impact on a particular configuration depends on myriad options, such as the number of register windows, the spill policy and the fill policy. Space limitations preclude pursuing this subject in more detail, but this is a topic of future research. We feel that the 'erratic' behavior of C++ programs, coupled with the relatively short procedures we recorded indicates that *variable sized* register windows [50], offering considerable more 'windows' containing a variable number of registers, will be increasingly promising.

Furthermore, even in the presence of register windows, link-time optimizations, such as those proposed by Wall [51] 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 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 [5], we found that 31% of all indirect function calls in similar C++ programs could be eliminating with simple link-time optimization.

In short, 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. We feel that more 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 sur-

prisingly, highly-optimizing compilers for object-oriented languages typically perform those optimizations when the full program is available [7, 25].

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 [28]; often, programmer intuition is incorrect. Similarly, we believe that the current design of languages such as C++ has been heavily 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.

6 Summary

We collected and instrumented a number of programs in order to empirically quantify differences between C and C++ programs. We measured a number of parameters interesting to compiler writers and computer architects. 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.

With the caveat that the results we concluded may be dependent on the programs we measured, we feel there are notable differences between C++ and C programs: C++ programs execute 32% fewer conditional branches, C++ programs use approximately 23 times more indirect function calls, C++ functions are shorter but have larger basic blocks, C++ programs tend to perform more procedure calls, C++ programs issue 10% more loads and stores than C programs, C++ programs have worse instruction cache locality and C++ programs allocate approximately 10 times more objects on the heap, and those objects are smaller than in C programs.

We also measured the performance of the Dhrystone benchmark (version 2) and found it to be very different in many ways than either the C or C++ programs.

Based on our measurements, we expect the following optimization or architectural features to perform worse in C++ than they do in C: local optimization (i.e., non-interprocedural), fixed-sized register windows, interprocedural analysis (call chains are deeper and more indirect calls are made), and efficient implementations of dynamic storage allocation.

Based on our measurements, we believe the following new approaches to optimizations or architectures could be successful in improving C++ program performance: better interprocedural register allocation techniques (to minimize register save/restore traffic), increased support for for indirect call prediction, and support for automatic dynamic storage allocation (or garbage collection).

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 like to thank Alex Wolf for supplying the DUMP program and numerous discussions on software metrics. Similarly, we thank Mark Linton for assistance with the InterViews applications and comments concerning their history. Peter Shirley contributed the RT ray-tracing application. Lastly, we thank the developers of the Internet, who had the foresite to create a tool uniquely qualified to make significant amounts of shared software available for

our measurement. This work is partially supported by Digital Equipment Corporation External Research Grant Number 1580.

References

[1] 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.

[2] G. Alexander and D. Wortman. Static and dynamic characteristics of xpl programs. *IEEE Computer*, pages 41–46, November 1975.

[3] Jean Loup Baer and R. Caughey. Segmentation and optimization of programs from Cyclic Structure Analysis. In *Proc. AFIPS*, pages 23–36, 1972.

[4] Brad Calder and Dirk Grunwald. Fast and accurate instruction fetch and branch prediction. In *21st Annual International Symposium of Computer Architecture*, April 1994. (to appear).

[5] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *1994 ACM Symposium on Principles of Programming Languages*, January 1994. (to appear).

[6] R. Campbell, V. Russo, and G. Johnston. The Design of a Multiprocessor Operating System. In *Proc. USENIX C++ Workshop*, 1987.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. *Software—Practice and Experience*, 1994. To appear.

[13] 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.

[14] 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.

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

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

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

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

[19] 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.

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

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

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

[23] Mark D. Hill. *TYCHO*. University of Wisconsin, Madison, WI. Unix manual page.

[24] 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.

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

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

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

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

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

[30] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report 1083, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI, March 1992.

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

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

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

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

[39] 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.

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

[41] David S. Rosenblum and Alexander L. Wolf. Representing semantically analayzed c++ code with reprise. In *Proceedings of the 3rd C++ Conference*, pages 119–134. USENIX, April 1991.

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

[43] Amitabh Srivastava. Unreachable procedures in object-oriented programming. *Letters on Programming Languages and Systems*, 1(4), 1993. (to appear).

[44] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimizations at link-time. *Journal of Programming Languages*, March 1992. (Also available as DEC-WRL TR-92-6).

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

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

[47] 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.

[48] 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.

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

[50] 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.

[51] 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.

[52] 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.

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

[54] 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.

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

[56] Benjamin Zorn and Dirk Grunwald. Evaluating models of memory allocation. *ACM Transactions on Modelling of Computer Systems*, 1994. (to appear).