Execution Characteristics of Object Oriented Programs on the UltraSPARC-II

R. Radhakrishnan and L. John
Electrical and Computer Engineering Department
University of Texas at Austin
Austin, TX 78712
(512) 232-1455
radhakri@ece.utexas.edu
ljohn@ece.utexas.edu

Abstract

It is widely accepted that object-oriented design improves code reusability, facilitates code maintainability and enables higher levels of abstraction. Although software developers and the software engineering community have embraced object-oriented programming for these benefits, there have been wide concerns about the performance overhead associated with this programming paradigm on modern processors. In this paper, we characterize the performance of several C and C++ benchmarks on an UltraSPARC-II processor. Various architectural data related to execution behavior of the benchmarks are collected using on-chip performance monitoring counters. Factors including CPI, instruction and data cache misses, processor stalls due to instruction cache misses and branch misprediction, from real execution of several programs are measured and presented. While previous research evaluates the behavioral differences between C and C++ programs based on profiling and simulation, we will measure execution behavior. Results show that the programs in the C++ suite incur a higher CPI, higher i-cache misses, and higher branch mispredictions than the programs in the C suite. A strong correlation was observed between CPI and branch mispredictions for the C++ application programs.

Keywords: Machine Measurement, Object Oriented Programming, Performance Monitoring, Program Characterization, Superscalar Architecture.

** This work is supported in part by National Science Foundation under grant number CCR-9697098.*

1 Introduction

Software development methodology has evolved significantly over the last decade, primarily to cater to the complexity of modern software systems. It has been become widely accepted that object-oriented programming (OOP) improves code reusability, facilitates code maintainability and enables higher levels of abstraction. This has led to the popularity of the object oriented paradigm despite concerns that the run-time performance of object-oriented programs lag behind that of traditional programs. Although most modern commercial applications are developed in object oriented languages, the impact of object-oriented programs on processor performance has not been adequately characterized to allow the optimization of processors. It is possible that cache design, branch prediction, instruction scheduling techniques and several design aspects have to be reconsidered when migrating to modern workloads. Detailed workload characterization studies are required to precisely analyze the correlation of each aspect with the overall execution behavior. Such studies will lead to a through understanding of how processor design technology can be better matched to contemporary software development practices.

Object oriented programming introduces the notion of objects, classes, encapsulation, inheritance, polymorphism, templates, genericity, etc into programming in an attempt to solve some of the most difficult problems facing software development. An object encapsulates both data and operations on the data [3]. Encapsulation allows OOP to integrate control and data into an object thus hiding all the details of the object in the object itself [17]. Inheritance enables the creation of new classes by extending and adapting old classes and enables reuse. Polymorphism is creating functions that have many forms and postponing the function invocation and its binding to run time. Polymorphism provides virtual function interface which will allow late binding and provide full flexibility in program extensions. The compiler sets up a dispatch entry for the function in the class virtual function table or *vtable* and *dynamic dispatch* is employed. The concepts of generics and templates allows the generation of sophisticated programs from simple specifications. Genericity allows one to build collections of items, where the type of items is known and the items can be retrieved from the collection as that type, without type casting.

Although it has been noted that the cache performance of object-oriented programs are inferior to that of traditional procedural programs [4], and that virtual functions and dynamic dispatch are expensive [9] [1] [17], the impact of object oriented programs on components of modern superscalar processors has not been fully understood. Superscalar processors employ significant overlap and parallelism between instructions and operations. It is not always known whether the cache miss delays and the branch misprediction delays are hidden by overlap of other operations or not. It is also difficult to construct, validate and verify accurate and detailed system level simulators that model all system level components. The availability of performance monitoring counters on modern processors has yielded the possibility of accurately characterizing various aspects of superscalar processor performance. This is the technique that we employ in this paper.

In this paper, we characterize the performance of several C and C++ benchmarks on an UltraSPARC-II processor [26] based system. The choice of C++ to represent OOP is appropriate, given its popularity for building large-scale software applications [2] [19] [20]. Its similarity to the structured programming language C, offers an opportunity to use structured programs for comparison. In addition the same compiler can be used to compile the programs written in both C and C++. This would allow us to see the performance difference caused by the micro-architecture and not because of the difference in code generated by the different compilers. Various architectural data related to execution behavior of the benchmarks are collected using a performance monitoring tool. Factors including CPI, instruction and data cache misses, processor stalls due to instruction cache misses and branch misprediction from actual execution of several programs are measured and presented.

This paper makes two contributions. First, it presents detailed information on the execution characteristics of several large C++ programs on a popular platform, and draws out a correlation between the various individual execution characteristics such as cache misses, branch mispredictions and the observed cycles per instruction (CPI). Second, it compares and contrasts the performance of C and C++ programs. The few available studies that evaluate the performance of C++ programs are based on profiling or simulation [4] [9], while our results are based on actual measurement of execution. Thus, our results confirm and augment the existing knowledge on behavior of C and C++ programs. Our experiments show that the programs in the C++ suite incur a higher CPI, higher i-cache misses, and higher branch mispredictions than the programs in the C suite. A strong correlation between CPI and branch mispredictions was observed for the C++ application programs. Program characterization studies such as this are extremely important for the design of efficient processors. [1]

1.1 Related Work

A major study in comparing the differences between C and C++ programs was done by Calder et. al. [4] which appeared in the Journal of Programming languages in 1994. They observed the behavioral differences between C and C++ programs on the DEC Alpha architecture using the ATOM tracing tool [25]. They presented statistics on the instruction mix, function size, basic block size, cache behavior etc. The results they obtained for cache behavior were based on simulation. Our studies are based on measurement of performance using the on-chip performance monitoring counters of the UltraSPARC-II.

Driesen et. al. [9] analyzed the dispatch overhead of the standard virtual function table dispatch on a set of C++ applications. They used executable inspection and processor simulation for their study. The cost associated with dynamic dispatch was measured for the C++ programs on a superscalar processor. Other related work includes efforts to optimize virtual function calls [1], reduce indirect function calls [5] and improve branch prediction [6] [7].

[1] A look at the advanced program of 1998 International Symposium on Computer Architecture [14] shows that the there are six workload characterization and measurement studies on several platforms, in the first two sessions of the conference.

1.2 Overview

In Section 2, we describe the benchmarks we used for our study. The UltraSPARC-II architecture is described in Section 3. The overall experiment methodology is outlined in Section 4. Statistics from profiling experiments are presented in Section 5 and the measured execution behavior is presented in Section 6. Section 7 offers summary and concluding remarks.

2 Benchmarks

Investigating the behavioral differences between C and C++ requires a suite of benchmark programs written in C and C++. The availability of the same applications written in the two different languages would have been ideal for comparing the language features. However, in the absence of such a benchmark suite, we used 6 C++ programs available from [9] [10] and 5 C programs from the SPECint95 benchmark suite [23]. Previous researchers [4] [9] selected these C++ programs on the criteria that they were large in code size and were application programs. The C++ programs that are used in this study make a large number of indirect function calls and heavy use of dynamic dispatch is employed. The programs we utilized for our study are described below.

2.1 C++ Programs

`deltablue`: An incremental dataflow constraint solver.

`eqn`: A type-setting program for mathematical equations. Three different files which were the output of *groff* were used as input.

`groff`: Groff version 1.10, a version of the *ditroff* text formatter. The input was a collection of man pages.

`idl`: SunSofts IDL compiler (version 1.3) using the demonstration backend which exercises the front end but produces no translated output.

`ixx`: IDL parser generating C++ stubs, distributed as part of the Fresco library. The program was developed and structured different from IDL, though it is similar.

`richards`: An operating system simulation benchmark. The benchmark schedules the execution of four different kinds of tasks. It contains a frequently executed send routine.

2.2 C programs

`go`: It is an example of the use of artificial intelligence in game playing. There is a significant amount of pattern matching and look-ahead logic in this benchmark.

`ijpeg`: This program does image compression/decompression on in-memory images based on the JPEG facilities. The common utility has been adapted to run using of memory

Program Name	Instructions	
	Static	Dynamic
deltablue	30541	40774931
eqn	43871	2853819
groff	36955	142627451
idl	84557	86803692
ixx	72744	30666373
richards	22026	6612956
go	96984	522932433
ijpeg	83079	1426133278
li	58114	167929839
m88ksim	53631	1098440
vortex	161821	2372772250

Table 1: Static and Dynamic code sizes for the C++ and C programs

buffers rather than reading/writing files. The benchmark application performs a series of compressions at differing quality levels over an images.

`li`: Li is a Lisp interpreter written in C. The workload used is a translation of the Gabriel benchmarks. This benchmark has been developed at Sun Microsystems and is based on XLISP 1.6. XLISP is a small implementation of Lisp with object-oriented programming.

`m88ksim`: M88ksim is a simulator for the 88100 microprocessor, written in C. It can measure the number of clocks which an 88100 microprocessor would take to execute a program. It is essentially an integer program, although the exact instruction mix of the simulator depends on the program being simulated. We pass the training input provided in the SPEC95 suite as input.

`vortex`: VORTEx is a single-user object-oriented database transaction benchmark which exercises a system kernel coded in integer C. The VORTEx benchmark is a derivative of a full object oriented database management system. VORTEx stands for "Virtual Object Runtime EXpository." Transactions to and from the database are translated though a schema. The schema as provided with the benchmark is pre-configured to manipulate three different databases: mailing list, parts list, and geometric data. The benchmark builds and manipulates three separate, but inter-related databases based on the schema. The workload of VORTEx has been modeled after common object-oriented database benchmarks.

The C programs from the SPEC95 suite were run with the *training* input, except for *ijpeg* and *vortex* which were run with the *ref* input.

Table 1 gives information about the static and dynamic size of the code for the different programs. The static and dynamic code size is different from what was observed in the study conducted by Calder et. al. [4], for the programs common to both studies. This is because the programs are being tested on a different hardware platform and the compilers used are different. However, it is seen that the program characteristics observed for the

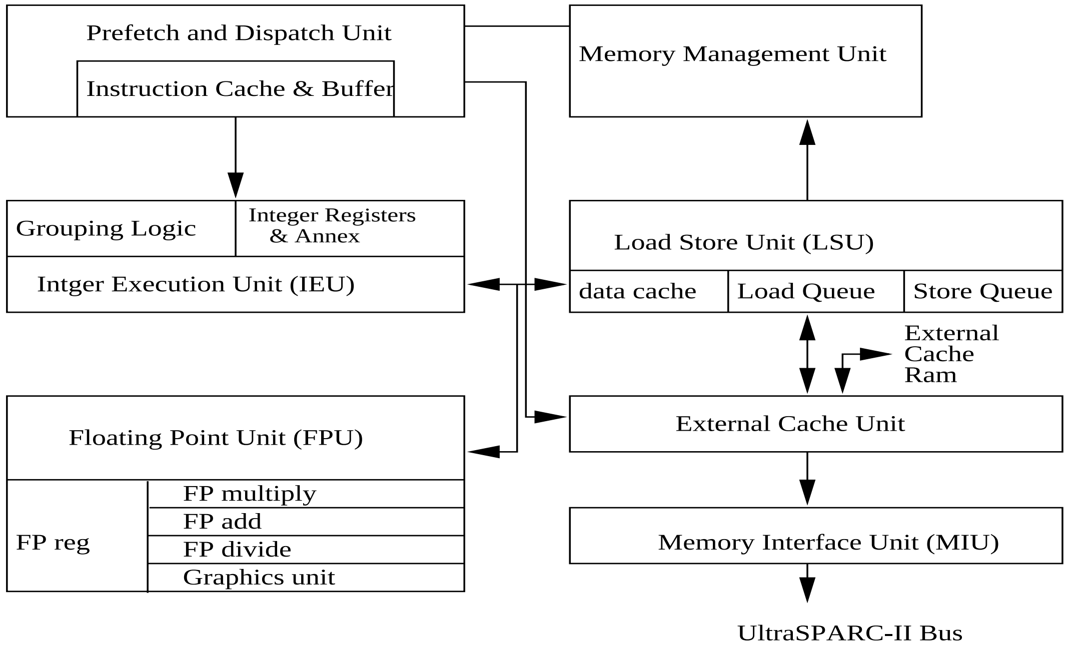


Figure 1: Functional Block Diagram of the UltraSPARC-II Processor

different platforms are similar. These characteristics are explained in detail in section 5.

3 The UltraSPARC-II Architecture

We give a brief description of the UltraSPARC-II architecture used to run the programs. The UltraSPARC-II is a high-performance, superscalar processor implementing a 64-bit RISC architecture [26]. It is capable of sustaining the execution of up to four instructions per cycle. This is supported by a decoupled prefetch and dispatch unit with instruction buffer. Pipeline execution is decoupled from data cache misses with load buffers on the input side and store buffers on the output side. Instructions predicted to be executed are issued in program order to multiple functional units and execute in parallel. Out of order execution is supported for these predictively-issued instructions. The instructions issued in one cycle may include instructions from different basic blocks, in order to increase instructions issued per cycle.

Integer Execution Unit (IEU) The main computational components of the IEU are two arithmetic logic units (ALUs). A multi-cycle integer multiplier and integer divider are also part of the IEU. Eight register windows and four sets of global registers are provided in the IEU. The number of functional units are important in the execution performance of programs on a superscalar processor. Insufficient number of units may cause resource conflicts or stalls in the pipeline.

Floating Point Unit (FPU) The separation of the execution units in the FPU allows issue and execution of two floating point instructions per cycle. Most instructions are fully pipelined, have a latency of three and are not affected by the precision of the operands (single or double precision).

Prefetch and Dispatch Unit The prefetch and dispatch units are used to fetch and store instructions before they are actually needed in the pipeline. A dynamic branch prediction scheme is used to prefetch across conditional branches. It is implemented in hardware and the outcome of a branch is based on a two-bit history of the branch. Up to 12 instructions can be buffered in the instruction buffer before they are sent to the rest of the pipeline.

Instruction Cache (I-Cache) The I-Cache is a 16-Kilobyte two-way set-associative cache with 32 byte blocks. The instruction cache returns up to 4 instructions from an 8-instruction-wide line. The I-cache performance would be an important measure to compare C++ and C programs. It is expected that object-oriented programs would have inferior cache behavior, due to its lack of tight-loops and more distributed control in general.

Data Cache (D-Cache) The data cache is a write-through non-allocating 16-Kilobyte direct-mapped cache with two 16-byte sub-blocks per line. The data cache miss rate is a measure of the locality of reference of a programs access to data in the stack, static data segment and heap.

External Cache Unit (ECU) The main role of the ECU is to handle I-Cache and D-Cache misses efficiently. The ECU can handle one access per cycle to the external cache. It also provides overlap processing during load and store misses, or it has non-blocking capabilities.

Load/Store Unit (LSU) The LSU is responsible for generating the virtual addresses of all loads and stores, for accessing the data cache. They also decouple a load miss from the pipe through the load buffer and decouple stores through a store buffer. One load or store can be issued per cycle. Since there is only a single LSU, a program having a large number of loads/stores would cause contention for this resource.

4 Experiment Methodology

Our experiments consisted on monitoring the execution of the selected benchmarks on the UltraSPARC-II. All the programs were compiled with optimization enabled (*-O2 -msupersparc*) using GNU gcc compiler (version 2.7.2). The operating system used was SunOS version 5.5.1. The performance of the programs was measured using the performance counters that are provided in the UltraSPARC chip. Important statistics like cycle count, instructions executed, cycle stalls, and cache information could be accessed using the performance counters. The performance registers were used to collect statistics for important units in the processor for different programs. The package *perf-monitor* [18] developed at the Swedish Institute of Computer Science was employed in this process. Table 2 lists the events measured using the performance monitoring counters.

We also present statistics from the profiling of the benchmark programs. *Spix* [24] a tool from Sun Microsystems was used to collect profiling data for the benchmarks. *Spixtools* with the help of *Shade* [22] allow the classification and analysis of instruction set usage which allowed us to gather valuable statistics on the program.

Event id	Performance Metric
Cycle_cnt	Accumulated cycles
Instr_cnt	The number of instructions completed
Dispatch0_IC_miss	Cycles I-buffer empty from I-Cache miss
Dispatch0_storeBuf	Cycles store buffer full
Dispatch0_FP_use	Cycles stalled waiting for fp dependency
Load_use	Cycles stalled waiting for load
Load_use_RAW	Cycles stalled on internal conditions
IC_ref	I-Cache references
IC_hit	I-Cache hits
DC_rd	D-Cache read references
DC_rd_hit	D-Cache read hits
DC_wr	D-Cache write references
DC_wr_hit	D-Cache write hits
EC_ref	E-Cache references
EC_hit	E-Cache hits
EC_wb	E-Cache misses that do writebacks
EC_rd_hit	E-Cache read hits from D-Cache misses
EC_ic_hit	E-Cache read hits from I-Cache misses

Table 2: Events measured using the on-chip counters on the UltraSPARC-II Processor and associated Performance Metrics

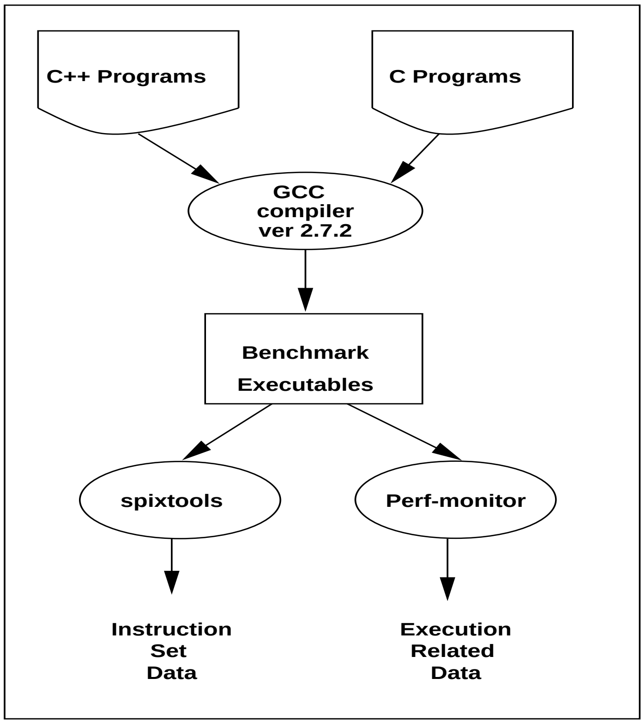


Figure 2: Experiment Methodology

5 Instruction Set Usage Statistics

In this section profiling information obtained using *Spix* are presented. At the bottom of each table we provide harmonic mean for the C programs and the C++ programs. At the end of the section the harmonic mean of the different program statistics obtained using *Spix* are plotted to present summary information for the two suites of programs.

5.1 Function Calls

The instruction level parallelism available from a program is heavily influenced by the frequency of function calls and average function size. Table 3 lists the number of function calls made in the benchmarks. The percentage of C and C++ function calls in the benchmarks are also shown. Most C++ programs often use functions defined in predefined C libraries. True C++ functions include methods and other functions defined in C++. We see that C functions account for a large percentage of functions in C++ programs, especially *ixx*. It is seen for C++ program *richards* that there is no significant C function calls. For C programs, the C function calls represent all the function calls made in the program. The C++ method calls and function calls are listed in third and fourth column. It is seen that the C++ functions form a major percent of the total function calls in a C++ program. This observation was also seen by Calder et. al. [4] in their studies on the Alpha architecture. The number of indirect calls in the programs are listed in Table 4. We see that indirect calls are more frequent in C++ programs than C programs. Only two of the C programs *li* and *ijpeg* show significant indirect calls. None of the other C programs show any indirect function calls. Table 4 also shows what percentage of all calls are indirect calls.

Dynamic function size is important in the performance of the program, since small functions have a greater function call overhead relative to larger functions. Table 5 gives information about the average size of the function calls and the number of instructions associated with each call. The first column gives the harmonic mean for the number of instructions executed for the C functions (per invocation) in the C and C++ programs. The average number of instructions per function invocation in the C programs is twice the corresponding quantity in the C++ programs. Hence C++ programs would benefit more from optimizations like inlining. The smaller function size for C++ functions can be attributed to the object-oriented design philosophy, where programs are written with small and specific functions (methods) for the objects in the class. This can cause an overhead in terms of register usage and saving.

5.2 Memory Operations

The frequency of memory operations were measured and compared for the C and C++ programs and are presented in Table 6. Overall C++ programs were seen to execute more load instructions and less store instructions than C programs dynamically. The difference in percentage of stores for C and C++ would have been less significant if it were not for the low percentage of stores performed by *idl*, which makes the mean for C++ lower than

Program Name	C Functions		C++ Functions	
	Number	%	Number	%
deltablue	485614	22.96	1629220	77.04
eqn	4554	10.38	6172	89.62
idl	529500	15.12	2972272	84.88
ixx	270687	34.09	523261	65.91
richards	–	–	240731	100
go	5637131	100	–	–
ijpeg	2163960	100	–	–
li	166578336	100	–	–
m88ksim	2403489	100	–	–
vortex	49679309	100	–	–
C++-mean	392560	21.77	5701791.10	80.41
C-mean	4622332	100	–	–

Table 3: Number of C Function Calls and C++ Function Calls in C++ and C Programs (Dynamic)

Program Name	Indirect Calls	
	Number	% of all Calls
deltablue	1419947	67.14
eqn	1313	2.99
idl	1691443	48.30
ixx	156593	19.72
richards	65790	27.32
go	–	–
ijpeg	108928	5.03
li	108511	0.06
m88ksim	–	–
vortex	–	–
C++-mean	6445.94	11.46
C-mean	108719.10	0.11

Table 4: Number of Indirect Function Calls in C++ and C Programs (Dynamic)

Program	C functions	C++ functions	Inst/Call
deltablue	15.37	17.67	19.23
eqn	37.84	64.84	207.15
groff	25.93	43.94	88.9
idl	40.07	11.80	23.67
ixx	14.56	26.61	37.38
richards	–	12.32	27.44
go	103.16	–	92.59
ijpeg	261.03		658.40
li	35.21	–	76.68
m88ksim	42.31	–	52.82
vortex	46.45	–	47.56
C++-mean	22.35	20.11	34.57
C-mean	57.41	–	76.54

Table 5: Harmonic Mean of Number of Instructions per Invocation and Instructions/Function Call for C++ and C programs

C programs. The *idl* program was seen to have the lowest frequency of stores in the study by Calder et. al. [4] also.

5.3 Control Transfer Instructions

To better understand the behavior of programs it is important to study the frequency of control transfer instructions like branches, jumps and calls which cause a break in the program flow. The higher frequency of these instructions results in a smaller basic block size making it difficult for a superscalar processor to find the required nonspeculative instruction level parallelism (ILP).

The measurements for the branch, jump and call instructions (static and dynamic) are given in Table 7. The C++ programs contain a higher percentage of branch instructions than the C programs. Multiple instruction issue processors need to predict several branches every cycle in order to achieve the full instruction issue bandwidth and to fully utilize all the functional units. The observed instruction mix emphasizes the need for better branch prediction schemes to improve the performance of C++ programs, especially regarding optimization of virtual function calls. For jump and call instructions also the C++ programs have a higher mean than the C programs. The number of function calls is less in *ijpeg* as compared to the total instructions in the program, and we see that it has the least number of calls, jumps and branches in the table. *ijpeg* is a program which has little control flow, this is also indicated by its high number of instructions per function call. This is why *ijpeg* has a lower percentage of control transfer instructions as compared to the other programs. Without considering *ijpeg* the mean for the C programs is still significantly lower than the

Program Name	Static		Dynamic	
	%Loads	%Stores	%Loads	%Stores
deltablue	13.42	8.47	25.51	6.17
eqn	15.19	6.89	21.77	12.29
groff	12.73	7.72	14.17	8.58
idl	23.93	10.20	21.26	2.72
ixx	17.73	7.10	15.18	7.84
richards	12.01	8.39	26.18	8.38
go	16.29	6.18	20.53	7.35
ijpeg	19.10	10.06	17.30	6.94
li	14.66	8.28	20.11	10.16
m88ksim	13.10	7.96	14.58	8.39
vortex	17.02	7.93	18.43	8.91
C++-mean	14.98	7.99	19.55	6.15
C-mean	15.77	7.88	17.91	8.19

Table 6: Percent of Static and Dynamic Load and Stores for C++ and C Programs

Program Name	Static			Dynamic		
	%Branches	%Calls	%Jump	%Branches	%Call	%Jump
deltablue	13.98	4.25	2.11	12.46	2.49	9.08
eqn	12.96	5.80	2.62	19.18	0.98	1.19
groff	14.08	4.73	1.98	18.70	1.78	1.91
idl	9.74	4.56	3.55	16.92	2.58	6.93
ixx	12.27	5.99	3.32	15.46	3.12	4.36
richards	13.72	4.24	2.32	18.60	2.65	4.63
go	12.38	3.10	1.24	13.26	1.14	1.15
ijpeg	11.48	2.75	2.98	8.07	0.14	0.20
li	13.48	5.75	1.75	17.72	1.46	2.75
m88ksim	13.49	4.46	2.36	15.20	1.97	2.04
vortex	12.09	6.34	1.05	17.48	2.28	2.27
C++-mean	12.58	4.83	2.52	16.50	1.95	2.90
C-mean	12.53	4.02	1.61	13.20	0.51	0.69

Table 7: Control Transfer Instructions for C++ and C programs

mean for the C++ programs.

Figure 3 illustrates the summary of the instruction usage statistics. The mean presented is the harmonic mean for all the programs in each suite.

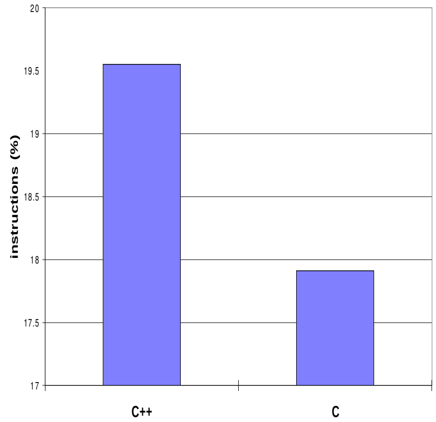
6 Measured Execution Behavior

It is seen that C++ programs execute more function calls than C programs. It is also seen that there are fewer instructions per function call in C++ compared to C programs. This naturally makes it difficult to find parallelism in the code for C++ programs leading to inferior performance. Also the increased number of branches and abundance of virtual functions would cause lower performance for C++ programs. To quantify the performance, we measure cycles per instruction (CPI) based on total execution time and instructions completed available in the on-chip counters. We also measure data cache, instruction cache and external cache miss ratios which can be compared to previous simulation studies done by Calder et. al. [4]. The number of stalls caused in the processor due to instruction cache misses, branch mispredicts, store buffer full conditions and data loading delays are also measured to analyze the bottleneck.

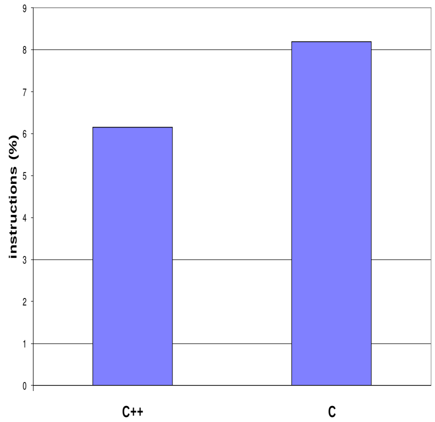
The statistics obtained using *perf-monitor* while running the programs are shown in Table 8. The first column in Table 8 lists the cycles per instruction(CPI) for the different programs. It is seen that the mean CPI for C++ programs is higher than the mean for the C programs. The C++ programs show consistently higher CPI than the C programs for all programs except *eqn*. The *eqn* benchmark has a high number of instructions per call resulting in a high basic block size. This is what helps *eqn* to consume less cycles per instruction than what is observed in the other C++ programs. Typical C++ programs do not show such a high number of instructions per function or a high basic block size. Similarly, *ijpeg* in the C benchmarks shows a higher than average instruction count per function (more than *eqn*) and its CPI is only 0.83. Again this is because it is a program that does manipulation of image files where it is easy to find parallelism. Overall it can be concluded that C++ programs take more cycles per instruction than C programs. The mean CPI for C++ programs is 1.57 compared to 1.17 for the C programs.

The second column in Table 8 lists the percentage of the instruction cache misses. It is expected that C++ programs would have a higher miss rate because of inferior locality due to its more distributed control and larger number of function calls. In general, the instruction cache miss ratio is worse for C++ programs compared to C programs.

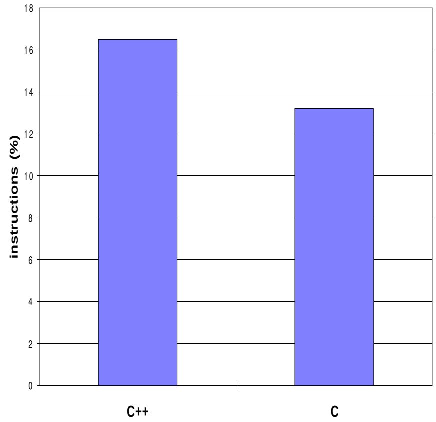
The third column in Table 8 lists the miss rate for the data cache for the different programs. From the harmonic mean it can be seen that C programs have a slightly higher miss rate than C++ programs. This is a surprising result considering that C++ programs allocate significant amount of data on the heap. The mean for the C++ programs is affected by *richards* which has a very low miss rate of 0.04%. It was observed that the C++ program *deltablue* exhibited the highest miss rate among all the programs considered and this was the trend seen by Calder et. al. [4] also.



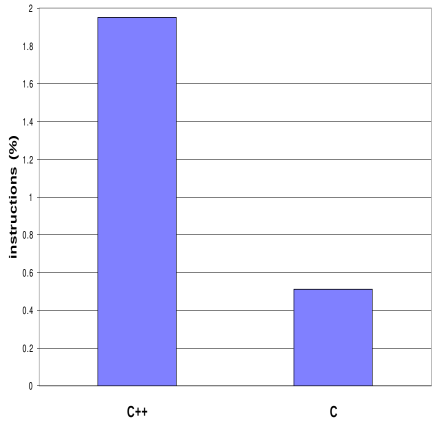
Load instructions



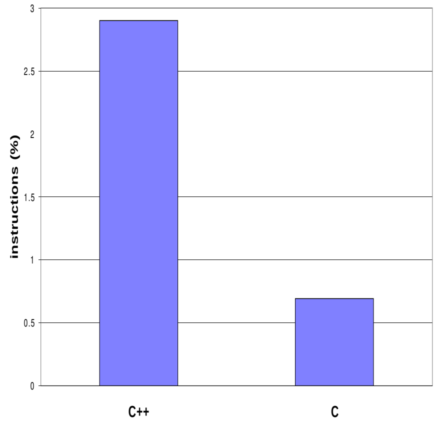
Store Instructions



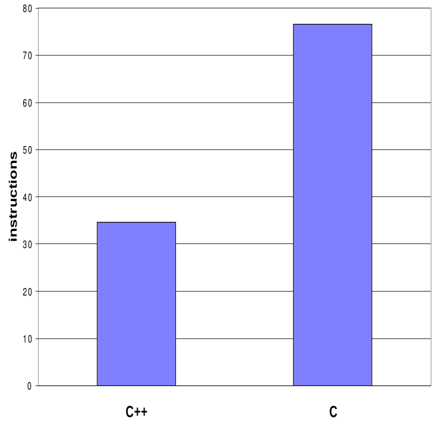
Branches



Calls



Jumps



Instructions per Function Call

Figure 3: Summary of Instruction Mix and Function Size Statistics for C and C++ Programs. The harmonic mean of the programs in each suite is presented here.

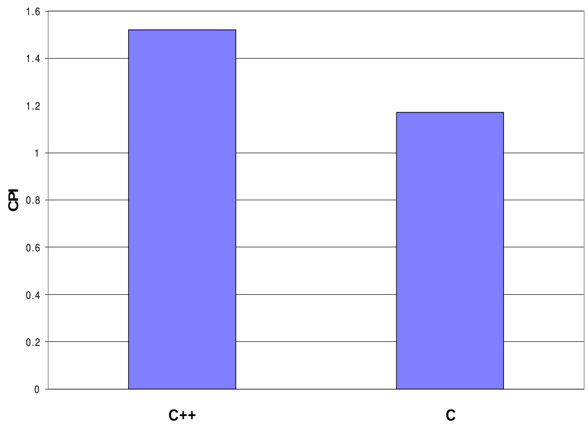
Program		Cache miss rates			Percent of Cycles stalled due to:			
Name	CPI	IC_miss	DC_miss	EC_miss	Inst Cache	Mispredict	Store Buffer	Data Cache
deltablue	2.13	2.20	31.33	2.82	7.88	18.75	1.08	0.46
eqn	1.15	1.16	7.90	2.35	4.85	2.99	0.80	4.28
groff	1.65	6.62	9.03	4.72	22.46	10.29	0.32	0.82
idl	1.47	1.09	10.79	3.51	4.10	11.26	0.56	0.52
ixx	1.48	2.70	12.62	3.80	8.44	8.40	3.25	1.98
richards	1.47	0.02	0.04	0.05	0.12	16.03	0.02	5.67
go	1.29	5.66	9.16	1.76	13.43	7.57	0.19	0.79
ijpeg	0.83	1.18	13.35	0.96	1.98	1.64	0.21	1.15
li	1.18	0.36	11.52	0.49	1.47	7.32	0.05	3.65
m88ksim	1.47	2.82	18.52	8.53	13.44	5.88	8.97	2.31
vortex	1.39	2.63	11.44	8.19	12.19	5.11	1.11	0.73
C++-mean	1.52	1.82	11.33	3.25	6.75	7.22	0.67	2.29
C-mean	1.17	1.17	13.11	1.20	3.57	3.51	0.18	1.73

Table 8: Performance of Large Benchmarks on UltraSPARC-II (measured using on-chip counters)

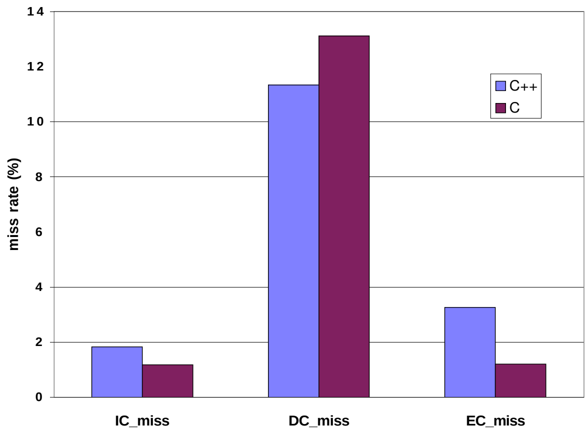
In column 4 of the table, we list the miss rates for the external cache. It is seen that the mean for the C++ programs is significantly higher than that for the C programs, by a factor of almost 3.

Columns 5, 6, 7, and 8 list the percentage of processor cycles stalled due to instruction cache misses, branch mispredictions, store buffer full conditions, and data load waiting. As expected from the instruction cache miss rate, we see that the average percentage of stalled cycles due to instruction cache misses is higher for C++ programs than C programs. But if we carefully analyze the individual programs, we see that one program *groff* accounts for a lot of these stalls and otherwise the means are comparable for the C and C++ programs. As expected, the cycles stalled due to a branch mispredictions is significantly higher for C++ programs than C programs. In column 7 we present the cycle stalls caused due to the store buffer being full. Although C++ encounters more stalls due to store buffer full conditions, the absolute amount of such stalls is smaller than the stalls due to other listed factors. It can also be observed that C++ programs encounter more delays waiting for data than C programs. While there is good correlation between instruction cache miss rates and stalls due to instruction cache misses, we could not find a correlation between data cache miss rates and load wait cycles. Figure 4 illustrates the summary of all the measured performance metrics. The harmonic mean for all the programs in each suite is used in these plots.

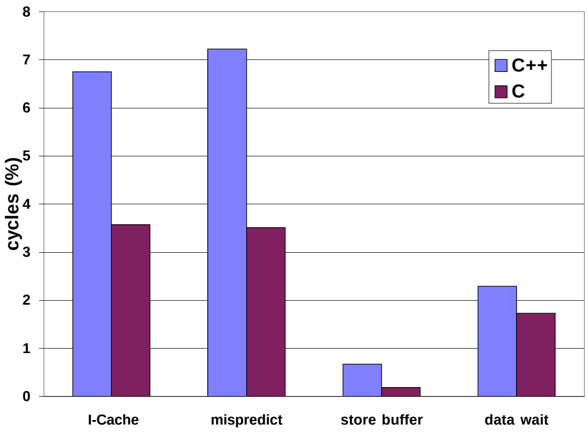
Figure 5 illustrates the correlation between CPI and percentage of cycles stalled for branch mispredictions. A very strong correlation is observed between the CPI of the C++ programs and the cycles due to branch mispredictions. branch mispredictions.



(a) Cycles per Instruction



(b) Miss Rates for IC, DC and EC.



(c) Percentage of Cycles Stalled

Figure 4: Summary of Measured Performance Factors for C and C++ Programs. The harmonic mean of the programs in each suite is presented here. Figure (a) illustrates the mean CPI, while figure (b) illustrates the mean instruction cache, data cache and external cache misses. Figure (c) illustrates the percentage of cycles stalled due to i-cache misses, branch mispredicts, store buffer full conditions and data wait conditions.

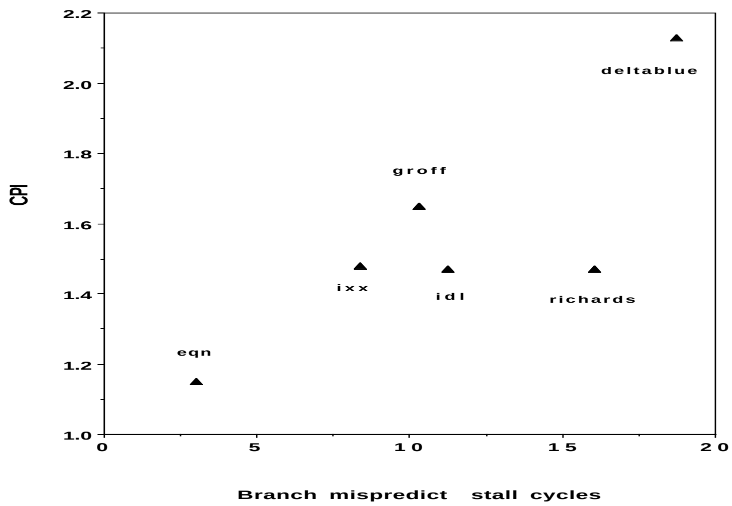


Figure 5: Correlation between CPI and branch mispredictions for C++ programs

7 Summary and Conclusion

In this paper we measured the dynamic execution behavior of 6 large C++ programs and 5 large C programs on the UltraSPARC-II. The performance monitoring counters on the processor were utilized to perform measurement of CPI, cache miss rates, processor stalls due to cache misses, branch mispredictions, etc. This kind of a study based on measurements could not be done a few years earlier because processors did not incorporate such performance counters. Our studies confirm and augment results from earlier profiling and simulation studies on characterization and optimization of object oriented programs [4] [9].

We performed profiling studies as well as measurement and the major observations from profiling are the following.

- The C++ programs contain approximately 21.3% transfer of control instructions, as opposed to 14.5% in the C programs.
- The fraction of memory access instructions in the C and C++ programs were approximately the same although the C++ programs performed more loads than C programs and the C programs performed more stores than the C++ programs.
- The C functions were seen to execute twice the number of instructions performed by the C++ functions (or in other words, the number of instructions per function call in C++ was twice than that of C functions).

The important observations that were made based on actual machine measurement are the following.

- The C++ programs exhibited a worse CPI than the C programs (1.57 versus 1.17)

- The C++ programs encountered higher instruction cache miss rates and external cache miss rates than the C programs.

- The data cache miss rates for both C and C++ programs were seen to be similar. This was surprising, since C++ programs allocate a significant amount of objects to the heap.

- The C++ programs wasted more cycles in processor stalls due to instruction cache misses, branch mispredictions, and store buffer full conditions.

- Among all the factors analyzed, the strongest correlation to the high CPI of C++ programs was exhibited by branch mispredictions.

Our study provides valuable insight into the needs of object-oriented software and the stresses placed on processors by contemporary software development practices. We use measurements on an a superscalar processor to confirm prior work in comparing C and C++. To the best of our knowledge this is the first reported study that uses measurement on a superscalar processor, to explain how features of object oriented programs affect processor execution. Evaluation using measurement deprives one from varying cache and processor features as in simulation, however yields results that can be heavily trusted. We plan to further study the differences in executing such large C and C++ programs on different architectures, including the PentiumII which has a CISC instruction set architecture. We would also like to include other popular object oriented languages like Java in future characterization studies.

Acknowledgments

This work was supported in part by NSF grant CCR-9697098. We would like to thank Magnus Christenson and other researchers at the Swedish Institute of Computer Science for developing and making available the *perf-monitor* tool. We also express our thanks to those who made available the benchmarks at the UC Santa Barbara and gdb ftp sites. Acknowledgment is also provided for Sun Microsystems for making available the *Spix* and *Shade* performance analysis tools. UltraSPARC is a trademark of Sun Microsystems.

References

[1] G. Aigner and Urs Holzle, Eliminating Virtual Function Calls in C++ Programs, ECOOP '96 Conference Proceedings, Linz, Austria, July 1996, Springer Verlag LNCS 1098, pp. 142-166.

[2] Grady Booch, "Object Oriented Design with Applications", Benjamin Cummings, 1991.

[3] T. Budd, "An Introduction to Object-Oriented Programming", Addison-Wesley, 1991.

[4] B. Calder, D. Grunwald, and B. Zorn, "Quantifying Behavioral Differences Between C and C++ Programs", Journal of Programming Languages, Vol. 2, No. 4, pp. 313-351, 1994.

[5] B. Calder and D. Grunwald. "Reducing Indirect Function Call Overhead in C++ Programs", 21st Annual ACM Symposium on Principles of Programming Languages, pp. 397-408, Jan 1994.

[6] B. Calder and D. Grunwald, "Reducing Branch Costs via Branch Alignment", In 6th Symposium on Architectural Support for Programming Languages and Operating Systems, pp. 242-251, October 1994.

[7] B. Calder and D. Grunwald, "Fast and Accurate Instruction Fetch and Branch Prediction", In 21st Annual Symposium of Computer Architecture, pp. 2-11, April 1994.

[8] B. J. Cox, "Object-Oriented Programming - An Evolutionary Approach", Addison-Wesley, 1986.

[9] K. Driesen and Urs Holzle, "The Direct Cost of Virtual Function Calls in C++" Procedings of OOPSLA-1996, San Jose, CA, October 1996.

[10] Free Software Foundation, Boston, MA. ftp://prep.ai.mit.edu/pub/gnu/

[11] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Patterns : Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[12] U. Holzle and D. Ungar, "Do Object-Oriented Languages Need Special Hardware Support?", Proceedings of OOPSLA-1995.

[13] U. Holxle and D. Ungar, "Optimizing Dynamically Dispatched Calls with Run-Time Type Feedback", Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 94), Orlando, June 1994.

[14] ISCA 1998 Advance Program, http://www.cs.wisc.edu/isca98

[15] M. Johnson, "Superscalar Microprocessor Design", Prentice-Hall, 1991.

[16] Jouppi N. P., "Available Instruction-Level Parallelism for Superscalar and Superpipelined Processors", IEEE Transactions on Computers, Vol 38, no 12, pp 1645-1658, December 1989.

[17] M. A. Ellis and B. Stroustrup. “The Annotated C++ Reference Manual”, Addison-Wesley, Reading, MA, 1990.

[18] Magnus Christensson, Perf-Monitor, A Package for Measuring Low-Level Behavior on the UltraSPARC, Swedish Institute of Computer Science.

[19] B. Meyer, “Object-Oriented Software Construction”, Prentice Hall 1988.

[20] B. Meyer, “From Structured Programming to Object-Oriented Design: The Road to Eiffel”, Structured Programming, 1:19-39, 1989.

[21] D. R. Musser and A. A. Stepanov, “Algorithm-Oriented Generic Software Library Development”, Rensselaer Polytechnic Institute Computer Science Department Technical Report 92-13, April 1992.

[22] Robert F. Cmelik and David Keppel, “Shade: A Fast Instruction-Set Simulator for Execution Profiling” Sun Microsystems Inc, Technical Report SMLI TR-93-12, 1993.

[23] Standard Performance Evaluation Corporation, Manassas, VA.

[24] “Introduction to SpixTools”, Spixtools Users Manual, Sun Microsystems, Inc, 1993.

[25] A. Srivatsava 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, pp. 196-205, Orlando, Fl, June 1994.

[26] The UltraSPARC-II Data Sheet, STP1031, Sun MicroElectronics, July 1997.