Performance Impact of Object Oriented Programming

R. Radhakrishnan, A. Muthiah and L. John
Electrical and Computer Engineering Department
University of Texas at Austin
Austin, TX 78712
(512) 232-1455
ljohn@ece.utexas.edu
radhakri@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 the software engineering community has embraced object-oriented programming for these benefits, it has not been clear what performance overheads are associated with this programming paradigm. In this paper, we present some quantitative results based on the performance of a few programs in C and C++. Several programs were profiled and the statistics of several program executions at various compiler optimization levels were generated on two architectures, the MIPS and SPARC. One observation was that in spite of a static code increase in C++, the dynamic instruction counts were either comparable or smaller in C++. However the cache miss ratios and traffic ratios were significantly worse for C++ (often twice). It was also seen that some of the C++ features such as function overloading and free unions did not incur any run time overhead. These results should be of interest to computer architects and compiler writers who are designing and optimizing systems for executing object-oriented programs. Although it is difficult to devise representative and comparable programs in two different languages that use two different programming paradigms, we believe that our study will provide some initial data points and hopefully will lead into more related research.

Keywords: OOP, Cache Performance, Instruction Mix, Program Behavior, Quantitative Approach.

** This work was supported in part by the National Science Foundation under grant number CCR-0624378 and a grant from Oak Ridge Associated Universities.*

1 Introduction

Object-oriented programming (OOP) is one of today's buzzwords [10]. On one hand, it is a programming paradigm. On the other hand, it is a set of software engineering tools to build more reliable and reusable systems [10]. It has been widely accepted today that object-oriented concepts will produce programs which are superior than conventional programs in terms of robustness, extendibility, reusability, and maintainability [4] [9].

Object-oriented means good, not just in the computer press, but even in the business press and marketing brochures [13]. With all the hype it is not surprising that many are skeptical, but most programmers agree that the claimed advantages of robustness, extendibility, reusability etc exist. The first and fundamental approach to writing computer programs, machine language programming or even assembly language programming, was too low-level, verbose and difficult to maintain [11]. The second historical approach, which introduced data abstraction into languages, structured high-level languages, made programs easier to code and maintain, but even with excellent optimizing compilers, often, assembly programs run faster than high-level code. Structured high-level programs that model the behaviors of related objects tend to contain messy multi-way case statements. OOP, the next historical try, represents another significant improvement to programming style. How much does this cost us in terms of performance? We have heard users of simulation languages (which are object-oriented to a large extent) complain about slow programs and high memory consumption. At times, we have also heard that the newer versions of certain software packages are slow due to developing the package in an object-oriented language such as C++. Considering all these, we believe that it is a significant research problem to examine whether we are losing performance while adopting the OOP paradigm. Are we sacrificing performance in order to achieve the maintainability and reusability and if so, how much performance are we losing?

2 Features of Object Oriented Programs

An object is similar to a value in an abstract data type - it encapsulates both data and operations on that data. In simple words, an object is data and code stuck together. The major features of object-oriented programs are described below.

Encapsulation:

Encapsulation allows OOP to integrate control and data into an object thus hiding all the details of the object in the object itself [7]. Each data-item is joined exclusively with the procedures manipulating this data-item. The attached procedures of a data-item are called methods, all methods together are called interface, and the data item together with its methods is called the object. A very important feature of an object is that some of its data or code can be hidden from other objects. This isolation of object's internal structure is called encapsulation. Things that are visible to other objects are the interface. Encapsulation allows the internal structure of objects to change drastically without affecting other objects as long as the interface remains fixed.

Inheritance:

Inheritance is deriving new objects from existing objects [7]. It is the mechanism for creation of new classes by extending and adapting old classes. It enables the reuse of behavior of already defined classes in the definition of a new class. In addition to the attributes and methods contained in the new class, more attributes and methods can be inherited from other classes. Inheritance

makes "programming by differences" possible. Rather than write each class from scratch, the programmer can adapt an existing class by changing only the parts that need to be different. Inheritance gives the child class the methods and attributes of its parents. Therefore, methods applicable to objects of the parent class can also be used on objects of the child class.

Polymorphism:

Polymorphism is creating functions that have many forms and postponing the function invocation and its binding to run time [7]. In OOP, the same operation can be applied to different kinds of objects. Parent objects can invoke derived object's member functions selectively at run time. Thus polymorphism provides virtual function interface which will allow late binding and provide full flexibility in program extensions [7].

Hierarchical ordering is one of the most powerful ways by which we could manage complexity, i. e. organizing related concepts into a tree structure with the most general concept as the root. Virtual functions can often be used to define a set of operations for the base class, which is at the top of the hierarchy.

The introduction of objects, classes, encapsulation, inheritance and polymorphism into programming represents an attempt to solve some of the most difficult problems facing software development. Classes and objects could become the equivalent of interchangeable, standard components. Writing a program would consist of selecting parts from a catalog and snapping them together. Quality of software would significantly improve if it is built from pretested standard parts. And this is what object oriented programming has to offer.

3 Objectives

In this paper, we attempt to make a study to quantify the performance differences between object-oriented and non object oriented programs. The same programs are written in an object-oriented paradigm (using C++) and in a conventional programming paradigm (using C). The study is done using C and C++ due to the popularity of the languages and the potential comparability between the two. Since C is a subset of C++, we could compile both the C and C++ versions of the test programs using the same compiler, thus eliminating differences arising from compilers.

We study several programs that use the key features of OOP languages. We perform the experiments on two platforms, the Sun SPARC and the DEC 5000 workstations, that use the SPARC and MIPS R3000 processors respectively. The execution of the two programs are profiled using instruction profiling tools such as *pixie* on the MIPS and *spix* on the SPARC. We study several program characteristics and the instruction and data cache behavior of the programs. Cache profiling is performed using *qpt* and *dinero* tools that are available from the University of Wisconsin, Madison [15].

4 Related Research

There has not been a lot of research investigating the behavioral differences between object-oriented and non-object-oriented programs as it applies to program execution efficiency. Calder et. al. [3] quantified the behavioral differences between C and C++ programs on the DEC Alpha architecture using the ATOM tracing tool [12]. They observed several differences including

1. C++ programs often perform an order-of-magnitude more indirect procedural calls.
2. C and C++ programs have basic blocks of approximately the same size.
3. C++ programs issue more loads and stores than C programs.
4. C++ programs have worse instruction cache locality.
5. In data cache behavior, there is not much difference between C and C++ programs.

They used 11 C programs including the six SPEC92 C programs and 12 C++ programs from different sources. They welcomed further research on other platforms and using other programs to confirm or contradict their results.

Holzle and Ungar [5] presented results on the behavior of an object-oriented language called SELF. SELF is a pure object-oriented language like Smalltalk-80. Holzle et. al. [5] compared SELF programs with C programs from the SPEC suite. They stated that they had only two small programs that allowed direct comparisons between the different languages. They observed that SELF is very much like C in most features like basic block size, frequency of various classes of instructions etc.

5 Experiment Methodology

This sections describes our benchmarks, profiling tools, and performance metrics.

5.1 Benchmarks

A major problem in our research was the non-availability of object-oriented programs and their equivalent non object-oriented counterparts. Calder et. al. [3] used a representative set of C programs and a representative set of C++ programs. Calder et. al. mention that many of their C++ programs were not written using object-oriented methodology. While it is okay to include such programs in a set of representative C++ programs (representative of what people do with C++), we were interested in comparing the performance impact of the OOP paradigm itself and needed programs designed using OOP methodology and equivalent programs designed in a procedural language. So we had to construct the programs in the two methods ourselves. This limited the size of our programs but the static size of our programs are comparable to the static size of many of the SPEC programs. We developed two large database programs and six smaller programs each of which illustrates some feature pertinent to object-oriented methodology. There are three different inputs given to the first database program, therefore a total of ten different cases are used to compare the performance between object-oriented and non object-oriented programs. The programs may be viewed at http://www.ece.utexas.edu/projects/ece/hpcl/oop.html.

Automated University Personnel (udb)

In this application, a system that keeps track of personnel and students in a University is designed. The personnel include the top level administrators, staff in the administrative offices such as the admissions office, financial aid office, etc. It also includes staff in the various colleges and departments, deans of the colleges, chairs of departments, faculty and staff in the various departments, students in the various departments, etc. Faculty and staff have the common attributes of

social security numbers, name, address, salary, insurance, etc but faculty have grants and contracts in addition. Students share many of the common attributes that faculty and staff have, however, they have some special attributes that are unique to them such as GPA, list of courses registered, etc. Here faculty, staff and students are alike in that all of them are persons with the same common attributes such as last name, first name, address, social security number, phone number etc. Students have GPA that are unique to them while faculty has grants and contracts. This can be modeled effectively and efficiently in an object-oriented model where we set up a base class called *Person* and derived classes called *faculty, staff* and *students* which inherit common attributes from the base class. The property of inheritance helps in the efficient implementation of this aspect.

By property of encapsulation, a class will encapsulate both data as well as functions associated with the data. Student will encapsulate their attributes such as name, GPA, etc as well as functions associated with it such as add another student to the list, print the record information on screen/printer etc. Similarly faculty would contain their attributes as well as functions to add records and print records. One cannot fail to notice that some of these functions such as add another record to a particular class or print that record are essentially the same for different classes except that the number and types of arguments are different for faculty and students. This is made possible by the property of polymorphism.

Parts Management System (dbc)

This is a database program that maintains a database to keep track of inventory in a warehouse. The inventory consists of machine parts that have a unique identification number, part description, maximum quantity and current stock-on-hand as attributes. The information is maintained in a separate file. The operations that the system supports are, to add new machine parts into the inventory, update the information on the existing stock, and delete parts that are obsolete. The user can also get current information about existing stock, and also query for any particular machine part. In the object-oriented approach, a base class called *Inventory management* is set up with the attributes parts and number of records. The operations add, update, delete, browse and print are defined as methods in the base class. The class *Parts* is derived from the base class *Inventory management*. If the warehouse is to diversify and include clothing as well, the system will be modified by defining the object clothing to be similar to object parts with additional data, namely color and size. Hence if an object-oriented approach is adopted, the high level solution is optimal as there is no duplication of code. Reuse of code is one of the major advantage of an object-oriented approach and a cascading effect of easy maintenance automatically follows. On the other hand, for the non object-oriented approach all algorithms have to be rewritten for the new addition and the structure parts has to be duplicated. This can get complicated depending on the magnitude of the application. Problems such as duplicate function names should also be handled.

Miscellaneous Programs (Microbenchmarks)

Several microbenchmarks were created and each contained specific features pertinent to C++, so that the effect of these features on program performance could be observed. These programs are described below:

Mem This program allocates memory for an employee structure and stores information about the employee in the allocated structure and prints out the information in the stored structure. Dynamic allocation in the C programming language is an extremely important part of the language. Almost all large C programs extensively allocate and free memory on demand. However malloc() and free() commands are not part of the standard language. They are part of add-on libraries (stdlib.h,malloc.h) into scope. In designing C++, dynamic allocation was considered im-

portant enough to include it in the standard language. This program is designed to illustrate the performance impact of this C++ feature.

intnce This program creates a class animal with certain common universal characteristics and a class dog and a class cat which along with characteristics of an animal have some very special characteristics belonging exclusively to them. The same was created in C without the advantages of being able to use property of inheritance. This program is an example of one of the most important features of object-oriented programming i.e inheritance. In C since inheritance cannot be accommodated, the properties have to be repeated across the structures. Hence, if one minor property has to be modified all objects with that property have to be modified whereas in OO approach only the parent class needs to be modified and all derived classes will automatically reflect changes made in the base/parent class.

cons In this program we set up a class called rectangle which can be instantiated in 2 ways, either with user specified coordinates or with a default setting (if no coordinates are specified). The same can done in C without the advantages of being able use constructor initialization lists. A constructor is a method of a C++ class which always has the same name as the class and is automatically invoked by the C++ program when the instance comes. If no method is defined as a constructor a default constructor is defined by the compiler. Therefore in this program if an object such as a rectangle is instantiated with parameters such as height and width the area will be calculated with those parameters and in case of instantiation with NULL parameters some initial values will be assumed as specified in the constructor of the object with NULL parameters. However to implement the same in C we will have to define the same object twice or call a function once to set the values if given or call another function to set some default values if not specified and then call the area function. This C++ feature contributes to the ease of project management, and we are interested in finding out the difference in performance due to this feature.

ovld This program was designed to investigate the run-time performance impact of function overloading. The program is about geometrical objects such as rectangle, circle, square etc. All these objects have one thing in common, namely area. Therefore depending upon the geometrical object, the formula used to calculate the area will vary but the fact that all 3 of them are just "areas" does not change (because it is a common property shared by all geometrical objects). Therefore with one area function in C++ we can calculate the area (because the compiler constructs what is called as a Vtable) which will resolve which formula to use based on the geometrical object that calls this common area function at run time resolution. One cannot unfortunately do the same in C because of the lack of provision for run time resolution. As in C++ code, functions with same name will not be allowed in C. In C++, overloading is allowed and is in fact used quite frequently. The compiler can determine the proper function to call by the data type of the actual parameters. This is very useful because one does not have to browse through a million line code project to make sure function names are not the same. Therefore in C care should be taken to make sure that these functions are called by different names even though they perform the same function. In the C version of this program, the function area() is changed to 3 different names, mainly because the compiler will have to resolve which function is being called. The ability to overload function names is part of the concept of polymorphism.

ref This program is to illustrate the easy C++ way to pass parameters by reference to the called program. This program adds up the intermediate values between the range that you pass and stores the result in the variable result which is passed as a global. In the program in C++ you will note that, there exists a new way of passing variables, namely by reference. To achieve the same in C, that is, to achieve the same effect of passing k and accepting it as an address, which is

&result in C, we will have to pass &k as formal parameter and declare result as a pointer. Every reference to result must be prefixed with an asterisk (*) to dereference the value. Forgetting an asterisk leads to serious problems. The method in C++ is a lot more straightforward. We want to investigate whether that has any performance impact.

unn This program is to investigate the effect of free unions on performance. The program has an object "contact" that could be a *customer* or an *employee*. For *customer*, the id we need is *customer id* and for employee the id would be *emp id*. Both of them have a common feature namely the name field. But for customer we would be interested in payment due and for employee the needed field is salary. Therefore the best representation for such a diversified record structure would be to have a struct called *contact* and two unions within this structure to be able to choose only the fields we are interested in at any given time based on type of contact i.e customer or employee. But the access of individual fields in C is much more complex than in C++ because the embedded union within the structure have to be referred to by a union name. Unions in C++ are similar to those in C i.e each member having the same starting address but there are some differences. A union does not need to be named in C++ and can be defined with another union, struct or class. Unions without names are called *free unions*.

5.2 Profiling Tools, Performance Metrics, Configurations

The benchmark programs were compiled with g++ compiler on the Sun SPARC and DEC MIPS platforms with compiler options set to the default (-O1) and also the highest level possible (-O4) (in separate experiments). On the MIPS architecture, *pixie* and *pixstats* were used to profile the executables. Similarly, *spix* and *spixstats* were used on the Sun SPARC architecture. *Pixie* and *spix* annotate the executable and when the annotated executable is run, several information about the program execution is recorded. *Pixstats* and *spixstats* use this information and generate statistics showing static and dynamic instruction mix, branch behavior, register usage etc.

The cache performance of the programs were studied using *qpt* and *dinero* from the Wisconsin WARTS toolset [15]. The programs were compiled with the default option and also the highest level possible (-O4) and the trace was generated using *qpt*. Qpt rewrites the program's executable file by inserting code to record the frequency of each basic block or to trace every instruction and data reference.

The trace generated was fed to the *dinero* simulator to study the cache performance. Studies were done for cache sizes of 1, 2, 4, 8, 16, 32 and 64 kbytes. Block sizes of 8, 16, 32 and 64 bytes were used for each cache size. All caches simulated were direct mapped. The fetch policy used was demand fetching, i. e. no prefetching was used. Only blocks that are needed to service a cache reference are fetched. The copy back write policy is used; so main memory is updated only when a dirty block is replaced.

To measure the cache performance we use the cache *miss ratios* and the *traffic ratio* generated by the different programs. Cache miss rate is a measure of the locality of reference for a program, hence the percentage of misses was found for the various cache configurations. As the traffic ratio is also an important parameter affecting the performance, we calculated the traffic ratios for all the different programs. *Traffic ratio* is defined as the number of bytes transferred between the cache and main memory divided by the number of bytes transferred between the processor and main memory in the absence of a cache. In other words, it is the ratio of the number of main memory byte transfers in the presence of a cache to that in the absence of a cache. Both these metrics were obtained directly from *dinero*.

Program Name	Program	optimization-O1		optimization-O4	
		Static Instructions	Total Instructions	Static Instructions	Total Instructions
dbc	C	3,006	474,271	2,884	472,365
	C++	5,380	218,062	5,183	216,492
udb1	C	1,879	53,336	1,805	52,951
	C++	3,938	48,754	3,784	47,996
udb2	C	1,950	52,668	1,853	52,258
	C++	4,045	48,694	3,862	47,959
udb3	C	1,954	77,568	1,856	77,423
	C++	4,058	60,156	3,879	59,228
cons	C	1,173	22,454	1,131	22,320
	C++	2,915	26,030	2,845	25,688
intnce	C	1,457	39,315	1,422	39,032
	C++	3,235	34,154	3,235	33,840
mem	C	1,327	39,767	1,307	39,704
	C++	2,238	18,736	2,199	18,511
ovld	C	357	8,067	305	2,718
	C++	357	8,067	305	2,718
ref	C	326	99,390	294	40,332
	C++	326	99,390	294	40,332
unn	C	350	22,808	310	3,703
	C++	350	22,808	310	3,703

Table 1: Information about static and dynamic code sizes on the MIPS machine. Total instructions indicate the number of dynamic instructions executed during each program run. The programs were compiled by g++, using O1 or O4 option as indicated.

6 Performance comparison of C and C++ programs

The initial few subsections in this section provide information on the instruction mix of the programs. The latter few subsections provide information on the cache performance. First, we present instruction counts, number of loads, stores and branches (static and dynamic) to compare the programs. Branch instructions are important because they create potential pipeline stalls in pipelined machines. We have compared the statistics generated by the C and C++ programs on two platforms, a DEC station 5000 with a MIPS processor and a Sun Sparc. The programs were compiled using g++ compiler with compiler options set to -O1 and -O4. The statistics generated by compiling using the different options are also compared, this checks whether the same improvement is obtained in the object-oriented programs while going from -O1 level to -O4 level as in non object-oriented programs.

6.1 Instruction Counts

Table 1 shows the static and dynamic instruction counts on the MIPS machine and Table 2 shows the corresponding quantities on the SPARC. Static instructions generated in the C++ versions are more than their C counterparts because they include libraries such as iostream which are bigger than C's stdio libraries. For programs compiled on the MIPS machine, the static instruction count is lower than the corresponding static instruction counts generated on the SUN machine.

Program Name	Program	optimization-O1		optimization-O4	
		Static Instructions	Total Instructions	Static Instructions	Total Instructions
dbc	C	37,098	467,597	36,130	446,578
	C++	48,594	384,834	47,584	356,911
udb1	C	17,282	43,807	17,343	43,480
	C++	21,491	40,750	21,451	39,946
udb2	C	17,296	43,986	17,400	43,608
	C++	21,507	40,879	21,517	40,078
udb3	C	17,295	66,244	17,382	65,834
	C++	21,522	50,448	21,530	49,531
cons	C	10,877	26,087	10,831	25,879
	C++	20,511	24,563	20,444	23,654
intnce	C	11,053	36,671	11,074	36,185
	C++	21,140	27,905	20,950	27,499
mem	C	16,967	29,034	16,925	28,911
	C++	20,719	21,391	20,680	21,155
ovld	C	2,225	10,719	2,169	7,935
	C++	2,225	10,719	2,169	7,935
ref	C	2,188	89,191	2,155	30,529
	C++	2,188	89,191	2,155	30,529
unn	C	2,436	22,392	2,176	3,104
	C++	2,436	22,392	2,176	3,104

Table 2: Information about static and dynamic code sizes on the Sun Sparc. Total instructions indicate the number of dynamic instructions executed during each program run. The programs were compiled by g++, using O1 or O4 option as indicated.

Programs	wrt C O1				wrt C O4
	C O1	C O4	C++ O1	C++ O4	C++ O4
udb1	100	99.28	91.41	89.99	90.64
udb2	100	99.22	92.45	91.06	91.77
udb3	100	99.81	77.55	76.36	76.50
dbc	100	99.60	45.98	45.65	45.83
cons	100	99.40	115.92	94.67	95.24
intnce	100	99.28	86.87	86.07	86.70
mem	100	99.84	47.11	46.55	46.62
ovld	100	33.69	100	33.69	100
ref	100	40.57	100	40.58	100
unn	100	16.24	100	16.24	100

Table 3: Comparison of Dynamic Instructions for the MIPS (Normalized)
Normalizing is done with reference to the number of instructions for the C program compiled with the O1 option. In the last column, we normalize the counts with respect to O4 level of the C version, so that we can compare the relative counts of C and C++ programs in the best optimized option.

Also, the total instruction count for programs compiled on the DEC machine are higher than the corresponding total instruction counts generated on the MIPS machine. This is due to the dynamic linking of libraries on the MIPS machine, as compared to the static linking of libraries on the SUN. Looking at the instructions generated for both the object-oriented and non object-oriented programs, it is observed that the number of instruction generated is lower for the higher optimization in both paradigms. This is seen for both the static and dynamic instructions.

Tables 3 and 4 illustrate the normalized instruction counts. Normalizing is done with reference to the number of instructions for the C program compiled with the O1 option. In the last column, we normalize the instructions with respect to O4 version of C, so that we can compare the relative counts of C and C++ programs in the best optimized option.

The surprising observation here is that in general C++ programs do not execute more instructions than equivalent C programs. The larger static sizes and higher execution times of C++ programs had persuaded many users to believe that the dynamic sizes of C++ programs are larger than their C equivalents. Our studies show that it is not true. Based on our observations regarding the cache behavior of C and C++ programs, we believe that the slower execution of C++ programs is due to their worse cache behavior.

6.2 Loads and Stores

Table 5 shows the frequency of memory operations for the MIPS machine. The percentages of the loads and stores in each program is calculated with reference to the total number of instructions in the respective program. In tables 6 and 7 we compare the number of loads and stores that are normalized to the C program compiled with the -O1 option. This gives more accurate information on the amount of increase or decrease in the instructions while comparing the different programs and the different options for the same program.

From the information in Table 5, it is observed that more loads are executed in C++; the difference is approximately 5%, which is close to that obtained by Calder et al. This is observed

Programs	wrt C O1				wrt C O4
	C O1	C O4	C++ O1	C++ O4	C++ O4
udb1	100	99.25	93.02	91.19	91.87
udb2	100	99.14	92.94	91.12	91.19
udb3	100	99.38	76.15	74.77	75.24
dbc	100	95.52	82.31	76.34	79.92
cons	100	96.54	91.63	88.24	91.40
intnce	100	98.67	76.10	74.99	76.00
mem	100	99.58	73.68	72.86	73.17
ovld	100	74.03	100	74.03	100
ref	100	34.23	100	36.01	105.22
unn	100	13.86	100	13.86	100

Table 4: Comparison of Dynamic Instructions for the Sparc (Normalized)
Normalizing is done with reference to the number of instructions for the C program compiled with the O1 option. In the last column, we normalize the counts with respect to O4 level of the C version, so that we can compare the relative counts of C and C++ programs in the best optimized option.

when we compare the percentage of loads in each program. If we compare the number of loads normalized to the C program compiled with O1 option, it is seen that the number of loads is higher by approximately 6% on the average.

In Table 5, we also see the percentage of instructions that are stores. The stores were observed to be consistently lower for the C++ programs, by approximately 1%. This contradicts with Calder et. al's study, where they stated that C++ programs execute approximately 2% more stores than C programs. This could be because our programs are microbenchmarks and their programs were bigger and constituted more functions and methods, leading to more register saves and restores across function calls. In that case, their observation may be more valid for large C++ programs. For the 'cons' program on MIPS, we also see more stores in C++.

6.3 Static Basic Block and Dynamic Basic Block sizes

Table 8 shows the information about the static and dynamic basic block sizes and the instructions per block in both static and dynamic blocks. Basic block refers to any single-entry, single-exit sequence of straight line instructions. Larger basic blocks offer more opportunity for architecture-specific optimizations, such as instruction scheduling. It is seen from the statistics generated for the different programs that there is a small difference between the basic block sizes of C and C++ programs. The basic block sizes are larger in the object-oriented programs and is in agreement to the results obtained by Calder et. al. [3].

6.4 Branches

Tables 9 and 10 show the number of branches executed by the C and C++ programs normalized with respect to number of branches in the O1 optimization level of the C programs. It is observed that the C++ versions execute fewer branches compared to the C versions.

Program Name	Program	optimization-O1			optimization-O4		
		%loads	%stores	%branches	%loads	%stores	%branches
udb1	C	20.93	13.11	15.71	20.87	13.16	15.76
	C++	25.60	12.87	13.37	25.56	12.71	13.59
udb2	C	20.93	13.08	15.75	20.86	13.14	15.80
	C++	25.61	12.83	13.41	25.57	12.69	13.61
udb3	C	20.94	13.06	15.76	20.80	13.06	15.74
	C++	25.74	12.85	13.35	25.71	12.71	13.56
dbc	C	21.13	13.58	15.93	21.17	13.63	15.99
	C++	25.62	11.85	13.75	25.65	11.81	13.81
cons	C	18.67	12.28	14.80	18.66	12.25	15.78
	C++	25.54	13.81	12.50	31.07	11.97	15.30
intnce	C	20.73	13.29	15.67	20.68	13.35	15.64
	C++	25.74	12.77	13.28	25.76	12.66	13.40
mem	C	20.40	13.58	16.35	20.38	13.55	16.37
	C++	23.33	12.86	12.24	23.33	12.91	12.40
ovld	C	25.36	16.85	1.94	7.65	2.64	5.73
	C++	25.36	16.85	1.94	7.65	2.64	5.73
ref	C	34.58	11.71	5.27	11.04	12.04	12.95
	C++	34.58	11.71	5.27	11.04	12.04	12.95
unn	C	3.92	8.97	19.35	33.83	49.90	4.21
	C++	3.92	8.97	19.35	33.83	49.90	4.21

Table 5: Information about loads, stores and branches on the MIPS. The table contains the percentage of loads, stores and branches for each program.

Programs	Loads					Stores				
	wrt C O1				wrt C O4	wrt C O1				wrt C O4
	C O1	C O4	C++ O1	C++ O4	C++ O4	C O1	C O4	C++ O1	C++ O4	C++ O4
udb1	100	98.98	111.78	109.88	111.01	100	99.68	89.78	87.29	87.56
udb2	100	98.88	113.10	111.22	112.47	100	99.81	90.74	88.34	88.51
udb3	100	99.16	95.33	93.77	94.56	100	99.74	76.30	74.25	74.45
dbc	100	99.76	55.74	55.39	55.52	100	99.96	40.11	39.69	39.71
cons	100	99.33	158.56	157.51	158.56	100	99.13	130.37	129.00	130.13
intnce	100	99.07	107.90	106.96	107.95	100	99.79	83.50	82.03	82.20
mem	100	99.71	53.87	53.24	53.38	100	99.61	44.63	44.25	44.42
ovld	100	13.09	100	13.09	100	100	5.29	100	5.29	100
ref	100	12.95	100	12.95	100	100	41.72	100	41.72	100
unn	100	18.27	100	18.27	100	100	90.23	100	90.23	100

Table 6: Comparison of Loads and Stores for the MIPS (Normalized)
Normalizing is done with reference to the number of loads or stores for the C program compiled with the O1 option. In the last column, we normalize the counts with respect to O4 level of the C version, so that we can compare the relative counts of C and C++ programs in the best optimized option.

Programs	Loads					Stores				
	wrt C O1				wrt C O4	wrt C O1				wrt C O4
	C O1	C O4	C++ O1	C++ O4	C++ O4	C O1	C O4	C++ O1	C++ O4	C++ O4
udb1	100	97.98	115.66	114.31	116.90	100	98.73	71.80	70.31	71.21
udb2	100	97.54	115.48	114.01	116.89	100	98.54	71.95	70.37	71.41
udb3	100	98.24	95.59	94.52	96.21	100	98.97	60.36	59.15	59.77
dbc	100	95.40	103.66	92.14	96.59	100	96.45	51.61	44.69	46.34
cons	100	99.42	156.09	155.57	156.48	100	98.91	86.24	74.85	75.67
intnce	100	99.21	108.42	107.94	108.80	100	98.85	61.82	60.71	61.42
mem	100	99.30	130.49	129.31	130.22	100	99.26	42.54	42.19	42.50
ovld	100	10.47	100	10.47	100	100	34.04	100	34.04	100
ref	100	13.24	100	13.24	100	100	43.15	100	43.15	100
unn	100	7.41	100	7.41	100	100	84.65	100	84.65	100

Table 7: Comparison of Loads and Stores for the Sun SPARC (Normalized)
Normalizing is done with reference to the number of loads or stores for the C program compiled with the O1 option. In the last column, we normalize the counts with respect to O4 level of the C version, so that we can compare the relative counts of C and C++ programs in the best optimized option.

Program	Static (C)		Dynamic(C)		Static(C++)		Dynamic(C++)	
	BB	inst/BB	B.B	inst/BB	BB	inst/BB	B.B	inst/BB
udb1	439	4.1	12,083	4.4	699	5.4	8,406	5.7
udb2	453	4.1	11,950	4.4	720	5.4	8,411	5.7
udb3	454	4.1	17,638	4.4	723	5.4	10,305	5.8
dbc	708	4.0	109,272	4.3	941	5.5	39,724	5.4
cons	267	4.2	4,819	4.6	518	5.5	4,200	6.1
intnce	304	4.7	8,849	4.4	616	5.3	5,868	5.8
mem	328	4.0	9,261	4.3	435	5.1	3,398	5.4
ovld	74	4.1	805	3.4	74	4.1	805	3.4
ref	72	4.1	6,097	6.6	72	4.1	6,097	6.6
unn	68	4.6	211	17.5	68	4.6	211	17.5

Table 8: Information about Basic Blocks. The table shows the information about the number of static and dynamic basic blocks and the instructions per block in both static and dynamic basic blocks.

Programs	wrt C O1				wrt C O4
	C O1	C O4	C++ O1	C++ O4	C++ O4
udb1	100	99.57	77.82	77.82	78.15
udb2	100	99.52	78.73	78.70	79.08
udb3	100	99.69	65.71	65.70	65.90
dbc	100	100	39.68	39.58	39.58
cons	100	105.99	97.86	97.86	92.33
intnce	100	99.04	73.58	73.58	74.29
mem	100	99.98	35.29	35.29	35.29
ovld	100	99.36	100	99.36	100
ref	100	99.98	100	99.98	100
unn	100	3.53	100	3.53	100

Table 9: Comparison of Branches for the MIPS (Normalized)
Normalizing is done with reference to the number of branches for the C program compiled with the O1 option. In the last column, we normalize the counts with respect to O4 level of the C version, so that we can compare the relative counts of C and C++ programs in the best optimized option.

Programs	wrt C O1				wrt C O4
	C O1	C O4	C++ O1	C++ O4	C++ O4
udb1	100	99.72	79.15	78.83	79.05
udb2	100	99.63	79.23	78.90	79.19
udb3	100	99.73	65.41	65.17	65.34
dbc	100	96.18	60.36	54.40	56.55
cons	100	99.74	71.88	71.62	71.80
intnce	100	99.19	62.09	61.97	62.48
mem	100	99.97	52.62	52.59	52.61
ovld	100	60.70	100	60.70	100
ref	100	49.03	100	49.03	100
unn	100	8.86	100	8.86	100

Table 10: Comparison of Branches for the Sun SPARC (Normalized)
Normalizing is done with reference to the number of branches for the C program compiled with the O1 option. In the last column, we normalize the counts with respect to O4 level of the C version, so that we can compare the relative counts of C and C++ programs in the best optimized option.

6.5 Cache Performance

In this section we look at the instruction and data cache performance of the different programs. We compare the *miss rates* and the *traffic ratios* for the caches. The traces are from DEC Station 5000 which has the MIPS processors with DEC Ultrix operating system. We simulated direct mapped caches whose sizes ranged from 1 kbyte to 64 kbytes. Block sizes from 8 bytes to 64 bytes were considered. The simulation was done for split (separate instruction and data cache) as well as unified caches. The miss ratios for the instruction and data cache are presented in tables 11 to 20. The traffic information for the split and unified caches are presented in tables 21 to 28.

Instruction cache The instruction cache miss rate is a measure of the locality of reference of instruction fetches in a program. The miss ratios for the different block sizes and cache sizes are listed in tables 11 to 20. The tables show that the instruction cache miss rate of C++ programs (except three small programs with identical code) is often 80 to 100% higher than that of equivalent C programs. The reason is that the object-oriented programs do not exhibit significant locality of reference; a method in an object could invoke another method in another object, and both methods may be mapped to conflicting cache blocks. The high miss rates in instruction caches is also due to the large number of function calls in C++. The smaller dynamic function size of C++ programs is another cause for high instruction cache miss rates. Programs executing a small number of instructions in each function, suffer more from instruction cache conflicts. If two functions are aligned to memory addresses that get mapped to the same cache block, they will constantly displace each other from the cache.

For the same cache sizes we see that there is a large difference in the miss rates between C and C++ programs. Increasing the cache size helps to a certain extent, but as we see that after a certain cache size the miss rates remain constant for many of the programs (diminishing returns).

Data cache The miss rates for data caches are presented in tables 11 to 20. For the data cache also, we see that the miss rate is more for C++ programs compared to the C programs. The data cache miss ratios are 80 to 100% higher for C++ programs than equivalent C programs except in the case of three small programs with identical code. The difference is thus almost as significant as that for the instruction cache. In the study conducted by Calder et al. [3] they expected higher data miss rates, but the results that they obtained did not confirm it. The C++ programs are expected to have higher data cache miss rates as there are more heap allocated data structures in C++ than in C.

Traffic Ratios

Comparing the traffic ratios that are given in tables 21 to 28, for two of the larger and two of the smaller programs, the cache traffic ratios are significantly worse for C++ programs compared to C (often more than 100%) and the difference is more pronounced at higher block sizes. Comparing the split and unified caches, for small cache sizes, there is a difference of approximately 15% for programs compiled with the O1 option and 20% when compiled with the O4 option (the unified cache results in more traffic than split caches). The difference in the traffic between split and unified caches reduces with increase in cache size.

udb1 C-program				
Instruction				
Cache	Block size (bytes)			
size	8	16	32	64
1K	19.51	14.00	9.12	8.56
2K	5.53	3.60	2.45	1.86
4K	3.96	2.52	1.68	1.27
8K	2.78	1.77	1.11	0.80
16K	2.06	1.21	0.75	0.50
64K	1.83	1.03	0.59	0.34
Data				
Cache	Block size (bytes)			
size	8	16	32	64
1K	03.87	05.09	5.64	6.11
2K	3.71	4.89	5.30	5.74
4K	2.29	1.99	2.10	2.93
8K	1.79	1.21	0.83	0.60
32K	1.79	1.19	0.80	0.58
64K	1.68	1.09	0.72	0.50

udb1 C++-program				
Instruction				
Cache	Block size (bytes)			
size	8	16	32	64
1K	30.82	18.10	10.96	7.33
2K	24.25	14.33	8.53	5.60
4K	17.85	10.48	6.48	4.31
8K	11.24	6.59	4.09	2.68
16K	7.26	4.21	2.54	1.62
32K	4.43	2.47	1.40	0.88
64K	4.18	2.29	1.27	0.73
Data				
Cache	Block size (bytes)			
size	8	16	32	64
1K	08.59	08.49	7.54	10.25
2K	8.32	8.21	7.28	10.06
4K	6.87	6.57	5.90	8.81
8K	3.44	2.89	2.72	3.16
16K	2.27	1.61	1.20	0.98
32K	2.20	1.56	1.14	0.89
64K	2.18	1.52	1.10	0.85

Table 11: Miss ratios for Instruction and Data caches (udb1)

udb2 C-program				
Instruction				
Cache size	Block size (bytes)			
	8	16	32	64
1K	19.49	14.01	09.10	6.66
2K	5.45	3.53	2.40	1.89
4K	3.95	2.50	1.66	1.26
8K	2.75	1.73	1.08	0.76
16K	2.08	1.21	0.73	0.48
32K	1.90	1.06	0.61	0.36
64K	1.90	1.06	0.61	0.36
Data				
Cache size	Block size (bytes)			
	8	16	32	64
1K	6.57	10.50	12.27	12.82
2K	6.07	9.95	11.54	12.04
4K	4.52	6.11	8.10	9.13
8K	2.17	1.74	1.79	2.50
16K	1.70	1.11	0.74	0.51
32K	1.68	1.09	0.72	0.49
64K	1.68	1.09	0.72	0.49

udb2 C++-program				
Instruction				
Cache size	Block size (bytes)			
	8	16	32	64
1K	30.70	18.03	10.93	7.32
2K	24.17	14.27	8.80	5.65
4K	17.82	10.46	6.58	4.36
8K	11.21	6.56	4.04	2.69
16K	7.29	4.22	2.54	1.66
32K	4.51	2.51	1.46	0.88
64K	4.26	2.33	1.30	0.75
Data				
Cache size	Block size (bytes)			
	8	16	32	64
1K	08.71	08.74	7.89	10.87
2K	8.48	8.47	7.65	10.65
4K	7.02	6.83	6.28	9.47
8K	3.39	2.87	2.71	3.14
16K	2.22	1.57	1.18	0.98
32K	2.16	1.53	1.12	0.89
64K	2.13	1.49	1.08	0.85

Table 12: Miss ratios for Instruction and Data caches (udb2)

udb3 C-program				
Instruction				
Cache size	Block size (bytes)			
	8	16	32	64
1K	19.30	13.79	9.05	6.68
2K	5.18	3.40	2.35	1.86
4K	3.61	2.34	1.57	1.21
8K	2.24	1.46	0.92	0.66
16K	1.46	0.86	0.52	0.36
32K	1.28	0.72	0.41	0.24
64K	1.28	0.72	0.41	0.24
Data				
Cache size	Block size (bytes)			
	8	16	32	64
1K	04.66	06.20	8.95	9.62
2K	4.42	5.93	8.48	9.10
4K	2.72	2.68	3.75	4.87
8K	1.90	1.70	2.01	3.05
16K	1.33	0.90	0.64	0.47
32K	1.33	0.90	0.64	0.47
64K	1.18	0.76	0.50	0.34

udb3 C++-program				
Instruction				
Cache size	Block size (bytes)			
	8	16	32	64
1K	30.88	18.11	10.98	7.32
2K	24.44	14.45	8.92	5.72
4K	17.87	10.51	6.64	4.40
8K	11.14	6.56	4.07	2.70
16K	6.89	4.03	2.45	1.62
32K	3.75	2.11	1.24	0.75
64K	3.46	1.89	1.06	0.61
Data				
Cache size	Block size (bytes)			
	8	16	32	64
1K	05.39	05.41	05.01	6.10
2K	5.07	4.99	4.66	5.81
4K	3.79	3.27	3.19	4.46
8K	3.14	2.63	2.47	2.87
16K	1.86	1.30	0.97	0.81
32K	1.79	1.26	0.91	0.72
64K	1.77	1.23	0.88	0.69

Table 13: Miss ratios for Instruction and Data caches (udb3)

dbc C-program				
Instruction				
Cache size	Block size (bytes)			
	8	16	32	64
1K	11.75	7.79	4.46	7.20
2K	4.41	2.80	1.77	1.33
4K	2.46	1.54	1.06	0.83
8K	1.21	0.75	0.50	0.35
16K	0.71	0.43	0.27	0.18
32K	0.31	0.17	0.10	0.05
64K	0.31	0.17	0.10	0.05
Data				
Cache size	Block size (bytes)			
	8	16	32	64
1K	01.82	2.01	2.09	3.37
2K	1.49	1.61	1.68	2.86
4K	0.86	0.65	0.60	0.85
8K	0.77	0.55	0.42	0.38
16K	0.56	0.39	0.26	0.19
32K	0.42	0.28	0.20	0.12
64K	0.29	0.20	0.15	0.08

dbc C++-program				
Instruction				
Cache size	Block size (bytes)			
	8	16	32	64
1K	32.88	19.77	11.99	7.87
2K	25.55	15.64	9.82	6.62
4K	12.45	7.75	5.22	3.61
8K	7.60	4.49	2.90	1.90
16K	3.24	1.94	1.30	0.86
32K	1.71	1.00	0.60	0.38
64K	1.37	0.76	0.45	0.26
Data				
Cache size	Block size (bytes)			
	8	16	32	64
1K	08.93	07.54	06.43	6.56
2K	8.78	7.40	6.26	6.33
4K	6.96	6.00	5.17	4.44
8K	6.58	5.63	4.74	3.99
16K	6.33	5.46	4.54	3.70
32K	6.27	5.42	4.51	3.67
64K	0.56	0.37	0.27	0.20

Table 14: Miss ratios for Instruction and Data caches (dbc)

cons C-program				
Instruction				
Cache size	Block size (bytes)			
	8	16	32	64
1K	17.62	14.29	9.29	6.62
2K	8.06	4.88	3.12	2.07
4K	5.71	3.48	2.20	1.49
8K	3.36	1.95	1.12	0.76
16K	2.71	1.52	0.87	0.49
32K	2.71	1.52	0.87	0.49
64K	2.71	1.52	0.87	0.49
Data				
Cache size	Block size (bytes)			
	8	16	32	64
1K	03.98	03.45	3.33	8.72
2K	3.36	2.46	2.25	6.01
4K	3.20	2.26	1.83	5.35
8K	3.14	2.19	1.75	5.27
16K	2.01	1.20	0.77	0.54
32K	2.01	1.20	0.77	0.54
64K	2.01	1.20	0.77	0.54

cons C++-program				
Instruction				
Cache size	Block size (bytes)			
	8	16	32	64
1K	38.66	22.06	13.26	8.30
2K	35.80	20.37	12.18	7.68
4K	30.02	17.48	10.60	6.65
8K	24.60	14.46	8.90	5.63
16K	15.15	9.04	5.52	3.54
32K	6.14	3.43	2.04	1.25
64K	5.86	3.23	1.81	1.04
Data				
Cache size	Block size (bytes)			
	8	16	32	64
1K	07.25	06.06	05.15	7.34
2K	5.27	4.53	3.94	5.00
4K	4.51	3.72	3.18	4.34
8K	3.04	2.27	2.01	2.69
16K	2.46	1.80	1.41	1.15
32K	2.46	1.80	1.41	1.15
64K	2.35	1.68	1.27	1.01

Table 15: Miss ratios for Instruction and Data caches (cons)

intnce C-program				
Instruction				
Cache size	Block size (bytes)			
	8	16	32	64
1K	16.42	14.62	9.41	8.40
2K	4.89	3.01	2.09	1.69
4K	3.59	2.19	1.50	1.18
8K	2.31	1.30	0.76	0.50
16K	1.94	1.07	0.61	0.36
32K	1.94	1.07	0.61	0.36
64K	1.94	1.07	0.61	0.36
Data				
Cache size	Block size (bytes)			
	8	16	32	64
1K	02.93	02.69	3.02	4.45
2K	2.22	2.43	3.89	2.26
4K	2.26	1.79	1.49	2.12
8K	2.24	1.78	1.47	2.09
32K	1.29	0.79	0.48	0.39
64K	1.29	0.79	0.48	0.30

intnce C++-program				
Instruction				
Cache size	Block size (bytes)			
	8	16	32	64
1K	33.21	19.77	12.04	7.90
2K	24.60	14.80	9.10	6.04
4K	17.60	10.53	6.60	4.48
8K	12.53	7.36	4.50	2.99
16K	7.88	4.51	2.68	1.73
32K	5.34	2.95	1.68	0.99
64K	5.04	2.75	1.54	0.89
Data				
Cache size	Block size (bytes)			
	8	16	32	64
1K	07.80	06.14	05.23	7.27
2K	7.68	6.00	5.07	7.03
4K	6.14	4.88	4.06	5.51
8K	2.61	2.07	1.64	2.08
16K	2.32	1.62	1.22	0.98
32K	2.32	1.62	1.22	0.98
64K	2.25	1.53	1.12	0.85

Table 16: Miss ratios for Instruction and Data caches (intnce)

mem C-program				
Instruction				
Cache size	Block size (bytes)			
	8	16	32	64
1K	18.17	11.81	8.03	5.85
2K	12.36	8.04	5.20	3.62
4K	7.24	4.97	3.22	2.33
8K	4.30	2.78	1.72	1.35
16K	2.16	1.31	0.77	0.55
32K	1.73	0.97	0.55	0.32
64K	1.73	0.97	0.55	0.32
Data				
Cache size	Block size (bytes)			
	8	16	32	64
1K	06.29	6.20	8.58	10.26
2K	5.98	5.76	7.16	8.82
4K	4.81	4.62	5.42	6.95
8K	1.88	1.63	1.59	1.26
16K	1.75	1.50	1.33	0.77
32K	1.75	1.50	1.32	0.74
64K	1.75	1.50	1.32	0.74

mem C++-program				
Instruction				
Cache size	Block size (bytes)			
	8	16	32	64
1K	38.06	22.16	13.31	8.64
2K	34.36	20.29	12.44	7.96
4K	25.56	15.98	9.79	6.32
8K	15.31	9.21	5.78	3.79
16K	12.32	9.22	4.28	2.74
32K	6.19	3.40	1.89	1.12
64K	6.18	3.39	1.88	1.11
Data				
Cache size	Block size (bytes)			
	8	16	32	64
1K	11.15	9.66	10.51	10.25
2K	8.24	7.36	8.59	7.71
4K	7.25	6.51	8.02	7.11
8K	4.45	3.55	3.14	1.92
16K	4.36	3.47	3.01	1.79
32K	4.36	3.47	3.01	1.79
64K	4.24	3.33	2.73	1.51

Table 17: Miss ratios for Instruction and Data caches (mem)

ovld C-program				
Instruction				
Cache	Block size (bytes)			
size	8	16	32	64
1K	6.14	3.53	2.17	1.36
2K	5.96	3.31	1.91	1.07
4K	5.96	3.31	1.91	1.07
8K	5.96	3.31	1.91	1.07
16K	5.96	3.31	1.91	1.07
32K	5.96	3.31	1.91	1.07
64K	5.96	3.31	1.91	1.07
Data				
Cache	Block size (bytes)			
size	8	16	32	64
1K	9.71	6.47	5.00	3.82
2K	9.71	6.47	5.00	3.82
4K	9.71	6.47	5.00	3.82
8K	9.71	6.47	5.00	3.82
16K	9.71	6.47	5.00	3.82
32K	9.71	6.47	5.00	3.82
64K	9.71	6.47	5.00	3.82

ovld C++-program				
Instruction				
Cache	Block size (bytes)			
size	8	16	32	64
1K	6.14	3.53	2.17	1.36
2K	5.96	3.31	1.91	1.07
4K	5.96	3.31	1.91	1.07
8K	5.96	3.31	1.91	1.07
16K	5.96	3.31	1.91	1.07
32K	5.96	3.31	1.91	1.07
64K	5.96	3.31	1.91	1.07
Data				
Cache	Block size (bytes)			
size	8	16	32	64
1K	9.71	6.47	5.00	3.82
2K	9.71	6.47	5.00	3.82
4K	9.71	6.47	5.00	3.82
8K	9.71	6.47	5.00	3.82
16K	9.71	6.47	5.00	3.82
32K	9.71	6.47	5.00	3.82
64K	9.71	6.47	5.00	3.82

Table 18: Miss ratios for Instruction and Data caches (ovld)

ref C-program				
Instruction				
Cache size	Block size (bytes)			
	8	16	32	64
1K	0.40	0.23	0.14	0.09
2K	0.39	0.22	0.12	0.07
4K	0.39	0.22	0.12	0.07
8K	0.39	0.22	0.12	0.07
16K	0.39	0.22	0.12	0.07
32K	0.39	0.22	0.12	0.07
64K	0.39	0.22	0.12	0.07
Data				
Cache size	Block size (bytes)			
	8	16	32	64
1K	0.35	0.24	0.19	0.17
2K	0.35	0.24	0.19	0.17
4K	0.35	0.24	0.19	0.17
8K	0.34	0.20	0.15	0.12
16K	0.34	0.20	0.15	0.12
32K	0.34	0.20	0.15	0.12
64K	0.34	0.20	0.15	0.12

ref C++ program				
Instruction				
Cache size	Block size (bytes)			
	8	16	32	64
1K	0.40	0.23	0.14	0.09
2K	0.39	0.22	0.12	0.07
4K	0.39	0.22	0.12	0.07
8K	0.39	0.22	0.12	0.07
16K	0.39	0.22	0.12	0.07
32K	0.39	0.22	0.12	0.07
64K	0.39	0.22	0.12	0.07
Data				
Cache size	Block size (bytes)			
	8	16	32	64
1K	0.35	0.24	0.19	0.17
2K	0.35	0.24	0.19	0.17
4K	0.35	0.24	0.19	0.17
8K	0.34	0.20	0.15	0.12
16K	0.34	0.20	0.15	0.12
32K	0.34	0.20	0.15	0.12
64K	0.34	0.20	0.15	0.12

Table 19: Miss ratios for Instruction and Data caches (ref)

unn C-program				
Instruction				
Cache	Block size (bytes)			
size	8	16	32	64
1K	0.40	0.23	0.14	0.09
2K	0.39	0.22	0.12	0.07
4K	0.39	0.22	0.12	0.07
8K	0.39	0.22	0.12	0.07
16K	0.39	0.22	0.12	0.07
32K	0.39	0.22	0.12	0.07
64K	0.39	0.22	0.12	0.07
Data				
Cache	Block size (bytes)			
size	8	16	32	64
1K	0.35	0.24	0.19	0.17
2K	0.35	0.24	0.19	0.17
4K	0.35	0.24	0.19	0.17
8K	0.34	0.20	0.15	0.12
16K	0.34	0.20	0.15	0.12
32K	0.34	0.20	0.15	0.12
64K	0.34	0.20	0.15	0.12

unn C++-program				
Instruction				
Cache	Block size (bytes)			
size	8	16	32	64
1K	0.40	0.23	0.14	0.09
2K	0.39	0.22	0.12	0.07
4K	0.39	0.22	0.12	0.07
8K	0.39	0.22	0.12	0.07
16K	0.39	0.22	0.12	0.07
32K	0.39	0.22	0.12	0.07
64K	0.39	0.22	0.12	0.07
Data				
Cache	Block size (bytes)			
size	8	16	32	64
1K	0.35	0.24	0.19	0.17
2K	0.35	0.24	0.19	0.17
4K	0.35	0.24	0.19	0.17
8K	0.34	0.20	0.15	0.12
16K	0.34	0.20	0.15	0.12
32K	0.34	0.20	0.15	0.12
64K	0.34	0.20	0.15	0.12

Table 20: Miss ratios for Instruction and Data caches (unn)

Program - udb1								
Cache size	Program - C				Program - C++			
	Block size (bytes)							
	8	16	32	64	8	16	32	64
1K	43.11	76.75	124.08	338.26	71.84	93.42	138.25	238.39
2K	17.79	37.36	68.21	196.82	52.47	70.24	98.38	170.82
4K	13.54	28.39	50.64	165.17	40.79	55.11	80.06	145.70
8K	7.32	10.99	17.38	34.29	26.11	34.05	49.24	87.12
16K	4.96	6.36	10.99	13.07	17.72	22.22	30.01	45.80
32K	4.40	5.38	6.97	9.47	11.26	13.90	18.71	29.68
64K	4.33	5.21	6.56	8.66	9.06	10.92	13.63	18.15

Table 21: Comparison of Traffic ratio for Unified Cache configuration.

Program - udb1								
Cache size	Program - C				Program - C++			
	Block size (bytes)							
	8	16	32	64	8	16	32	64
1K	32.53	51.01	74.71	144.54	52.28	67.01	89.42	150.79
2K	11.55	19.66	33.93	62.42	42.56	55.64	74.48	129.45
4K	7.84	10.80	16.84	33.70	32.21	42.10	58.53	106.87
8K	6.06	8.41	13.07	27.30	19.56	24.60	33.93	54.07
16K	4.60	5.68	7.30	10.03	12.72	15.39	19.57	26.69
32K	4.26	5.11	6.29	8.03	8.57	10.28	12.79	17.62
64K	4.21	5.00	6.11	7.69	8.18	9.65	11.86	15.55

Table 22: Comparison of Traffic ratio for Split Cache configuration. (udb2)

7 Summary and Conclusion

In this paper we compare C and C++ programs to investigate whether there is any loss of performance for the increased benefits of object-oriented programming. Calder et al. [3] had performed a similar study on DEC Alpha. We performed the study on MIPS and SPARC platforms and compare our results with Calder et. al.'s [3] and provide additional information such as traffic ratios of caches.

The following are the major observations from the study.

1. The number of static instructions is larger in the case of C++ programs than in the C programs.

2. The total dynamic instructions were more in the C programs than C++ programs for most of the programs that were studied.

3. The number of basic blocks were seen to be more in the case of object-oriented programs compared to the non object-oriented programs.

4. When we examine the percentage of loads, stores and branches in the C and C++ programs it is observed that the number of loads are more in C++ than in the C programs, but at the same time the branches and stores are lower for the C++ programs.

Program - dbc								
Cache size	Program - C				Program - C++			
	Block size (bytes)							
	8	16	32	64	8	16	32	64
1K	30.04	47.41	75.61	223.08	73.57	95.79	130.56	208.26
2K	12.69	21.16	37.71	130.27	51.14	67.28	92.16	143.64
4K	7.01	11.43	20.78	84.57	29.24	40.10	58.31	90.05
8K	3.18	4.31	6.26	53.11	19.83	26.62	39.63	62.25
16K	1.96	2.71	3.88	48.81	11.40	16.53	25.52	41.41
32K	1.11	1.53	2.21	3.13	8.81	13.31	20.81	34.42
64K	0.98	1.35	1.93	2.63	2.89	3.44	4.36	6.26

Table 23: Comparison of Traffic ratio for Unified Cache configuration. (dbc)

Program - dbc								
Cache size	Program - C				Program - C++			
	Block size (bytes)							
	8	16	32	64	8	16	32	64
1K	18.93	26.29	33.20	107.68	55.47	70.17	91.18	137.08
2K	7.79	10.90	16.09	35.30	44.69	57.98	78.09	121.37
4K	4.38	5.70	8.29	15.25	24.33	32.95	48.12	73.30
8K	2.48	3.22	4.46	6.87	16.95	22.82	33.16	50.36
16K	1.55	2.00	2.58	3.49	10.36	15.10	23.21	36.58
32K	0.82	1.01	1.29	1.53	8.08	12.28	19.07	30.83
64K	0.72	0.89	1.14	1.27	2.51	2.93	3.66	4.66

Table 24: Comparison of Traffic ratio for Split Cache configuration. (dbc)

Program - cons								
Cache size	Program - C				Program - C++			
	Block size (bytes)							
	8	16	32	64	8	16	32	64
1K	45.03	77.11	144.53	321.08	81.55	104.81	151.73	157.81
2K	28.64	45.19	97.60	239.77	74.69	94.28	128.39	209.75
4K	22.61	36.40	76.55	211.46	63.58	79.73	106.39	172.76
8K	9.29	12.27	17.33	47.31	45.61	57.85	77.83	129.28
16K	5.96	6.87	8.32	10.40	14.25	18.15	24.68	37.58
32K	5.96	6.87	8.32	10.40	14.25	18.15	24.68	37.58
64K	5.96	6.87	8.32	10.40	12.45	14.78	19.81	28.69

Table 25: Comparison of Traffic ratio for Unified Cache configuration. (cons)

Program - cons								
Cache size	Program - C				Program - C++			
	Block size (bytes)							
	8	16	32	64	8	16	32	64
1K	30.05	48.81	66.77	134.04	62.64	74.86	96.01	149.32
2K	15.02	18.75	25.84	59.85	56.39	67.15	85.21	125.71
4K	11.29	14.15	19.00	48.95	47.34	57.31	73.12	108.76
8K	7.65	9.39	12.21	39.64	38.30	45.59	58.97	84.97
16K	5.87	6.68	7.91	9.53	24.27	29.67	37.62	50.52
32K	5.87	6.68	7.91	9.53	11.36	13.60	17.67	24.32
64K	5.87	6.68	7.91	9.53	10.83	12.75	15.75	20.66

Table 26: Comparison of Traffic ratio for Split Cache configuration. (cons)

Program - intnce								
Cache size	Program - C				Program - C++			
	Block size (bytes)							
	8	16	32	64	8	16	32	64
1K	45.73	81.03	140.20	320.76	75.26	95.92	134.47	236.91
2K	24.30	33.10	51.27	107.03	57.47	73.84	102.33	185.74
4K	19.82	26.39	37.64	76.97	35.54	45.29	63.29	118.90
8K	16.38	21.02	26.86	55.63	24.54	30.31	40.54	66.71
16K	14.84	18.36	22.08	41.34	16.47	20.34	27.51	40.82
32K	14.84	18.36	22.08	41.34	11.96	14.63	19.23	27.05
64K	14.82	18.34	21.93	41.04	10.39	12.48	16.31	22.95

Table 27: Comparison of Traffic ratio for Unified Cache configuration. (intnce)

Program - intnce								
Cache size	Program - C				Program - C++			
	Block size (bytes)							
	8	16	32	64	8	16	32	64
1K	26.82	47.78	65.52	127.49	55.16	68.48	88.82	140.73
2K	9.32	12.45	19.97	43.82	42.64	53.91	71.32	177.77
4K	7.16	9.39	13.64	27.16	30.97	39.39	53.22	89.73
8K	5.25	6.72	9.16	18.93	20.61	25.27	32.47	50.21
16K	3.96	4.52	5.29	6.24	13.63	10.25	20.44	28.11
32K	3.96	4.52	5.29	6.24	9.97	11.74	14.69	19.57
64K	3.96	4.52	5.29	6.24	9.47	10.98	13.39	17.35

Table 28: Comparison of Traffic ratio for Split Cache configuration. (intnce)

5. The cache miss ratios are often 80% to 100% higher for the object-oriented programs in the case of both instructions as well as data.

6. The cache traffic ratio was higher for C++ programs than C programs.

7. C++ features such as overloading, free unions and ease of reference were not seen to cause any runtime overhead.

The high instruction cache miss ratio of object oriented programs is in general attributable to lack of locality arising from methods in an object invoking other methods in other objects, the large number of function calls in C++, smaller dynamic function size of C++ programs, etc. The higher data cache miss ratio is attributable to the large number of heap allocated data structures in C++. Although the above features are expected to result in higher cache miss ratios, we have not quantified the impact of each of the above factors.

We believe that the slower execution time of C++ programs arises mainly from the cache misses. Surprisingly there is no dynamic code increase for C++ programs and branches are fewer. We believe that cache performance can be improved by compiler and hardware optimizations. Once cache performance is improved, we believe that C++ programs will have comparable or better execution speeds as equivalent C programs, especially because there is no dynamic code expansion and there are fewer branches. This is only a preliminary study and we are in the process of performing more detailed analysis which hopefully will lead to better hardware and compiler optimizations.

Note: The test programs used in this study may be viewed at

http://www.ece.utexas.edu/projects/ece/hpcl/oop.html

References

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

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

[3] B. Calder, D. Grunwald, and B. Zorn, "Quantifying Behavioral Differences Between C and C++ Programs", Journal of Programming Languages, Vol. 2, Num 4, 1994.

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

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

[6] Ted Lewis, "If Java is the Answer, What was the Question?", Binary critic, IEEE Computer, Page 133, March 1997.

[7] R. K. Karne,"Object-oriented Computer Architectures for New Generations of Applications", Computer Architecture News, Vol. 23, No. 5, December 1995, pp. 8-19.

[8] L. John and R. Radhakrishnan, "c_ICE: A Compiler-based Instruction Cache Exclusion Scheme", Workshop on Interaction between Compilers and Computer Architectures-1997.

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

[10] B. Muller, "Is Object-Oriented Programming Structured Programming?", ACM SIGPLAN Notices, Vol. 28, Number 9, September 1993, pp. 57-66.

[11] P. Pfeiffer, "Report on the Second Annual Alan J. Perlis Symposium on Programming Languages", ACM SIGPLAN Notices, Vol. 28, Number 9, September 1993, pp. 6-12.

[12] 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.

[13] R. Bielak, "Object Oriented Programming: The Fundamentals", ACM SIGPLAN Notices, Vol. 28, Number 9, September 1993, pp. 13-14.

[14] P. Wang, C++ Object Oriented Programming,

[15] WARTS: The Wisconsin Architectural Research Tools Set, University of Wisconsin, Madison.

[16] P. Wegner, "The Object-Oriented Classification Paradigm", In Bruce Shriver and Peter Wegner, editors, Research Directions in Object-Oriented Programming, MIT Press, Cambridge, MA, 1991.

[17] Rebecca J. Wirts-Brock and Ralph E. Johnson, "Surveying current Research in Object-Oriented Design", Communications of the ACM, 33(9), 1990.