Understanding the Branch Performance of Object Oriented Workloads

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

Abstract

Object oriented programming (OOP) methodology has gained widespread acceptance despite concerns on slow execution. Object oriented languages typically use indirect function calls to implement polymorphism, and there have been concerns about the branch predictability of object oriented programs. Our measurements on the UltraSPARC-II using the on-chip performance monitoring counters of the SPARC chip indicated that C++ programs exhibit a worse CPI than similar C programs and that they encounter more branch misprediction stalls. It was also observed that there is a strong correlation between the CPI of the C++ programs and the branch misprediction stalls encountered by these programs. This paper is an attempt to correlate some of the innate features of C++ programs with their branch behavior. We analyze the branch behavior of several large C++ applications and C programs from the SPEC95 suite. Deeper understanding of the inherent branch behavior of object oriented programs will enable the design of effective branch predictors for them. This is particularly important given the popularity of object oriented programming and its advantages from the software engineering standpoint.

Keywords: Branch Prediction, Branch Behavior, Machine Measurement, Object Oriented Programming, Performance Monitoring, Workload Characterization.

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

1 Introduction

Object oriented programming introduces the notion of objects, classes, encapsulation, inheritance, polymorphism, templates, etc into programming in an attempt to solve some of the most difficult problems facing software development. The improvement in code reusabilty, code maintainability and increased levels of abstraction with use of object oriented strategies has made object oriented programming paradigm very popular in the software development community, despite concerns about the slow execution of these programs.

The primary reasons generally attributed to the slowness of object oriented programs have been poor instruction cache performance and branch behavior [4] [9]. Polymorphism, a feature provided by object oriented languages, is implemented through a virtual function interface that 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 the appropriate function is invoked at runtime. It has been known that virtual functions and dynamic dispatch are expensive [1] [9] [17]. However there have also been studies that indicate that the branch execution penalty for C++ programs are not worse than those for C programs [7].

The objectives of this paper are to analyze the branch behavior of object oriented programs and identify the impact of this branch behavior on execution time. The execution of C++ and C programs on the Sun UltraSPARC platform are profiled and analyzed. 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.

Past research on related topics include efforts to characterize OOP instruction mix [4], reduce indirect function calls [5], optimize virtual function calls [1], improve branch prediction [6] [7], analyze the dispatch overhead of the standard virtual function table dispatch [9], etc. Most of this research was based on simulations and executable inspection whereas we also have performance results based on on-chip performance monitoring counters. We make an effort to relate profiling data, measured execution behavior based on on-chip performance monitoring counters and results from branch prediction simulators. Our emphasis is on understanding the predictability of branches in object oriented programs, the consistency of branch targets, and other features that affect branch performance. Are branches in object oriented programs less predictable? Are branch targets in object oriented programs less consistent? Do object oriented programs need different kinds of branch predictors? We investigate these issues based on the execution of several C and C++ programs on the Sun UltraSPARC.

1.1 Overview

In Section 2, we describe the benchmarks we used for our study and their characteristics based on program analysis. In section 3, we present observations based on measurement on an actual platform, the UltraSPARC-II architecture. In section 4, we present an analysis of several different branch characteristics including predictability of branches, consistency of branch targets, etc. Simulations with a variety of branch predictors are presented. Section 5 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 several 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 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."

Program Name	Instruction Count Static	Dynamic	Basic Block Size	Taken Basic Block Size
deltablue	30541	40774931	8.02	11.97
eqn	43871	2853819	5.16	8.05
groff	36955	142627451	na	na
idl	84557	82898216	5.91	10.88
ixx	72744	30666373	6.47	10.58
richards	22026	6612956	5.37	10.12
go	96984	522932433	8.26	13.98
ijpeg	83079	1426133278	12.39	15.92
li	58114	167929839	6.67	14.13
m88ksim	53631	139464386	6.84	11.35
vortex	161821	2372772250	5.72	11.10

Table 1: Summary of Benchmark Characteristics

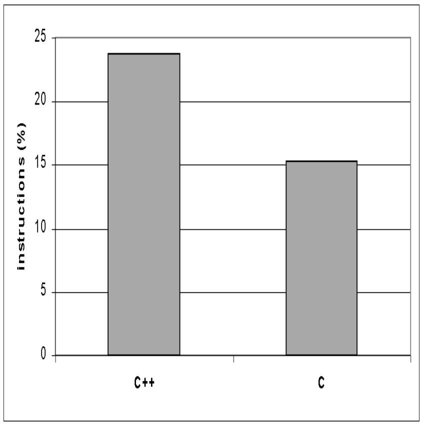
Transactions to and from the database are translated though a schema. The schema as provided with the benchmark is pre-configured to manipulate three different databases: mailing list, parts list, and geometric data. The benchmark builds and manipulates three separate, but inter-related databases based on the schema. The workload of VORTEx has been modeled after common object-oriented database benchmarks.

Table 1 gives information about the static and dynamic code size and basic block size for the different 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. 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. It may be observed that the basic block sizes of the C programs are slightly higher than those of the C++ programs. The effect is more pronounced in the taken basic block size indicating shorter uninterrupted instruction sequences in C++.

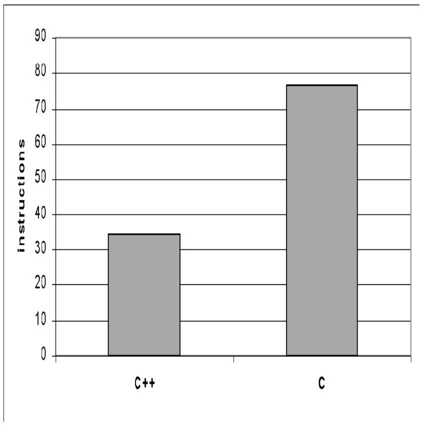
Table 2 lists the branch related statistics of the benchmarks. We see that indirect calls are more frequent in C++ programs than C programs. Only two of the C programs *li* and *ijpeg* show significant indirect calls. Table 2 also shows what percentage of all calls are indirect calls. The average number of instructions per function invocation in the C programs is twice the corresponding quantity in the C++ programs. 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. The frequency of control transfer instructions (branches, jumps and calls) are also shown in Table 2. The C++ programs contain a higher percentage of control transfer instructions than the C programs (21.3% versus 14.5%). One C program, *ijpeg* has a very high number of instructions per function call. This is a program that performs manipulation of image files. There is a significant amount of data parallelism and there are very few control flow interruptions as indicated by the low percentage of control transfer instructions. Figure 1 illustrates the harmonic mean of the frequency of the control transfer instructions in each suite.

Program Name	Control Transfer Instructions (Dynamic) %Branches	%Calls	%Jump	Total	Inst/Call	Indirect Calls % of all Calls
deltablue	12.46	2.49	9.08	24.03	19.23	67.14
eqn	19.18	0.98	1.19	21.35	207.15	2.99
groff	18.70	1.78	1.91	22.39	88.90	NA
idl	16.92	2.58	6.93	26.43	23.67	48.30
ixx	15.46	3.12	4.36	22.94	37.38	19.72
richards	18.60	2.65	4.63	25.88	27.44	27.32
go	13.26	1.14	1.15	15.55	92.59	-
ijpeg	8.07	0.14	0.20	8.41	658.40	5.03
li	17.72	1.46	2.75	21.93	76.68	0.06
m88ksim	15.20	1.97	2.04	19.21	52.82	-
vortex	17.48	2.28	2.27	22.03	47.56	-
C++-mean	16.50	1.95	2.90	23.69	34.57	11.46
C-mean	13.20	0.51	0.69	15.32	76.54	0.11

Table 2: Control Transfer Related Characteristics for C++ and C programs



% Control Transfer Instructions



Instructions per Function Call

Figure 1: Summary of control transfer related statistics for C and C++ Programs. The harmonic mean for each suite is presented.

Program Name	CPI	Cache miss rates IC_miss	DC_miss	EC_miss	Percent of Cycles stalled due to: 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 3: Performance of Large Benchmarks on UltraSPARC-II (measured using on-chip counters)

3 Measured Execution Behavior

In this section we present information gained by actual measurement of branch performance using the on-chip counters on the UltraSparc chip. We used a package called *perf-monitor* [18] developed at the Swedish Institute of Computer Sciences. The programs were compiled using the GNU gcc compiler (version 2.7.2), with optimization enabled *(-O2 -msupersparc)*. Events measured using the counters include total number of cycles, instruction count, instruction and data cache misses, and processor stall cycles due to branch mispredicts and a variety of other factors. We obtain these statistics and analyze the correlation between CPI and stalls due to branch mispredicts.

The statistics obtained using *perf-monitor* while running the programs are shown in Table 3. In general, the 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 *eqn* benchmark has a high number of instructions per call resulting in 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.

Table 3 also presents L1 instruction, L1 data and external(L2) cache miss ratios and the percentage of processor cycles stalled due to instruction cache misses, branch mispredictions, store buffer full conditions, and data load waiting. Generally speaking, the C++ programs encounter higher instruction cache misses and higher L2 cache misses. 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 stalls due to branch mispredictions are significantly higher for C++ programs than C programs. 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.

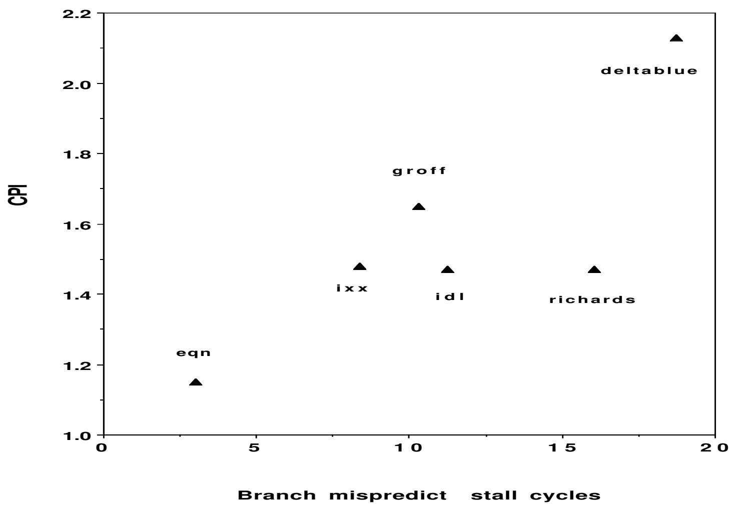


Figure 2: Correlation between CPI and branch mispredictions for C++ programs

Figure 2 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.

4 Branch Analysis

One aspect we were interested in analyzing was the consistency of the branch target. Although it can be expected that C++ programs may have more indirect branches which potentially could be to multiple targets, no quantification of the consistency of branch targets has been made in past studies. We perform an analysis of the number of targets each specific branch instruction has and the results are presented in Table 4. The column labeled *static* indicates what percentage of all static branch instructions have more than one target. The column labeled *dynamic* indicates how many of all dynamic branches were from branches with multiple targets. It may be observed that the C++ programs contain a significantly larger percent of multi-target branches (dynamic). This also suggests that it will be difficult to correctly predict the target for many C++ branch instructions.

How predictable are branches in C++? Although measurement of performance counters yield valuable information on stalls due to branch mispredictions, measurement is always limited to the specific configuration of the machine. The UltraSPARC-II has a dynamic branch prediction scheme based on a two-bit history of the branch. We investigated the performance of five different branch predictors, ranging from simple backward taken forward not taken scheme to a correlated branch predictor. The five different schemes are listed in Table 5. Branch direction prediction accuracies of the five different schemes are presented in Table 6. We could not observe any significant difference in behavior between branch predictor accuracy of C and C++ programs. However the prediction rates in Table 6 reflect only correctness of the direction of the branch (taken/no taken), not correctness of the target. We supplemented each of the aforementioned schemes with a BTB of 2048 entries. Table 7 illustrates the prediction rates considering the correctness of the target also.

Program Name	%Branches with 2 or more targets	
	Static	Dynamic
deltablue	8.49	52.57
eqn	9.06	1.94
idl	7.44	27.15
ixx	11.56	16.62
richards	6.34	11.53
go	4.26	8.81
ijpeg	4.95	1.72
li	7.29	19.73
m88ksim	5.17	11.36
vortex	6.56	11.54

Table 4: Percentage of control instructions that have multiple targets

Name	Description
BTFNT	Backward Taken Forward Not Taken
1 BHT PC	1 Branch History Table indexed by the PC of the branch instruction. There are 2048 entries in the table
Basic 2 Level	Two level correlation predictor with 3 bits of correlation. Each of the 8 branch history tables has 256 entries (or a total of 2048 entries).
Gshare PC	5 bits of global history are used to be XORed with the PC of the branch to index the 2048-entry branch history table

Table 5: Branch Prediction Techniques Simulated

Initially we were surprised to observe that the prediction rates for C++ programs are not worse than the C programs, however, we also noticed that the total number of static branches in our C++ programs were often less than 2048. Simulations with smaller BTBs are currently being performed to identify the correlation between the higher percentage of multi-target branches and the branch performance. We expect to present those results in the final version of the paper.

5 Summary and Conclusion

In this paper we analyze the branch performance of several large C++ and C programs on the UltraSPARC-II. Profiling of branch instruction mix was performed using *spixtools*. Then 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.

The major observations from profiling and simulations were

- The C++ programs contain approximately 23.69% transfer of control instructions, as opposed to 15.32% in the C programs.
- The basic block size of C++ applications is typically smaller than that of the C programs.

Program	Branch Prediction Accuracy			
Name	BTFNT	1 BHT	Basic 2 level	Gshare
deltablue	67.05	91.72	96.57	97.43
eqn	64.03	97.13	97.27	97.30
idl	54.30	96.61	96.99	96.90
ixx	61.15	92.70	93.38	93.48
richards	53.10	70.67	82.10	85.18
go	59.18	74.50	75.59	74.75
ijpeg	77.80	91.55	91.86	92.01
li	47.21	88.20	91.01	91.61
m88ksim	60.26	94.47	96.27	96.26
vortex	51.53	97.57	96.71	95.24

Table 6: Branch direction prediction accuracies for a few different schemes

Program	Branch Prediction Accuracy			
Name	BTFNT	1 BHT	Basic 2 level	Gshare
deltablue	64.72	89.66	94.42	95.27
eqn	63.76	96.94	97.08	97.10
idl	53.63	96.06	96.38	96.29
ixx	59.64	91.87	92.44	92.54
richards	53.04	70.62	82.04	85.12
go	55.69	72.20	73.12	72.30
ijpeg	77.50	91.27	91.58	91.73
li	46.43	87.45	90.25	90.85
m88ksim	58.65	93.20	94.95	94.87
vortex	51.53	97.50	96.71	95.24

Table 7: Branch target prediction accuracies for a few different schemes

- C++ programs contain a higher amount of multi-target branches potentially degrading the performance of branch target buffers, however from the limited simulations we performed, significant differences could not be observed in branch prediction rates.

The major results from measurement were

- The C++ programs exhibited a worse CPI than the C programs (1.57 versus 1.17). This result is based on measurement using performance counters.
- 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.

This is only a preliminary study into characterizing branch performance. More validation of these observations need to be performed. Our objective is to analyze several more properties of workloads related to branch performance such as the extent of correlation that exists between branches, average number of unique paths to a branch, etc. We plan to conduct such studies for a variety of workloads and intend to find trends in the observed properties.

Detailed characterization of modern workloads is required to precisely analyze the correlation of program features 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.

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.