Time Varying Behavior of Programs

Timothy Sherwood Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{sherwood,calder}@cs.ucsd.edu

UCSD Technical Report
CS99-630, August 1999

Abstract

Modern architecture research relies heavily on detailed pipeline simulation. Furthermore, programs often times exhibit interesting and important time varying behavior on an extremely large scale. Very little analysis has been conducted to classify the time varying behavior of popular benchmarks using detailed simulation for important architecture features.

In this paper we classify the behavior of the SPEC95 benchmark suite over their course of execution correlating the behavior between IPC, branch prediction, value prediction, address prediction, cache performance, and reorder buffer occupancy. Branch prediction, cache performance, value prediction, and address prediction are currently some of the most influential architecture features driving microprocessor research, and we show important interactions and relationships between these features.

In addition, we show that many programs have wildly different behavior during different parts of their execution, which makes the section of the program simulated of great importance to the relevance and correctness of a study. We show that the large scale behavior of the programs is cyclic in nature, point out the length of cyclic behavior for these programs, and suggest where to simulate to achieve results representative of the program as a whole.

1 Introduction

Each year a medley of architecture alternatives for increasing instruction level parallelism (ILP) abounds. Branch prediction, cache performance, value prediction, and address prediction are some of the most influential architecture features driving microprocessor research. In this study we attempt to classify popular benchmarks (SPEC95), showing the time varying behavior and correlation between these architectural features, in order to help guide research in these areas.

A secondary goal of this study is to show which parts of a program to simulate in order to achieve the representative behavior of a program. In order to evaluate new architecture features, detailed modeling of the pipeline, buses, and queuing delays are needed along with timing models and power estimation. Detailed simulations takes a great deal of processing power and time, thus many times only a small subset of a whole program may be sampled. Many programs have wildly different behavior during different parts of their execution making the section of the program's execution simulated of great importance to the relevance and correctness of the study. We examine this and point out where simulation should start and how long the simulation should run to capture the cyclic behavior of the application.

We provide these results for the SPEC95 benchmark suite, since they are the most widely used and available benchmarks for architecture research. Researchers have shown that the SPEC95 programs have several similar program characteristics with NT office applications and other popular programs [7]. Even so, other benchmarks (e.g., C++ and Java programs, databases, games, and office applications) need to be studied and made available using this same analysis. This is especially true for memory research.

The rest of the paper details our results and suggests directions for research and simulation. Section 2 describes the motivation for this study. Section 3 describes the simulator and architecture model used. Section 4 presents the time varying graphs showing the correlation of the different architecture features and performance. Section 5 examines which parts of a program to simulate in order to capture a representative sample of the program. Finally, section 6 summarizes the results and contributions of this work.

2 Motivation

Branch prediction, cache performance, value prediction, and address prediction are currently four of the more influential architecture features guiding architecture research. Little research has been done to show the correlation between these features and IPC performance, which is the goal of our study.

Branch prediction and memory design are well established research areas, since they address the two fundamental bottlenecks: Fetch Bottleneck and Memory Bottleneck. Branch prediction affects instruction supply, and the overall throughput of the processor. Whereas caches are used to try and alleviate the memory bottleneck.

Address prediction, which predicts the addresses for load instructions, has been used to address the memory bottleneck. It can be used to help guide memory disambiguation and to perform memory prefetching [2, 5].

More recently, value prediction has been proposed as an approach to break true data dependency chains by predicting the resulting value for an instruction, and by allowing dependent instructions to use this predicted value as a source value [8, 4]. This allows the dependent instructions to execute in parallel with the long latency instructions, reducing the lengths of the critical paths through a program.

To our knowledge, researchers have not yet looked in detail at the correlation between IPC, branch prediction, cache miss rates, address prediction, and value prediction. It is important to find where there exists correlation, and this study is a first step at trying to find these correlations. These results can then be used by researchers to focus on a subset of the programs and areas in those programs exhibiting positive or negative correlation.

Instruction Cache	32k 2-way set-associative, 32 byte blocks, 1 cycle latency
Data Cache	64k 4-way set-associative, 32 byte blocks, 2 cycle latency
Unified L2 Cache	1Meg 4-way set-associative, 32 byte blocks, 12 cycle latency
Branch Predictor	hybrid - 8-bit gshare w/ 8k 2-bit predictors + a 8k bimodal predictor
Out-of-Order Issue	out-of-order issue of up to 8 operations per cycle, 128 entry re-order buffer
Mechanism	load/store queue, loads may execute when all prior store addresses are known
Architecture Registers	32 integer, 32 floating point
Functional Units	8-integer ALU, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, 2-FP MULT/DIV
Virtual Memory	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 1: Baseline Simulation Model.

Another motivation for this study is to determine where we should perform simulations for our own architecture research. Our results show the variation between the startup of an application, and where and for how long the SPEC programs need to be simulated to achieve a representative sample of the application. This study has proved very useful to us, and we hope that it will help guide other researchers to consistent, meaningful and reproducible results.

3 Methodology

To perform our study, we collected information for all of the SPEC95 benchmarks running their SPEC95 reference input sets. Each program was compiled on a DEC Alpha AXP-21164 processor using the DEC C, C++, and FORTRAN compilers. The programs were built under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`).

The timing simulator used is derived from the SimpleScalar 3.0a tool set [1], a suite of functional and timing simulation tools for the Alpha AXP ISA. The simulator executes only user-level instructions, performing a detailed timing simulation of an aggressive 8-way dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch mis-prediction. The baseline microarchitecture model is detailed in Table 1. We modified the 3.0a release of SimpleScalar, so that the memory hierarchy buses were pipelined, with a transfer width of 8 bytes per cycle.

4 Program Behavior Over Time

To show the time varying behavior of the programs, SimpleScalar was modified to output and clear its statistics after every 100 million *committed* instructions. Only the statistic counters are cleared, the state of the machine (e.g., cache and branch prediction tables) are *not* cleared between intervals.

Results are then graphed for every 100 million committed instructions for the SPEC95 programs. This should yield a clear picture of the large scale runtime behavior exhibited by each application as well as indicating which sets of instructions are more indicative of the execution as a whole. It is however of small enough granularity that it

provides useful information about program start up times and can be easily simulated on any machine. Each program was run until completion, but we only graph enough intervals to show the cyclic nature for each program.

The following summarizes the data graphed:

- **Instructions Per Cycle**.
- **Percent RUU Occupancy**. SimpleScalar uses a unified Register Update Unit (RUU) to model its reorder buffer and reservation stations [13]. In our simulations we used a 128 entry RUU, and report results in terms of the percent of the RUU entries used on average.
- **Cache Miss Rate**. Cache miss rates are shown for a 32 KB 2-way associative instruction cache, and a 64 KB 4-way associative data cache. Both caches have 32 byte lines.
- **Branch Prediction Miss Rate**. We used McFarling's bi-modal gshare branch predictor [9]. An 8K entry 2-bit chooser table is used to choose between an 8K entry 2-bit bi-modal branch predictor and an 8K entry gshare table. A 256 entry 4-way associative branch target buffer is used to provide the predicted address, and a 32 entry return address stack is used to predict return instructions. The branch misprediction rate including all branch instructions is shown.
- **Address Prediction Miss Rate**. Miss rates are shown for 2-delta stride address prediction for an infinite sized table (each load gets its own entry) [5, 11]. The 2-delta address predictor will only change the stride if seen two times in a row. Confidence counters are not used, so miss rates are for predicting all load instructions.
- **Value Prediction Miss Rate**. Miss rates are shown for 2-delta value address prediction for an infinite sized table [4, 14]. The 2-delta value predictor will only change the stride if seen two times in a row. Confidence counters are not used, so miss rates are for predicting all load instructions.

Note, address and value prediction were not *used* in the architecture simulations. We only gathered their miss, along with the program's IPC, RUU occupancy, branch miss rate, and cache miss rates. Only loads are predicted for value and address prediction. We chose to examine stride address and value prediction rather than last value and context prediction because of its increased accuracy over last value prediction and it is less expensive to implement than context prediction [11].

4.1 Result Graphs

Figures 1, 2, 3, 4, 5, and 6 show the time varying performance of the SPEC95 benchmark suite. The legend is at the top of each figure. For each program, the results for IPC, average percent RUU occupancy, branch miss rate, value miss rate, address miss rate, and instruction and data cache miss rates are shown on the same graph. Since all of these different results are shown on the same graph, each graph has two Y-axis.

For each graph, the left and right Y-axis are labeled with the metrics that use that axis. For most of the graphs, percent RUU occupancy, and value and address miss rates use the left Y-axis. Similarly, I-Cache miss rate, branch

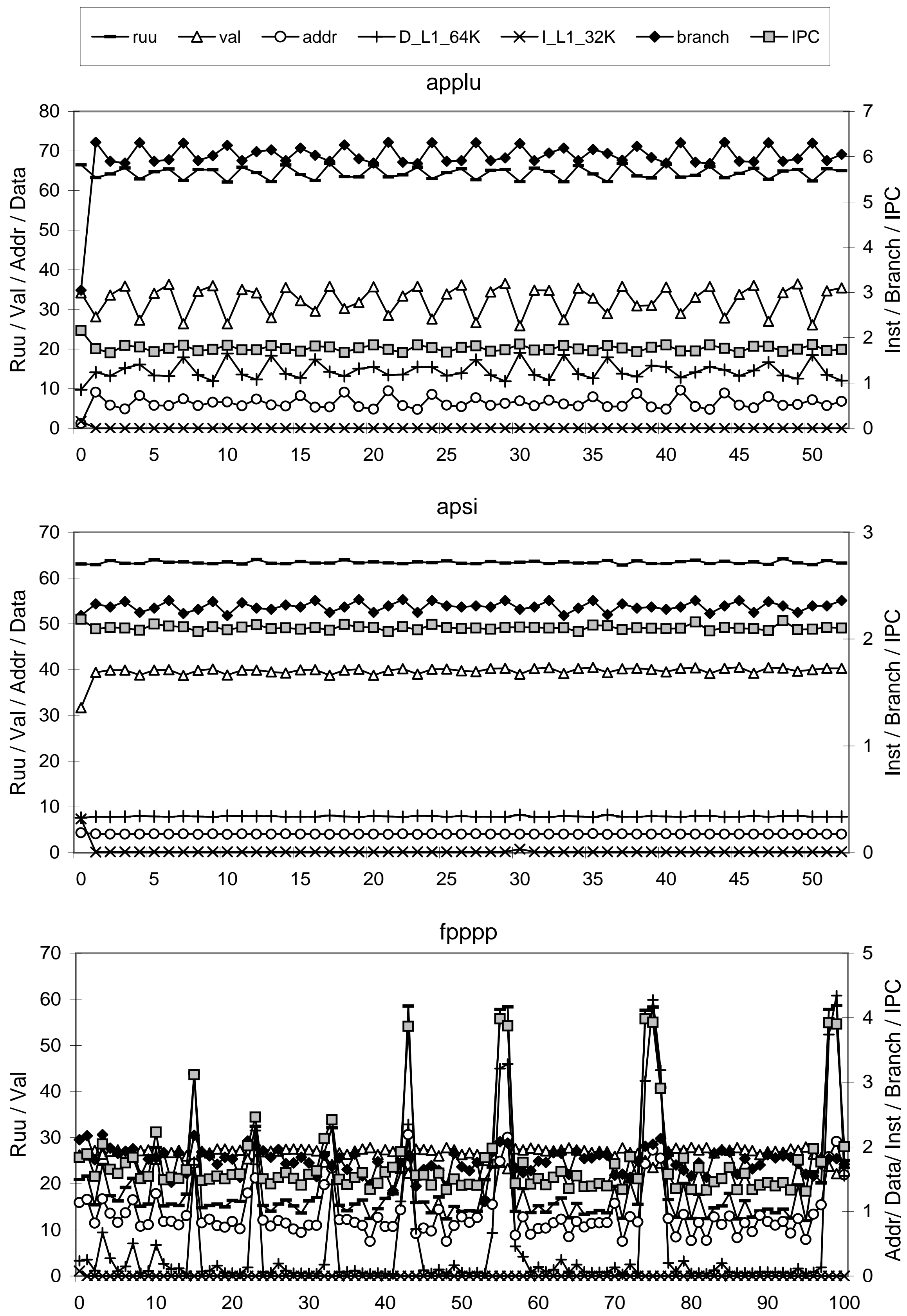

Figure 1: Time varying behavior for applu, apsi, and fpppp. The X-axis is in terms of 100 million committed instructions.

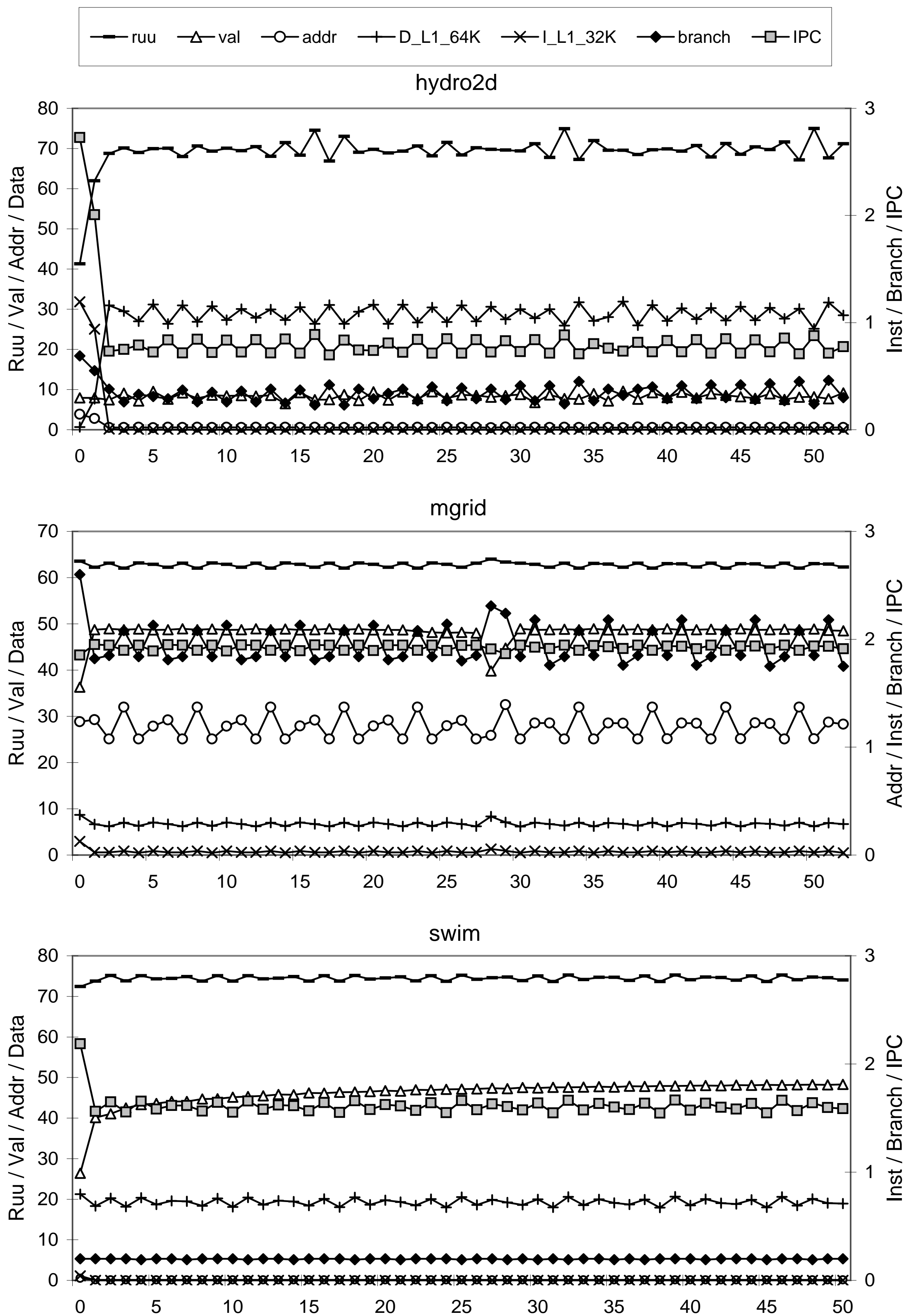

Figure 2: Time varying behavior for hydro2d, mgrid, and swim. The X-axis is in terms of 100 million committed instructions.

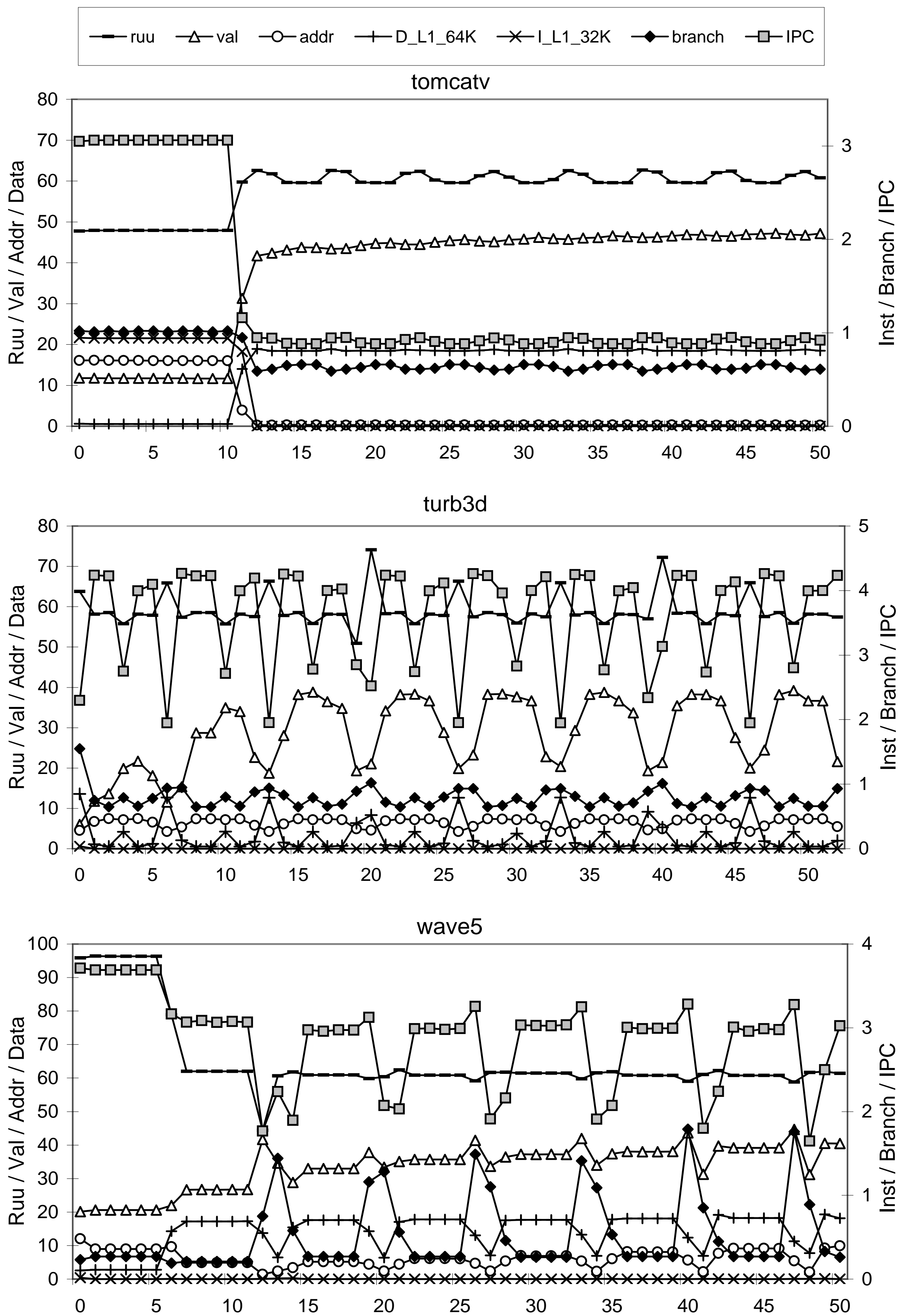

Figure 3: Time varying behavior for tomcatv, turb3d, and wave5. The X-axis is in terms of 100 million committed instructions.

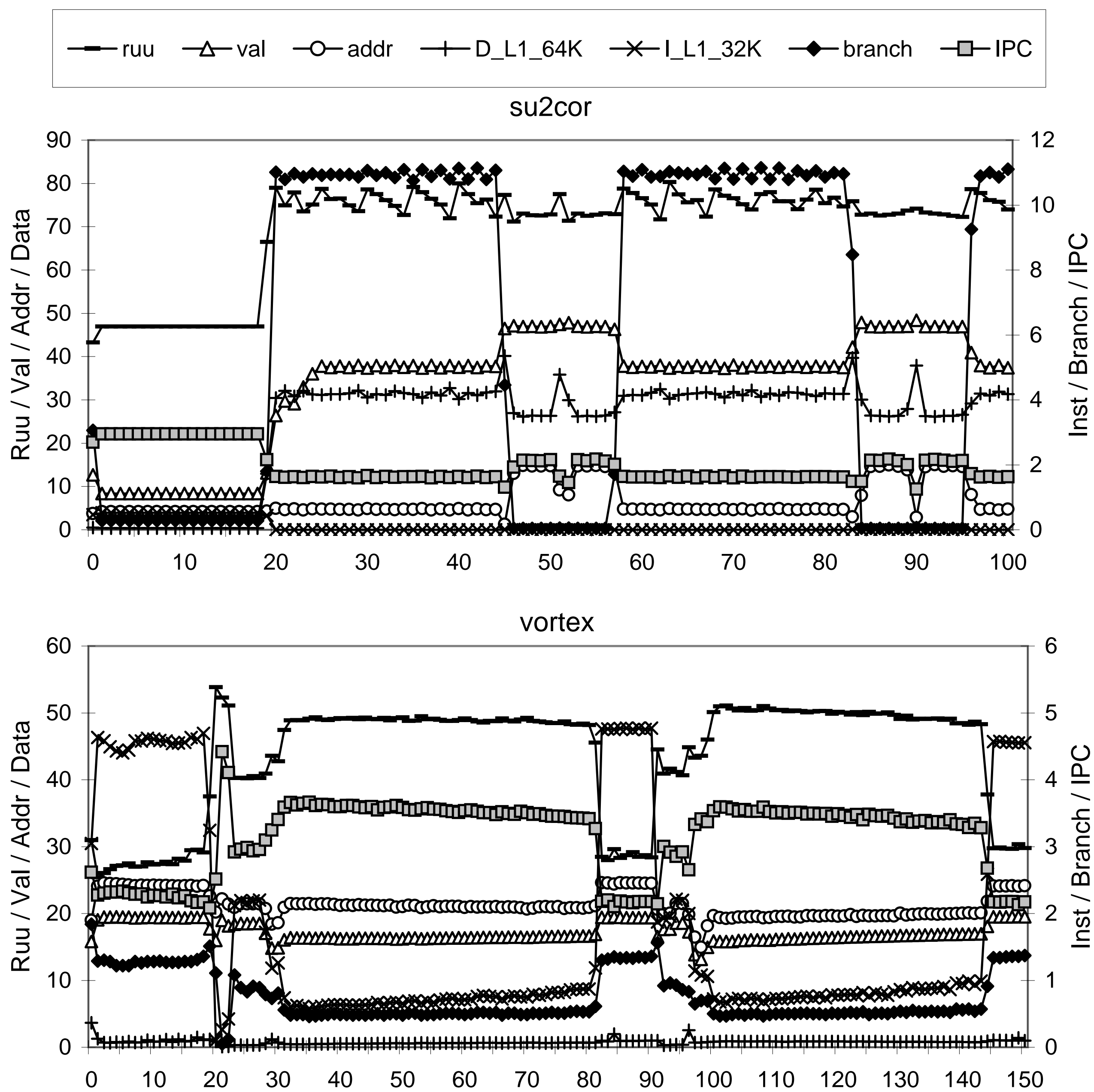

Figure 4: Time varying behavior for su2cor and vortex. The X-axis is in terms of 100 million committed instructions.

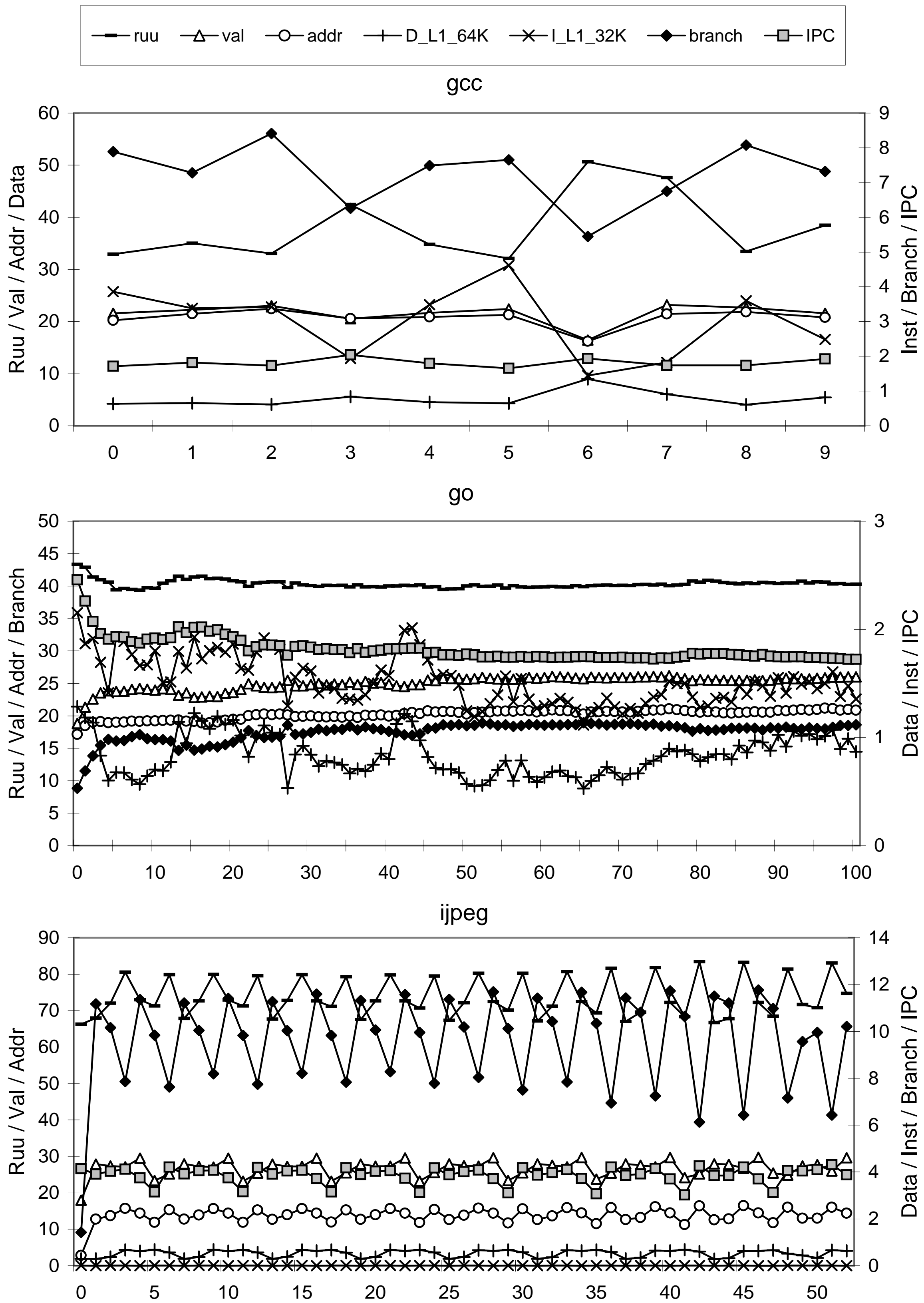

Figure 5: Time varying behavior for gcc, go, and ijpeg. The X-axis is in terms of 100 million committed instructions.

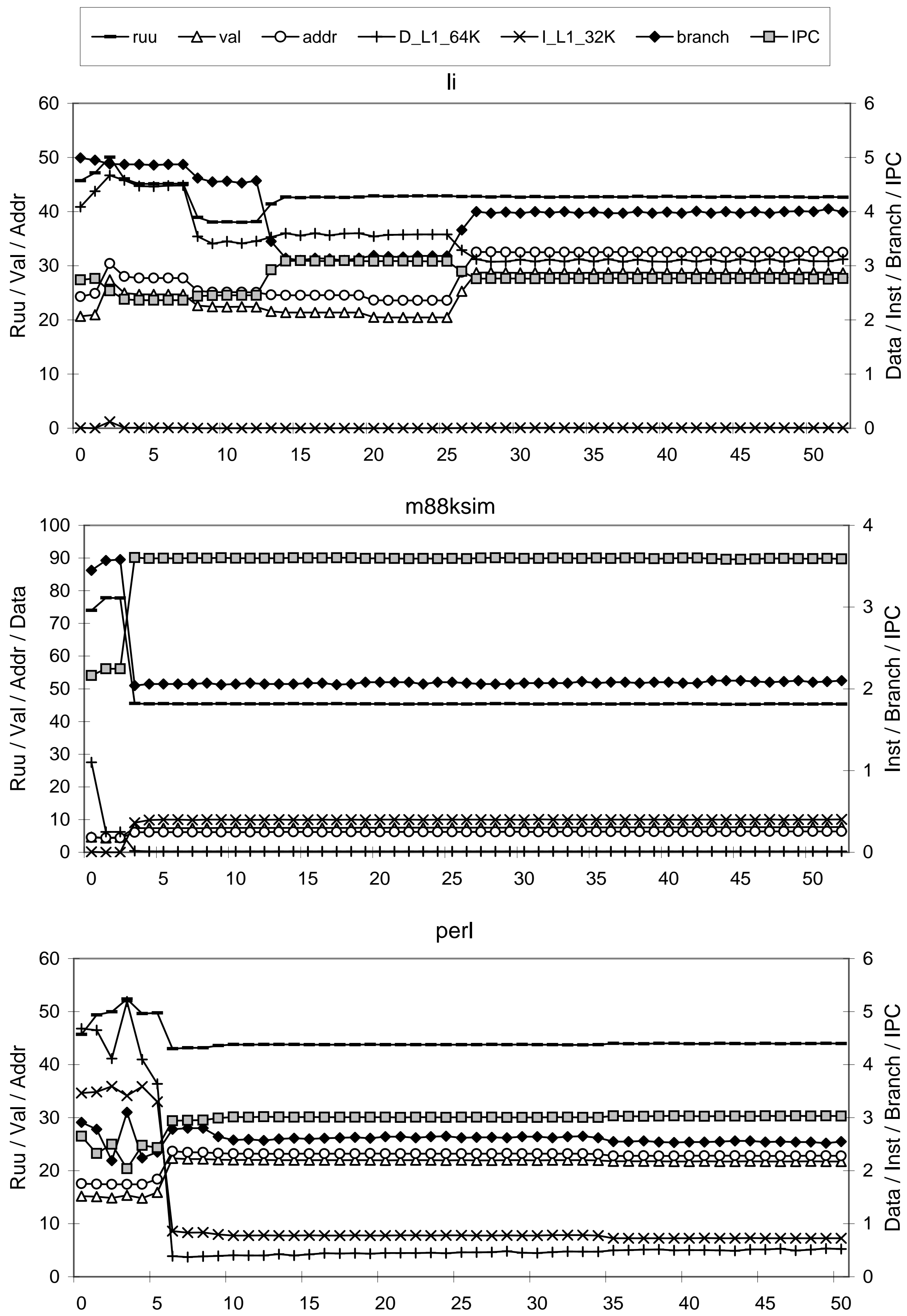

Figure 6: Time varying behavior for li, m88ksim, and perl. The X-axis is in terms of 100 million committed instructions.

miss rate, and IPC usually use the right Y-axis. The D-Cache miss rate is shown on either axis depending upon the program and axis scale in order to allow interesting trends to be seen. Therefore, the Y-axis label shows which metrics use that axis.

The X-axis is in terms of 100 million *committed* instructions. We ran all of the programs to completion, and found them to either (1) converge to a constant behavior until the last few 100 million instructions, or (2) have a repeatable cyclic behavior until the end of their execution. The only program that did not fit into this category was `gcc`, because of its different compilation phases and the ref input we chose (`1cp-decl`) ran for 1 billion instructions. Therefore, the results for most programs are shown for only 50 intervals (5 billion instructions), or until completion. We show more intervals for `vortex`, `su2cor`, `go`, and `fpppp` to capture their cyclic behavior.

Table 2 provides a correlation summary between the different architecture features, for the SPEC95 programs that had cyclic behavior. For programs with cyclic behavior, this table shows the positive and negative correlation between IPC, RUU occupancy, branch prediction, address prediction, value prediction, and data cache miss rates. A + means that the two metrics both increase or decrease together. A — means that when one metric increases the other decreases, and visa versa. A 0 means that there is basically no correlation visible. N/A means that one or both of the metrics never really varied during execution.

The results for `turb3d`, `wave5`, `su2cor`, and `vortex` in table 2 and figures 3 and 4 show interesting correlations between the different architecture features examined.

For `turb3d`, the dips in IPC are correlated to the increase (spikes) in data cache miss rate. The graphs also show that when the data cache miss rates spike, the value prediction miss rate of loads decreases significantly. This suggests that value prediction may helpful at alleviating these dips in IPC.

In `wave5`, the spikes in branch mispredictability are the main cause of the dips in IPC. The graph also shows that for the intervals where branch misprediction increases, the data cache miss rate decreases significantly.

When the IPC falls in `su2cor`, both the branch and data miss rates increase, and they appear to be highly correlated.

For `vortex`, it is clear to see that the reason for the dips in IPC are due to the I-cache miss rate spikes.

5 Where to Simulate

In this section we discuss simulation methodology and the different options for selecting where to simulate. We start out by first discussing simulation methodology, to motivate the need for fast forwarding and knowing in detail where to fast forward.

5.1 Simulation Models

When examining and optimizing strictly in-order machines, a trace base simulation can be quite effective at representing the real execution on the machine. In an in-order machine events occur sequentially with respect to the instructions, thus a single instruction contains a great deal of the machines state information.

program	IPC-R	IPC-B	IPC-V	IPC-A	IPC-D	IPC-I	B-V	B-A	B-D	D-V	D-A
ijpeg	0	0	+	+	+	NA	0	0	0	+	+
vortex	+	—	—	—	NA	—	+	+	NA	NA	NA
applu	—	+	—	0	+	NA	—	0	+	—	NA
fpppp	+	+	—	+	+	NA	—	+	+	—	+
mgrid	NA	+	NA	NA	NA	NA	NA	NA	NA	NA	NA
tomcatv	+	—	NA	NA	NA	NA	NA	NA	NA	NA	NA
turb3d	—	—	+	+	—	NA	—	—	+	—	—
wave5	NA	—	+	+	+	NA	—	—	—	NA	+
su2cor	—	—	+	+	—	0	—	—	+	—	—

Table 2: For programs with cyclic behavior, this table shows the positive and negative correlation between IPC, RUU occupancy (R), branch prediction (B), address prediction (A), value prediction (V), and data cache miss rates (D). A + means that the two metrics both increase or decrease together. A — means that when one metric increases the other decreases, and visa versa. A 0 means that there is basically no correlation visible. N/A means that one or both of the metrics never really varied during execution.

In contrast to this, a more modern out of order machine with speculative execution executes many instruction simultaneously, and the sequential nature is lost. This profoundly effects almost every analysis because time, and even order, are decoupled from the code found in the executable. Branch prediction is affected by the fact that many predictions may be needed per cycle, an affect outside the scope of trace based simulation. This in turn affects the memory hierarchy, which may fetch data for loads that we later squashed. It was shown by Pai et.al. [10] that in-order execution models cannot model the effects of out of order execution with reasonable accuracy. To correctly model such a machine requires complex software, software which due to it's complexity executes slowly.

SimpleScalar [1], one of the fastest simulators, executes on the order of 1000 times slower than hardware. Several techniques exist for speeding out of order execution simulation, such as memoization and direct execution.

To help decrease simulation time, Schnarr and Larus examined using direct execution with memoization [12]. With memoization, the simulation of an executable often times reaches sections of code that perform the same function as something already done in the past, these past executions are cached, and then replayed when they are seen again later. Direct execution, the translation of simulated hardware to native machine code, was also shown to be applicable to out of order execution with some effort by Krishnan and Torrellas [6].

However, even with these techniques, full simulation is still often many times too slow to simulate a large number of real programs to completion. The resources to execute hundred of billions of instructions for each of the eighteen SPEC95 benchmarks is far outside the average researcher, especially if several design alternatives are in need of evaluation. In addition, direct execution and memoization still has potential problems when dealing with speculative execution and recovery (e.g., value prediction), and execution driven simulation is still needed.

5.2 Limited Simulation

For the above reasons, to test a research design using detailed pipeline simulation, most researchers execute only a small fraction of the program. A few hundred million instructions may be typically executed, starting from a

predetermined point.

Historically researchers executed from the start of the application, but this usually does not represent the majority of the program's behavior because it is still in initialization phase. Recently researchers have started to *fast-forward* to a given point in execution, and then start their simulation from there, ideally skipping over the initialization code to an area of code representative of the whole. During fast-forward the simulator simply needs to act as a functional simulator, and may take full advantage of direct execution. After the fast-forward point has been reached, the simulator switches to full timing simulation. Unfortunately, without analysis to determine the relevance of the sampling being simulated, blindly fast-forwarding may still not provide a representative sample of the program's execution.

Another potential option is to perform sampling over a larger area of instructions. To provide meaningful results, one still has to sample at intervals of 10 million or more sequential instructions in order to provide meaningful results due the time it takes to warm up the architecture structures (.e.g, caches). Conte et.al. [3] show techniques for the reconciliation of such disjoint sample points. However the basic problem is still the same, the blind simulation of a random place in a program's execution, will not yield consistent and representative results.

5.3 Choosing Where to Simulate

One solution is to take advantage of the nature of the programs. Most programs tend to be cyclic in nature, due to the loop model prevalent in modern software. By *cyclic behavior* we mean that the IPC, along with the other attributes of the program, alternate between a high and low steady state repetitively until the program terminates. As the graphs in the last section already showed, these loop structures are often times quite large, on the order of six billion instructions for some programs. The results show that simulating one of these cyclic loops should exactly equal the results obtained from the complete execution of the program. This advocates for simulating an even number of cyclic phases at carefully chosen points in the program.

Table 3 shows the steady state statistics for the non-cyclic programs. Each of these programs has the form Initialization-SteadyState-Finish. The *start* column is the number of simulation intervals (in terms of 100 million committed instructions) after which the simulation reaches steady state. The rest of the data in the table corresponds to when the program is in its steady state phase. *IPC* is the instructions per cycle, *Br* is the branch misprediction rate reported from the simulator, *Data* is the L1 data cache miss rate for a 64k 4-way cache, and *Inst* is the L1 Instruction cache miss rate for a 32k 2-way cache. *Val* is the average steady state value misprediction rate, and *Addr* is the address misprediction rate. Value prediction and address prediction are not applied in the simulator and thus do not effect cache miss rates or IPC. The results show for `perl` that simulation needs to start at 900 million committed instructions in order to skip over the initialization phase. The `perl` graph in figure 6 shows during the initialization that the value prediction miss rate is 15%, but during the steady state it is 21%, as is summarized in table 3.

Table 4 shows, for the cyclic programs, the difference between various phases of execution for the above listed statistics. The first group of results are the steady state results when the IPC is high, and the second set of results are when the IPC is low. The new column *cycle*, is the number of hundreds of millions of committed instructions that

		Average Behavior						
program	start	IPC	Br	Val	Addr	Data	Inst	Ruu
gcc	0	1.8	7.9	21	21	4.7	7.6	34
go	30	1.8	17	25	20	0.7	1.4	40
li	27	2.7	4.0	28	32	3.1	0.0	42
m88ksim	5	3.6	2.1	7.6	6.3	0.3	0.4	45
perl	9	3.0	2.6	21	22	0.5	0.7	44
apsi	2	1.8	5.9	30	6	14	0.0	65
hydro2d	6	0.8	0.3	8.7	0.5	29	0.0	70
swim	3	1.6	0.2	47	0.1	19	0.0	75

Table 3: Steady state program behavior for those programs which are non-cyclic.

			Average Behavior When IPC High						Average Behavior When IPC Low					
program	start	cycle	%high	IPC	Br	Val	Addr	Data	%low	IPC	Br	Val	Addr	Data
ijpeg	4	6	75	4.0	11	27	26	0.6	25	3.4	11	26	26	0.3
vortex	30	62	88	3.6	0.5	16	21	0.5	12	2.2	1.3	19	24	1.0
applu	5	3	33	1.8	6.3	25	6	19	67	1.7	5.9	34	6.2	13
fpppp	45	14	12	4.0	2.0	25	1.7	3.2	88	1.4	1.7	26	0.9	0.05
mgrid	1	5	60	1.9	2.1	48	1.2	7.0	40	1.8	2.0	49	1.2	7.0
tomcatv	13	5	40	0.94	0.61	47	0.28	18.5	60	0.88	0.67	47	0.31	18.5
turb3d	13	7	57	4.2	0.79	28	7.4	0.5	43	2.8	0.93	20	4.2	12.0
wave5	14	7	28	3.0	0.26	40	10	18	72	1.7	0.91	30	2.2	7.5
su2cor	20	37	30	2.1	1	48	17	28	70	1.8	11	40	8	30

Table 4: Phase behavior of programs with visible large scale cycles

are executed in a single large cyclic phase. The column *high* shows the percent of executed instructions in the cyclic phase when the IPC is high, and *low* shows the percent of executed instructions in the cyclic phase when IPC is low. Note that for some programs, such as `su2cor` and `vortex`, the difference is quite large. If one simulated for less than the length of the cyclic phase, very different results could be seen depending upon the few 100 of million instructions simulated.

6 Summary

This study compares the time varying behavior of the SPEC95 programs. The results show that there are interesting correlations between IPC, value prediction, branch prediction, address prediction, and data cache misses. This type of classification is needed for popular programs to help guide researchers in these areas.

Another contribution of this study is showing where to fast forward to and how long to simulate for the SPEC95 programs in order to simulate a representative part of the program's execution. We found that after the initialization phase the programs (running the SPEC95 reference input) either (1) converge to a steady state, or (2) have a cyclic IPC behavior which is repeated until the program terminates. We plan on making the SPEC95 executables and inputs

used available, so that others can use this paper with those binaries and inputs for architecture simulation research.

We are currently preparing this work for submission by analyzing code examples to show why there was significant correlation between these architecture features for turb3d, wave5, su2cor and vortex.

References

[1] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[2] T.F. Chen and J.L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 51–61, October 1992.

[3] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, October 1996.

[4] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. EE Department TR 1080, Technion - Israel Institue of Technology, November 1996.

[5] J. Gonzalez and A. Gonzalez. Memory address prediction for data speculation. Technical report, Universitat Politecnica de Catalunya, 1996.

[6] V. Krishnan and J. Torrellas. A direct-execution framework for fast and accurate simulation of superscalar processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1998.

[7] D. Lee, P. Crowley, J.-L. Baer, T. Anderson, and B. Bershad. Execution characteristics of desktop applications on windows nt. In *25th Annual International Symposium on Computer Architecture*, 1998.

[8] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *17th International Conference on Architectural Support for Programming Languages and operating Systems*, pages 138–147, October 1996.

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

[10] V. S. Pai, P. Ranganathan, and S. V. Adve. The impact of instruction-level parallelism on multiprocessor performance and simulation methodolgy. In *3rd International Symposium on High Performance Computer Architecture (HPCA)*, February 1997.

[11] G. Reinman and B. Calder. Predictive techniques for aggressive load speculation. In *31st International Symposium on Microarchitecture*, 1998.

[12] E. Schnarr and J. R. Larus. Fast out-of-order processor simulation using memoization. In *ASPLOS-VIII. Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–294, October 1998.

[13] G.S. Sohi. Instruction issue logic for high-performance, interruptable, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[14] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual International Symposium on Microarchitecture*, December 1997.