Accurately Modeling Speculative Instruction Fetching in Trace-Driven Simulation

Ravi Bhargava, Lizy K. John, Francisco Matus *
Electrical and Computer Engineering Department
The University of Texas at Austin
{ravib,ljohn,matus}@ece.utexas.edu

Abstract

Performance evaluation of modern, highly speculative, out-of-order microprocessors and the corresponding production of detailed, valid, accurate results have become serious challenges. A popular evaluation methodology is trace-driven simulation which provides the advantage of a highly portable simulator that is independent of the constraints of the trace generation system. While developing and maintaining a trace-driven simulator is relatively easier than other alternatives, a primary drawback is the inability to accurately simulate speculative instruction fetching and subsequent execution. Fetching from an incorrect path occurs often in a speculative processor, however it is difficult to capture this information in a trace.

This paper investigates a scheme to accurately model instruction fetching within a trace-driven framework. This is accomplished by recreating an approximate copy of the object code segment, which we call resurrected code, *using a preliminary pass through the trace. We discuss a fast and memory-efficient method for implementing this resurrected code. In addition, we characterize UltraSPARC traces of C, C++, and Fortran programs generated using Shade to determine the potential of this method. Using these traces, and a modest branch predicting scheme, we find that in 14 of 16 cases more than 99% of all branches will find their target instruction in the resurrected code. Furthermore, on these occasions, a large amount of consecutive instructions are available along the mispredicted path. These results indicate that the inaccuracies associated with speculative fetching in trace-driven simulation can be significantly reduced using this resurrected code.*

L. John is supported by the National Science Foundation under Grants CCR-9796098 (CAREER Award), and EIA-9807112, and a grant from the Texas Advanced Technology Program. F. Matus is also with Advanced Micro Devices.

1 Introduction

Accurate simulation of modern microprocessors is an important process in both academic research and the industry [1]. Just as important as the accuracy of the simulation, if not more important, is the time required to produce results from the simulation. This time includes creating the simulation environment, validating the tools used in simulation, producing relevant test cases, and the running of the simulations themselves.

One existing method of simulation is trace-driven simulation where details from the dynamic execution stream of a process or processes are recorded and used to drive a software timing model of a microprocessor. Traces are most commonly produced by software monitoring methods such as trapping or manipulating object code [2] or by hardware monitoring methods.

One motivation for using trace-driven simulation is that many trace-driven simulators and tracing tools already exist. Due to stability and familiarity, these simulators supply a high level of comfort and confidence. While storing traces can be cumbersome and the actual simulation can be relatively slow, the maintenance and development of existing trace-driven simulators has proven to be more simple and convenient than alternatives.

Trace-driven simulation also enjoys the advantages of portability and flexibility, requiring only a trace and some simulation software in its simplest form. While trace generation systems have several constraints such as dependency on operating system, compilers, etc., the generated traces and the simulator can be transported across platforms rather easily. For interpreted languages, like Java, where there is no executable, traces are the most convenient way to capture the instruction stream of the program.

There are several alternatives to traditional trace-driven simulation. Execution driven simulation [3] is a relatively fast technique that executes many of the

instructions on the host machine instead of simulating all of the instructions. Execution-driven simulators [4] [5] take an executable (cross-compiled into a simulated instruction set architecture) as the input and therefore do not require a trace. Each of these methods has its drawbacks, and whether it is availability, portability, accuracy or robustness, there is still a place for trace-driven simulation.

The major drawback of trace-driven simulation is the inability of trace-generation mechanisms to report all of the speculation that takes place in modern superscalar processors. Specifically, speculative instructions are not yet adequately represented in traces. In state-of-the-art processors, when a branch is predicted, the processor starts to fetch instructions from the predicted target address. Many of these instructions are decoded, issued, and even executed, but are not committed until the actual branch target is resolved. If there was a misprediction, these instructions are flushed from the processor (squashing) and are never seen by the tracing tool.

This process of squashing has often been modeled by stalling instruction fetching until the mispredicted branch has been evaluated. Some simulators incorporate a penalty to model the branch misprediction [6]. This ignores the fact that these speculative instructions consume processor resources and affect the future state of the processor. Moudgill et al from IBM quantified the impact of not simulating mispredicted paths on a four-issue processor using programs from the SPEC95 integer benchmark [7]. They found that the variation in instructions completed per cycle is small (in all but one case it is less than 0.5%) and does vary in any one direction. They also found that mispredicted memory references may lead to additional cache hits, acting as a natural type of prefetch mechanism. Although these results are encouraging for users of trace-driven simulators, the growing concern is that increasing instruction widths and degrees of speculations will lead to an unacceptable level of error.

One proposed remedy is to use a trace-driven simulator along with some other technique to acquire the information needed to properly simulate speculative instruction execution. One such method is performed by Reilly and Edmondson with their Alpha Microprocessor simulator [8] in which they use Aint [9] in conjunction with a trace-driven simulator to supply basic blocks of instructions from speculated addresses.

In this paper, we propose stepping through a trace once, strategically storing all the instructions that are accessed in the sequence of their address to create an approximate copy of the source code, which we call the *resurrected code*. The resurrected code is later used in conjunction with the trace in order to perform accurate simulation. We present a method and data structure for creating and reusing the resurrected code efficiently. The analysis of our benchmark programs show that often less than 1% of all branches branch to an instruction that is not in the resurrected code.

The paper is organized in the following manner. Section 2 explains our simulation environment, including our tracing tools and benchmarks. Section 3 describes our method for implementing the proposed technique. Section 4 is an analysis of the effectiveness of the resurrected code structure. Section 5 concludes the paper.

2 Simulation Environment

2.1 Tracing and Profiling Tools

Traces are generated dynamically using the tool Shade [10] on a Sun UltraSPARC-II processor . Shade is a tool that dynamically executes and traces SPARC executables. It is customizable and provides several sections in which the user can specify the exact trace information to collect. At these points, the trace information can be dynamically handled in any manner. Shade only traces user and library code and does not analyze kernel code.

Detailed information can be collected dynamically for every instruction and opcode. We collect data such as the opcode fields (to identify type of branches), program counter, branch targets, and taken/not-taken branch information. This information is then processed by our own software simulation tools.

To collect static information, such as the number of static instructions, we use `spix` and `spixstats` which are part of SpixTools [11]. Spix requires that programs be statically compiled, therefore all of executables that we analyze with SpixTools and Shade are statically compiled. We use `spix` and `spixstats` primarily for cross-checking and validation.

2.2 Benchmarks

We use programs from SPEC CINT95, SPEC CFP95, and a suite of C++ programs. Eight of the SPEC integer programs, CINT95, are used in our study - `compress, gcc, go, ijpeg, li, m88ksim, perl,` and `vortex`. These are the same programs used in Moudgill et al's study [7] of mispredicted path penalty. Five programs from the floating-point SPEC, CFP95, are analyzed for our study - `fpppp, hydro, su2cor, tomcatv,` and `wave5`. These programs are computation-intensive and written in Fortran [12]. (The remaining SPEC CFP95 programs are not included due to time and length considerations.) The final suite of programs are written in

Table 1: Benchmark Descriptions

Program	Input	Description of Program
SPEC CINT95: C programs		
compress95	test.in	Compresses large text files
gcc	amptjp.i	Compiles pre-processed source
go	2stone9.in	Plays the game Go against itself
ijpeg	vigo.ppm	Performs jpeg image compression
li	train.lsp	Lisp interpreter
m88ksim	-c ctl.in	Simulates the Motorola 88100 processor
perl	scrabbl.pl scrabbl.in	Performs text and numeric manipulations
vortex	vortex.in	Builds, manipulates 3 interrelated databases
SPEC CFP95: Fortran Programs		
fpppp	natoms.in	Performs multi-electron derivatives
hydro2d	hydro2d.in	Hydrodynamical Navier Stokes equations
su2cor	su2cor.in	Masses of elementary particles are computed
tomcatv	tomcatv.in	Generation of 2-D coordinate system
wave5	wave5.in	Solve's Maxwell's equations on a cartesian mesh
Suite of C++ Programs		
deltablue	3000	Incremental dataflow constraint solver
eqn	eqn.input.all	Type-setting program for math. equations
idl	all.idl	SunSofts IDL compiler 1.3
ixx	object.h Som_Plus_Fresco.idl	IDL parser generating C++ stubs
richards	1	Operating system simulation benchmark

C++ and have been used to study the behavior of C++, specifically the cost of virtual functions calls [13] [14]. These programs are `deltablue, eqn, idl, ixx,` and `richards`. Short descriptions of these programs are in Table 1.

Table 2 provides a description of the basic characteristics of the benchmarks. Static instructions are acquired from `spixstats` and represent the unique instructions available in the executable. Dynamic instructions represent the number of instructions executed by each program. Long running programs are stopped at one billion instructions. The branch percentage refers to the percentage of dynamically executed instructions that are branches. Unconditional calls, jumps, and branches that break the instruction flow are considered to be branches and are included in our branch analysis.

2.3 Branch Prediction

Our study requires the simulation of dynamic branch prediction hardware in order to identify when misspeculation occurs. We use the Gshare branch prediction scheme as described by McFarling in [15]. The primary predictor is 2048 entries, direct-mapped, and indexed by the program counter plus five global history bits. This predictor is accompanied by a 512-entry, direct-mapped branch target buffer (BTB) to predict target addresses for the predicted branches.

We are most interested in branches that start fetching from the wrong location due to misprediction. There are two cases in which this happens and this is when we report a branch *misprediction*. One case surfaces when Gshare mispredicts the branch. The other occurrence is when the Gshare method correctly predicts taken, but the branch instruction's target address is not correct in the BTB. In all other cases, the prediction is based directly on Gshare.

3 Implementation

The objective of the paper is to present a framework for accurate simulation of instruction fetching in a trace-driven simulator. Figure 1.a illustrates the traditional approach to trace-driven simulation. The simulator receives instructions sequentially from the trace and these instructions fuel the simulation. For branch instructions, a branch predictor mispredict results in the simulator effectively stalls until the branch is resolved. The instruction stream corresponding to the mispredicted paths are not normally available for simulation. The resurrected code proposed in this paper becomes a source for fetching the instructions from the mispredicted paths.

3.1 Data Structure

Ideally, the structure that holds the instruction information should be memory efficient and quickly accessible both while creating and reusing it. Instructions have been shown to have both temporal and spatial locality, so instructions should, in general, be blocked together. On the other hand, one in every

Table 2: Benchmarks Characteristics

Benchmark Program	Static Instructions	% static code visited	Dynamic Instr. (in millions)	% Branches (dynamic)	% branches mispredicted
compress	44,214	6.7	38.8M	11.71	10.29
gcc	403,581	17.0	267M	20.25	14.40
go	96,981	54.6	373M	15.55	26.48
ijpeg	95,673	16.56	1000M	8.36	11.31
li	71,303	10.58	170M	20.98	14.14
m88ksim	71,367	14.46	125M	17.28	9.85
perl	118,799	15.34	41.7M	20.80	15.34
vortex	171,421	9.68	1000M	21.54	9.68
fpppp	91,104	13.06	258M	1.31	8.32
hydro2d	77,799	6.33	1000M	11.87	3.43
su2cor	75.984	15.58	1000M	14.20	8.38
tomcatv	66,901	7.85	1000M	7.95	8.45
wave5	111,645	19.65	1000M	5.12	7.78
deltablue	30,539	15.14	41.1M	23.91	6.84
eqn	43,868	26.00	48.2M	21.65	10.73
idl	84,554	18.47	85.5M	25.65	12.81
ixx	72,741	18.07	30.3M	22.64	11.54
richards	22,023	6.83	6.7M	25.30	20.74

four to six instructions can disrupt the control flow of the program. So it is quite possible that within these blocks there will be small holes where instructions are never reached.

Accounting for the above observations, we implement a dynamic, tree-like structure that is directly indexable, and attempts to minimize the amount of memory allocated to the "holes". The *resurrection tree* is predictably composed of "nodes", where each non-leaf node contains pointers (in C) to more nodes. In addition, each non-leaf node need not contain any other information. Leaf nodes do not allocate memory for the array of node pointers and must contain information about the instructions they are representing.

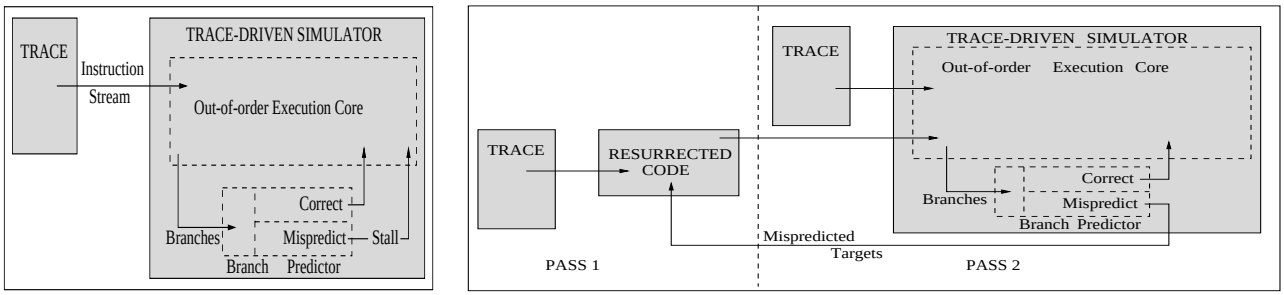
The resurrection tree is designed with a 32-bit, byte-addressable address space and 32-bit wide instruction format in mind. All instructions are 32-bits wide, so the program counter (PC) is always incremented by four bytes. Therefore the last two bits are unimportant and are truncated, leaving 30 bits. It would be convenient for the nodes at each level of the tree to be represented by one structure and therefore have the same number of children. This would require that each level of the tree to potentially have $2^{n \cdot x}$ nodes, where n is the current level of the tree and x is the increase in size (in bits) between levels (i.e. level 0 contains the root and would have $2^{0 \cdot x} = 1$ node).

We choose x to be five so that we have seven levels, 0 - 6, where level 0 is the root node and level 6 contains all of the unique, executed instructions. Notice that nodes at any level of the tree are directly indexable by the program counter and require no searching. Therefore all instructions can be accessed in constant time. The distribution table in Figure 2 shows what percentage of all nodes created are created at each level for the C programs. Approximately 95% of all of the nodes created are leafs which contain the static instructions. This indicates that the structure is not causing an excess of intermediate nodes that do not store instructions and therefore is efficiently using memory.

3.2 Using Resurrected Code

Doing complete speculative simulation using the resurrected code methodology requires two passes through the trace. On the first pass through the trace, each instruction is interpreted and then placed into the resurrection tree on the first instance of the instruction. On subsequent instances of the instruction, the instruction can be ignored. The presence of a branch predictor is not necessary on the initial pass unless trace characterization statistics (such as the ones presented in this paper) are desired.

Upon completion of the first pass, the resurrected code structure is complete and full simulation may take place (illustrated in Figure 1.b). The second pass through the trace progresses in a similar manner to traditional trace-driven simulation except in the case of mispredicted branches. On a mispredicted branch, the resurrected code structure becomes the new instruction source until the branch target is resolved.



(a) Traditional (b) With Resurrected Code

Figure 1: Traditional Use of Trace-Driven Simulator versus Trace-Driven Simulation with Resurrected Code

When this happens, appropriate squashing takes place and the simulator resumes accepting instructions from the trace. On the occasion that an instruction is not present in the resurrected code, it may be necessary to revert to the stalling technique of traditional trace-driven simulation.

The resurrected code may be stored in a file and used in several simulations. This storage can be done in many different manners depending on the goals of the user.

Table 3: Analysis of Resurrected Code

Benchmark Program	% empty targets	% with a branch
compress95	0.025	99.74
gcc	0.952	88.76
go	0.035	99.69
ijpeg	0.042	99.42
li	0.117	98.82
m88ksim	0.703	92.51
perl	0.373	87.45
vortex	1.096	88.14
fpppp	0.091	98.86
hydro2d	0.253	90.35
su2cor	0.002	97.18
tomcatv	1.460	80.73
wave5	0.436	83.10
deltablue	0.124	73.99
eqn	0.471	85.22
idl	0.090	68.38
ixx	0.565	78.86
richards	0.002	99.97

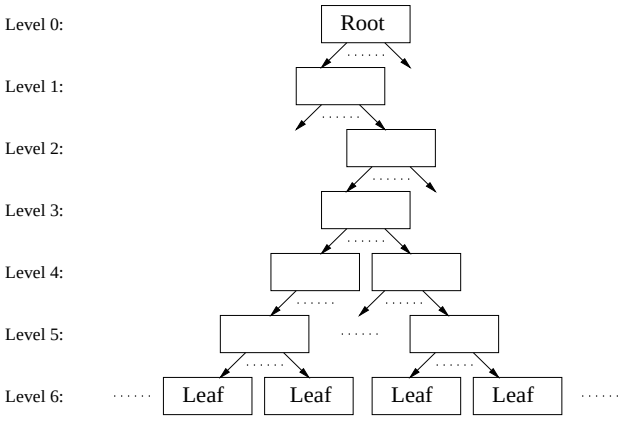
% empty targets is the percentage of all dynamic branch targets that are not found in resurrected code. *% with a branch* is the percentage of the continuous code sequences that follow a mispredicted branch which contain at least one branch.

4 Effectiveness

For this study, the most revealing statistic is the frequency at which speculated branch targets attempt to access instructions that are not in the resurrected code. This information is presented in the first column of Table 3 as the percentage of speculated branch targets that have no corresponding instruction in the resurrected code. This occurs less than 1% of the time in 16 of the 18 applications studied and less than 1.5% in all of the applications.

These percentages reveal the portion of the time that one can not properly model the speculative fetching of a microprocessor. In these few cases, it is necessary to choose an alternative technique until the branch target is resolved. This technique could be a variety of techniques, including stalling until the branch is resolved, random execution, or executing the next available instruction sequence. The significance of these low empty target percentages become apparent by comparing them to the percentage of mispredicted branches found in Table 2. Without resurrected code, those mispredict percentages represent how often this stalling occurs. Notice that the percentage of branches that stall range from 3.43% to 26.48% without the resurrected code, and improve to around 1% with the resurrected code. There is an improvement of one order of magnitude in most cases.

When branch targets are speculated, they begin to fetch starting at a certain instruction address. We refer to the number of consecutive instructions available at any such target as a *continuous code sequence*. Figures 3, 4 and 5 show a breakdown of the continuous code sequence sizes along mispredicted paths for each benchmark. We can see that in general the size of the continuous code sequences are pretty large. Although modern machines do path-based fetching, it is a reassuring to see that for most programs there is a significant sequential stream of code available to fetch if required.



Distribution table of the tree and its nodes for C programs.

	Lvl 1	Lvl 2	Lvl 3	Lvl 4	Lvl 5	Lvl 6
% Nodes	0.0055	0.0063	0.0393	0.4281	4.9427	94.732
% Nodes Full	0.0	0.0	0.0	0.09	6.81	N/A
Avg	1.75	2.0	12.5	135	1570	31,771
Max	3 (vortex)	5 (vortex)	48 (vortex)	427 (vortex)	3964 (gcc)	68,585 (gcc)

Figure 2: Resurrection Tree and Distribution Table

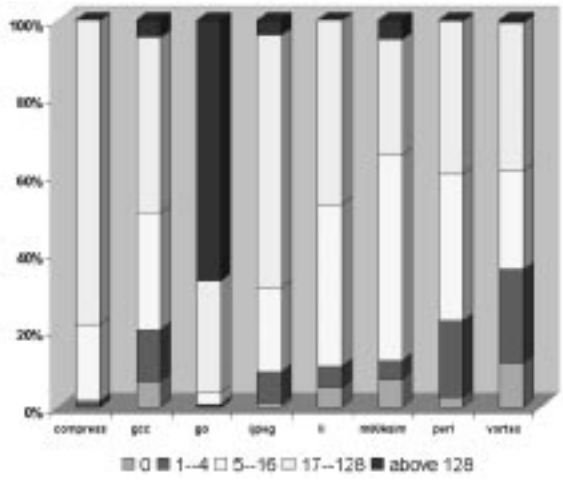


Figure 3: Breakdown of continuous code sequence lengths along mispredicted paths for C Programs

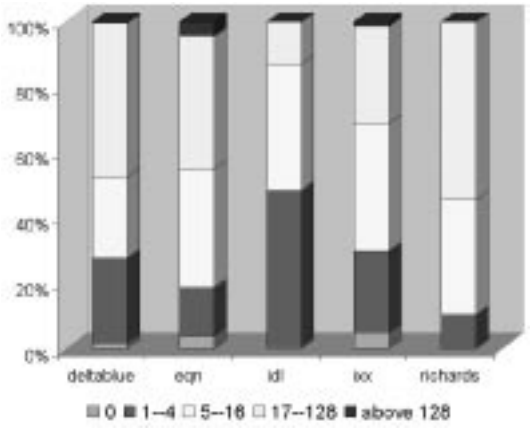


Figure 4: Breakdown of continuous code sequence lengths along mispredicted paths for C++ Programs

On the occasions that the continuous code sequence is zero, the simulator could revert to the stalling technique of traditional trace-driven simulators. We can see that `tomcatv` and `vortex` have the highest percentage of empty branch targets and accordingly the worst accuracy with the resurrected code. It is interesting to note that these two programs also have relatively low branch misprediction rates. In general, we found no correlation between the branch misprediction rate of a program and the potential accuracy increase with resurrected code.

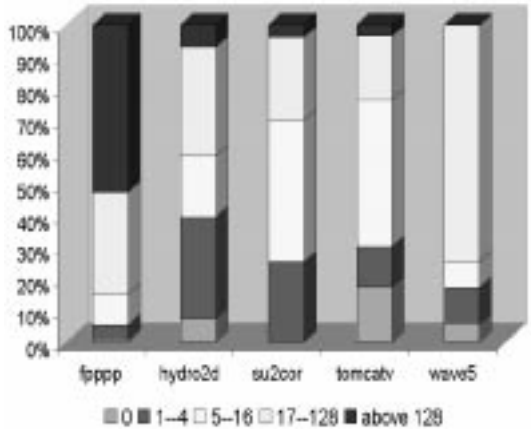


Figure 5: Breakdown of continuous code sequence lengths along mispredicted paths for Fortran Programs

The second column of Table 3 is the percentage of continuous code sequences following a mispredict which contain a branch. This number is significant because once the fetch mechanism reaches a branch while speculatively fetching, it makes another speculative decision based on the branch predictor and follows

that path. We can see that a large percentage of the continuous code sequences in the C and Fortran programs contain at least one branch and in most cases many more.

5 Conclusions

In this study, we investigate a scheme to improve the accuracy of trace-driven simulation of speculative processors. We create a structure called *resurrected code* by taking a preliminary pass through the trace and recreating an approximate copy of the code segment within an executable. During trace-driven simulation, a speculative fetch unit can access the resurrection tree like object code or an instruction cache to obtain instruction opcodes.

We construct the resurrected code such that it is a relatively inexpensive method for significantly reducing the impact of speculative mispredicted paths. With this structure in place, we find that a trace-driven simulator can fetch from the proper branch target for over 99% of all branches in 16 of our 18 benchmarks and more than 98.5% in all benchmarks. Without the proposed structure, traditional trace-driven simulators accurately model speculative instruction fetching only for correct branch predictions, which occur for only 74% to 97% of branches in our benchmarks. This is a significant increase in the ability of trace-driven simulation to mimic the true activity of a speculative processor.

References

[1] P. Bose and T. M. Conte, "Performance Analysis and Its Impact on Design," *IEEE Computer*, pp. 19–22, May 1998.

[2] T. M. Conte and C. E. Gimarc, *Fast Simulation of Computer Architectures*. Kluwer Academic Publishers, 1995.

[3] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair, "The Rice Parallel Processing Testbed," in *1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, (Santa Fe, New Mexico), pp. 4–11, May 1988.

[4] D. Burger, T. Austin, and S. Bennett, "Evaluating Future Microprocessors: The SimpleScalar Tool Set," Tech. Rep. CS-TR-96-1308, University of Wisconsin, Madison, WI, July 1996.

[5] V. S. Pai, P. Ranganathan, and S. V. Adve, "RSIM Reference Manual. Version 1.0.," Tech. Rep. 9705, Department of Electrical and Computer Engineering, Rice University, July 1997.

[6] B. Black and J. P. Shen, "Calibration of Microprocessor Performance Models," *IEEE Computer*, vol. 31, pp. 59–65, May 1998.

[7] M. Moudgill, J. Wellman, and J. E. Moreno, "An approach to quantifying the impact of not simulating mispredicted branches," *Workshop Digest of the PAID Workshop held in conjuction with ISCA98*, pp. 60–66, July 1998.

[8] M. Reilly and J. Edmondson, "Performance Simulation of an Alpha Microprocessor," *IEEE Computer*, pp. 50–58, May 1997.

[9] A. Paithankar, "AINT: A Tool for Simulation of Shared-Memory Multiprocessors," Master's thesis, University of Colorado, Boulder, Colo., 1996.

[10] R. F. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," Tech. Rep. SMLI 93-12 and UWCSE 93-06-06, Sun Microsystems Laboratories, Incorporated, and the University of Washington, 1993.

[11] B. Cmelik, "SpixTools Introduction and User's Manual," Tech. Rep. SMLI TR-93-6, Sun Microsystems Laboratory, Mountain View, CA, Feb 1993.

[12] Standard Performance Evaluation Corporation, "SPEC CPU95 Benchmark." *http://www.spec.org/osg/cpu95/*.

[13] K. Driesen and U. Holzle, "The Direct Cost of Virtual Function Calls in C++," in *OOPSLA-96*, (San Jose, Calif.), pp. 306–323, Oct 1996.

[14] B. Calder, D. Grunwald, and B. Zorn, "Quantifying Behavioral Differences Between C and C++ Programs," Tech. Rep. CU-CS-698-94, University of Colorado, Boulder, Jan 1994.

[15] S. McFarling, "Combining Branch Predictors," Tech. Rep. TN-36, Digital Western Research Labs, Palo Alto, Calif., Jun 1993.