# PERFORMANCE ANALYSIS THROUGH SYNTHETIC TRACE GENERATION

*Lieven Eeckhout*[*] *Koen De Bosschere* *Henk Neefs*

Department of Electronics and Information Systems (ELIS), Ghent University
Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium
{leeckhou,kdb,neefs}@elis.rug.ac.be

## ABSTRACT

Most research in the area of microarchitectural performance analysis is done using trace-driven simulations. Although trace-driven simulations are fairly accurate, they are both time- and space-consuming which makes them sometimes impractical. Modeling the execution of a computer program by a statistical profile and generating a synthetic benchmark trace from this statistical profile can be used to accelerate the design process. Thanks to the statistical nature of this technique, performance characteristics quickly converge to a steady state solution during simulation, which makes this technique suitable for fast design space explorations. In this paper, it is shown how more detailed statistical profiles can be obtained and how the synthetic trace generation mechanism should be designed to generate syntactically correct benchmark traces. As a result, the performance predictions in this paper are far more accurate than those reported in previous research.

## I. INTRODUCTION

Trace-driven simulations are are generally accurate [1]. However, there are serious practical shortcomings with trace-driven simulations. First, tracing complete benchmark executions is infeasible as this requires the storage of billions of instructions. Second, simulation time is also prohibitive for such huge traces; especially if traces are used to evaluate various processor configurations for various workloads, which requires many simulation runs.

In practice, researchers try to limit the size of traces by storing only a part of them. One can either take a contiguous part of the trace (preferably not a part that belongs to the initialization sequence of the program), or apply trace sampling [3]. Recently, statistical modeling [2, 7] was presented as a technique to accelerate the simulation process. First, a statistical profile or a set of statistical program characteristics is extracted from a program execution. This statistical profile is then used to generate a synthetic trace which is subsequently fed into a trace-driven simulator, which will estimate the attainable performance for the microarchitecture modeled. This approach has two major advantages. First, due to the statistical nature of the synthetic trace generation process, performance characteristics will quickly converge, and hence the number of clock cycles to simulate can be limited. As a result, this methodology can be used to perform a quick design space exploration in an early design stage. Second, by assuming statistical independence of various program characteristics, the statistical profile will be much more compact than a trace, and does (generally) not depend on the size of the trace.

A statistical profile includes many relevant properties of a benchmark execution except for dynamic properties, e.g. different phases in a program execution. The evaluation of our technique does however not reveal that this is a serious practical problem. If this effect would ever prohibit accurate performance modeling, a dynamic program execution could easily be segmented, and a separate statistical profile per segment could be measured.

Another disadvantage of current statistical modeling techniques, is that they cannot be used to study application locality. The effects of caches, TLBs and branch predictors cannot be scaled down to smaller traces and hence synthetic traces are inadequate to study them. A straightforward solution to this problem is to model cache, TLB and branch behaviour statistically as well, as is done in this paper.

This paper has two major contributions. First of all, we show how more detailed statistical profiles can be obtained by measuring conditional distributions and by including memory dependencies. The second contribution is that we propose a synthetic trace generation algorithm which guarantees syntactically correct synthetic traces, i.e. stores and conditional branches should not have a destination operand. Syntactical correctness of synthetic traces is motivated by the fact that we want to simulate synthetic traces on existing simulation tools. Both the more enhanced statistical profiles and the modified synthetic trace generator will lead to more representative synthetic traces and to far more accurate performance predictions than previously reported by Carl and Smith in [2]: the relative error between the predicted and the actual IPC reported here ranges from -8% to 14% for the SPECint95 benchmarks for a 16-issue out-of-order processor configuration with an

[*]Lieven Eeckhout is supported by a grant from the Flemish Institute for the Promotion of the Scientific-Technological Research in the Industry (IWT).

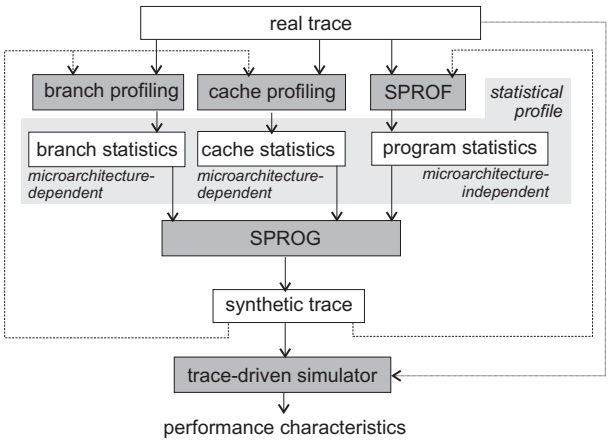


Figure 1: The SPROF/SPROG performance modeling environment. The coloured boxes are tools; the white boxes are intermediate files.

instruction window of 128 instructions.

This paper is organized as follows. Section II presents a general overview of our performance modeling environment called SPROF/SPROG. In section III it is shown which statistical characteristics are extracted from benchmark traces to form a viable statistical profile. Section IV presents the synthetic trace generation algorithm being used. Our performance modeling environment is validated in section V. Section VI discusses related work and, finally, we will conclude in section VII.

## II. THE SPROF/SPROG FRAMEWORK

Our general framework for performance modeling is depicted in Figure 1. First, a real program trace, for example a SPEC benchmark trace, is analyzed by two microarchitecture-dependent profiling tools and one microarchitecture-independent profiling tool. The microarchitecture-independent profiling tool called SPROF[1] extracts statistics concerning the instruction mix and the dependencies between instructions through register values as well as through memory values. The microarchitecture-dependent profiling tools extract statistics concerning the branch and cache behaviour of the program trace for a specific branch predictor and a specific cache organization. This is done by simulating the desired aspect, namely the branch or the cache behaviour. The complete set of statistics (program, branch and cache statistics) which are generated by trace profiling, is called a *statistical profile*.

Note that a statistical profile can be computed from an actual trace as is shown in Figure 1, but it is more convenient to compute it on-the-fly from either an instrumented functional simulator, or from an instrumented version of the benchmark program running on a real system without the need to store huge traces. A second important note is that although computing a statistical profile might take a long time, it should be done only once for each benchmark. And since performance analysis based on a statistical profile is fast, computing a statistical profile will be worthwhile.

Once a statistical profile is computed, SPROG[2] generates a *synthetic trace* using this statistical profile. This synthetic trace can now be simulated on a trace-driven simulator. If the synthetic trace captures the right execution characteristics, the performance characteristics of the original and the synthetic trace should be comparable when simulated on the same simulator modeling the same architecture.

The next section discusses statistical profiling; section IV provides details on the synthetic trace generation process.

## III. STATISTICAL PROFILING

While searching for a viable statistical profile, three major goals have to be fulfilled. First, the performance characteristics of the synthetic trace on a particular architecture should be comparable to the performance characteristics of the corresponding original trace. Second, the statistical profile should not be too complicated, i.e. the number of distributions involved should be limited while preserving high levels of performance prediction accuracy. A third goal is to generate benchmark traces which are syntactically correct. More specifically, a store operation should not have a destination operand—which cannot be guaranteed in [2, 7]—and an integer operation should not have four or five source operands. The underlying motivation for this goal is that we should be able to simulate the synthetic trace on the existing trace-driven simulator, see Figure 1. This goal can be fulfilled by both carefully selecting the statistical profile distributions and by carefully designing the synthetic trace generator, as will become clear in this paper.

Section A describes which microarchitecture-independent program statistics are measured by SPROF. The microarchitecture-dependent execution statistics are detailed in section B.

### A. Microarchitecture-independent statistics

The program execution statistics which are measured by SPROF, are microarchitecture-independent.

The first program characteristic which we identified, is the *instruction-mix* distribution. We have measured the probability that the type $T_x$ of instruction $x$ equals $t$, for $t$ one of the $N_{instr}$ instruction types included in the model. Formally stated, we measured $Prob\ [\ T_x = t\ ]$. In our model, we identify nine instruction classes ($N_{instr} = 9$) because of their varying execution latencies and semantics[3]: integer, load, store, conditional branch, jump,

[1] Statistical PROFiler

[2] Statistical PROGram trace generator

[3] These nine instruction classes were identified for the Alpha ISA. However, this does not affect the generality of the technique presented

integer multiply, floating-point operation, floating-point divide single-precision and floating-point divide double-precision.

The second distribution which we identified, is the *number-of-operands* distribution or the probability that instruction $x$ has $O_x$ source operands. Since the number of operands of an instruction is dependent on its type, we have actually measured $Prob\ [\ O_x = o \mid T_x = t\ ]$. The variable number of operands is caused by the fact that some instructions occur in a register-register as well as in a register-immediate format, while belonging to the same instruction class $t$ in our classification.

We have also measured the *age-of-register-operands* distribution [4] for each register operand of each instruction type. Indeed, we measured the probability that the $i$-th register operand of an instruction $x$ of type $T_x$ having $O_x$ operands, is produced $\delta$ instructions before it in the trace; i.e. instruction $x - \delta$ produces a register instance which is consumed by instruction $x$. Formally stated, we measured $Prob\ [\ A_{i,x} = \delta \mid T_x = t\ ,\ O_x = o\ ]$, with $i = 1, 2$ and $A_{i,x}$ the age of the $i$-th register operand of instruction $x$. Notice that in these distributions only real (RAW) data dependencies were considered.

From the experiments presented in section V, it became clear that these distributions are not sufficient to accurately predict the attainable performance of out-of-order processors. Therefore, we decided to measure these distributions conditionally on the types of the $n$ instructions before instruction $x$ in the trace, with $n$ ranging from 0 (which corresponds to the distributions described above) to 3. The resulting distributions then are, e.g. for $n = 2$: $Prob\ [\ T_x = t \mid T_{x-1} = t'\ ,\ T_{x-2} = t''\ ]$, $Prob\ [\ O_x = o \mid T_x = t\ ,\ T_{x-1} = t'\ ,\ T_{x-2} = t''\ ]$ and $Prob\ [\ A_{i,x} = \delta \mid T_x = t, O_x = o, T_{x-1} = t', T_{x-2} = t''\ ]$, for $i = 1, 2$. These distributions will be further denoted for arbitrary $n$ as $P_{T,n}$, $P_{O,n}$ and $P_{A,n}$, respectively.

Since we also wanted to model memory dependencies, we identified the probability that a load is *memory-dependent* on the $\delta$-th store before it in the trace. In other words, we measured the *read-after-write* distribution, $Prob\ [\ M_x = \delta \mid T_{x,\delta} = \text{ST}\ ,\ T_x = \text{LD}\ ]$, with $T_{x,\delta}$ the $\delta$-th memory operation before instruction $x$ in the trace.

Note that most of the distributions mentioned here are theoretically infinite. But for practical purposes, they can be truncated at a certain limit $N_{max}$. This limit imposes a constraint on the window size of the out-of-order architectures being modeled. In this paper, we chose $N_{max} = 128$ which allows the exploration of all contemporary out-of-order architectures.

As stated in section III, the amount of storage required to store a statistical profile should be limited. The total number of probabilities to be stored here is $\mathcal{O}(N_{max} \times N_{instr}^{n+1})$, which is exponential in $n$, where $n$ specifies what conditional distributions $P_{T,n}$, $P_{O,n}$ and $P_{A,n}$ are being used. However, for small values of $n$—in our case, $n$ ranges from 0 to 3—this is quite acceptable since the number of instruction classes $N_{instr} = 9$ is small as well. Noonburg and Shen [7] on the other hand, used the following distribution to determine the type $T_x$ and the *dependent instruction distance* $D_x$ of an instruction $x$: $Prob\ [\ T_x = t\ ,\ D_x = \delta \mid T_{x-1} = t'\ ,\ D_{x-1} = \delta'\ ,\ T_{x-2} = t''\ ,\ D_{x-2} = \delta''\ ]$. These distributions require $N_{max}^3 \times N_{instr}^3$ probabilities to be stored in a statistical profile. This is feasible for performance modeling of architectures with small instruction windows where $N_{max}$ can be restricted, e.g. $N_{max} = 5$ in [7]. But in our case this would require $128^3 \times 9^3 \approx 1.5$ G probabilities to be stored, which is impractical.

here.

### B. Microarchitecture-dependent statistics

The microarchitecture-dependent statistics are the branch and the cache statistics, see Figure 1. These were collected by simulating the traces on simple simulators which measured the branch and cache behaviour, respectively.

The branch statistics used in this paper consist of two probabilities only, namely the probability that a branch is correctly predicted by the branch predictor (commonly denoted as the branch prediction accuracy) and the probability that the branch target address is correctly predicted by the branch target buffer (BTB). The reason to distinguish between these two probabilities is that the penalties introduced by a branch target misprediction and a branch misprediction are completely different [5]. A branch which is correctly predicted but whose target is not, only introduces a single-cycle bubble in the pipeline; a mispredicted branch will cause the entire pipeline to be flushed and to be refilled when the branch is executed.

The cache statistics include two sets of distributions: the D-cache and the I-cache statistics. The D-cache statistics contain two probabilities, namely the probability that a load operation needs to access the L2 cache (L1 cache miss) and main memory (L2 cache miss) to get its data. The I-cache statistics consists of two probabilities as well, namely the probability that the fetch unit needs to access the L2 cache and main memory to get an instruction.

## IV. SYNTHETIC TRACE GENERATION

The synthetic trace generator SPROG takes as input a statistical profile. A synthetic trace is generated à la Monte Carlo: a random number is generated, which will determine a program characteristic using the cumulative distribution function.

The generation of a synthetic trace itself works on an instruction-by-instruction basis. Consider the generation of the $x$-th instruction in the instruction stream:

1. Determine the instruction type $T_x$ using the instruction-mix distribution.
2. Determine the number of source operands $O_x$ using the number-of-operands distribution.

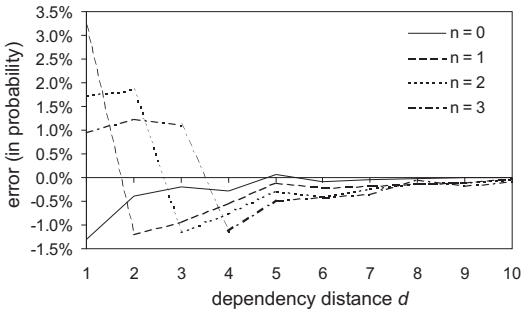


Figure 2: The deviation between the marginal age-of-register-operands distribution of the original and the synthetic trace (for the `li` benchmark) as a function of the dependency distance $d$ and the number of instructions $n$ which are used in the conditional distributions.

3. For each source operand, determine the instruction which produces this register instance using the age-of-register-operands distribution. In this step, syntactical correctness is guaranteed as follows: look for register dependencies until we get a register instance which is not created by a store or a branch operation. If after a certain maximum number of trails, 10,000 in our case, still no valid dependency is found, instruction $x$ is made dependent on an instruction which comes at least $N_{max}$ instructions before it in the trace. For processors with an instruction window size less than $N_{max}$, this means that the dependency is simply squashed. We made this design option in order not to eliminate too many register dependencies.

4. If instruction $x$ is a load operation, use the read-after-write distribution to determine which store operation accesses the same memory address as instruction $x$ does.

5. If instruction $x$ is a branch instruction, determine whether the branch and its target will be correctly predicted using the branch statistics.

6. If instruction $x$ is a load operation, determine whether the load will get its data from L1 cache, L2 cache or main memory using the D-cache statistics.

7. Determine whether or not instruction $x$ will cause an I-cache miss at the L1 or L2 level.

The method proposed here to enforce syntactical correctness has some serious implications on the representativeness of the synthetic trace. In Figure 2, the deviation is shown between the desired marginal age-of-register-operands distribution—measured from the real program trace—and the marginal age-of-register-operands distribution of the generated synthetic trace as a function of $n$ which specifies the conditional distributions $P_{T,n}$, $P_{O,n}$ and $P_{A,n}$ used. For $n = 0$, we observe that the distribution resulting from synthetic trace generation (generally) lies under the distribution of the real trace[4]. This is due to the fact that if no dependency is found which is not supposedly created by a store or a conditional branch, the instruction is made dependent on an instruction which comes at least $N_{max}$ instructions before it in the trace. For $n > 0$, the distribution of the synthetic trace lies above the desired distribution for dependency distances $d \leq n$. This can be explained as follows: when a dependency needs to be established, it will be more probable than specified in the original distributions that instruction $x$ will be made dependent on instruction $x - i$, with $i = 1, \ldots, 3$ given the fact that instruction $x-i$ is neither a store nor a conditional branch. And since it is more probable that an instruction is neither a store nor a conditional branch, the probabilities for $d < n$ will be higher for the synthetic trace than for the original one.

From these considerations we can conclude that it can not be guaranteed that the statistical profile of the synthetic trace equals the statistical profile of the real program trace. Therefore, we use *instantaneous positive-error* distributions. An instantaneous positive-error distribution is attained by computing the errors between the desired probabilities and the probabilities at that time during the synthetic trace generation process, only keeping the positive errors and normalizing them to a distribution. When dependencies are generated using the instantaneous positive-error distribution, dependency distances whose instantaneous probability is lower than the desired probability, will be benefited. At the same time, dependency distances whose instantaneous probability is higher than the desired probability, will be harmed. In fact, the use of the instantaneous positive-error distribution introduces a feedback loop in the synthetic trace generation algorithm. As a result, the synthetic trace will have the same statistical profile as the original trace, which has been verified.

| benchmark | input | $N_{skipped}$ | $N_{instrs}$ |
|---|---|---|---|
| li | train.lsp | 0 | 226 |
| go | 50 9 2stone9.in | 250 | 200 |
| compress | < train.in (50K) | 0 | 217 |
| gcc | gcc.i -O | 0 | 182 |
| m88ksim | -c < train.in | 350 | 200 |
| ijpeg | penguin.ppm | 100 | 170 |
| perl | scrabbl.pl < scrabbl.in | 600 | 200 |
| vortex | vortex.in (persons.250) | 800 | 200 |

Table 1: The SPECint95 benchmarks used, the input files, the number of initial instructions skipped, and the number of instructions incorporated in the trace.

## V. EVALUATION

### A. Benchmarks

The traces used to collect program execution statistics, see Table 1, were generated from the SPECint95 benchmarks

[4]Note that this is only true for small values of dependency distances; for higher dependency distances, in our case more than $N_{max}$, the opposite will be true since distributions sum to one.

on a DEC 500au station with an Alpha 21164 processor. The Alpha architecture is a load/store architecture and has 32 integer and 32 floating-point registers, each of which is 64 bits wide. The SPECint95 benchmarks have been compiled with the DEC cc compiler version 5.6 with the optimization flag set to -O4. The traces were carefully selected not to include initialization code.

### B. Out-of-order architecture

To validate our performance modeling methodology, contemporary out-of-order superscalar architectures were assumed. The architectures considered are organized as follows: an instruction window of 32, 64 and 128 instructions; an issue width of 4, 8 and 16; 2, 4 and 8 memory units; 3, 5 and 10 non-memory units, respectively. The fetch bandwidth and the reorder bandwidth were chosen to be the same as the issue bandwidth. The latencies of the instruction types are: integer 1 cycle, load 3 cycles (this includes address calculation and L1 D-cache access), multiply 8 cycles, FP operation 4 cycles, single and double precision FP divide 18 and 31 cycles, respectively. All operations are fully pipelined, except for the divide.

A dynamic memory disambiguation strategy is assumed; re-execution is implemented to recover from misspeculated loads, which re-executes the instructions which are dependent (directly or indirectly) on the misspeculated load.

The branch predictor is a hybrid predictor with a 4K meta predictor choosing between a 4K bimodal predictor and an 8-bit gshare that indexes into a 4K predictor [6]. The branch target buffer contains 512 sets and has an associativity of 4. The return address stack contains four entries. The pipeline in front of the instruction window contains four stages: a fetch stage, a decode stage, a register renaming stage and a dispatch stage which inserts the instructions in the instruction window.

The processor configurations simulated have a 32K direct mapped L1 I-cache and a 64K 2-way set-associative L1 D-cache, both having a block size of 32 bytes. The L2 cache is a unified 512K 4-way set-associative cache with a block size of 32 bytes. A load which needs to access L2 cache or main memory takes 11 or 81 cycles, respectively.

### C. Performance prediction accuracy

Figure 3 presents the relative error $\Delta_{IPC}$,

$$\Delta_{IPC} = \frac{IPC_{\textit{synthetic trace}} - IPC_{\textit{real trace}}}{IPC_{\textit{real trace}}}$$

between the IPC value of the synthetic trace and the corresponding real trace. This is shown for several processor configurations: instruction windows containing 32, 64 and 128 instructions and issue widths of 4, 8 and 16 in the first, the second and the third column, respectively. The first row presents the relative IPC error when only data dependencies through register and memory values are considered; i.e. perfect branch prediction and perfect caches are assumed. The various bars in the figures denote the various statistical profiles and the two synthetic trace generation algorithms used. The no fb and fb abbreviations indicate whether or not the instantaneous positive-error distribution is used by the synthetic trace generator; in other words, whether or not a feedback loop is used. The various n values indicate the specific conditional probabilities used in the statistical profiles: $P_{T,n}$, $P_{O,n}$ and $P_{A,n}$. Several important conclusions can be taken from these graphs. First, performance predictions are generally more accurate for processors with smaller issue widths and smaller instruction windows. This is due to the fact that for these processors, performance is more limited by machine parallelism than by program parallelism. Second, the feedback loop in the synthetic trace generator generally leads to lower IPC values and more accurate performance predictions. This is a logical consequence of the fact that less data dependencies are squashed to satisfy the syntactical correctness of synthetic traces, as explained in section IV. The third conclusion is that higher values of $n$ generally lead to more accurate performance predictions what could be expected since these statistical profiles contain more information than statistical profiles with lower values of $n$.

The second row of Figure 3 shows results of experiments on the original traces with a probabilistic branch predictor and probabilistic D- and I-caches. These experiments were conducted as follows: information is added to the original trace as is done in the last three steps of the synthetic trace generation algorithm, see section IV. The three microarchitecture-dependent statistics are validated here: the branch statistics, the D-cache and the I-cache statistics. From these graphs we can conclude that a probabilistic branch predictor never leads to significant performance prediction inaccuracies. Probabilistic I-caches and D-caches on the other hand, can lead to substantial inaccuracies. This is probably due to the clustering of cache misses in real program traces, which was first observed by Voldman *et al.* in [8] and which is not modeled here.

The third row of Figure 3 presents overall performance prediction results: i.e. a synthetic trace with a probabilistic branch predictor and probabilistic caches are considered. The relative IPC error varies between -8% and 14% over the SPECint95 benchmarks for the 16-issue processor with a window size of 128 instructions. This is significantly better than the results reported by Carl and Smith in [2].

## VI. RELATED WORK

Noonburg and Shen [7] presented a framework which models the execution of a program on a particular architecture as a Markov chain. The statistical profile used here is less complex than the one presented in [7], see section III, and the architecture is modeled as a trace-driven simulator which is much less complex than a corresponding Markov chain. Moreover, Noonburg and Shen [7] did not model memory dependencies and assumed only one dependency per instruction.

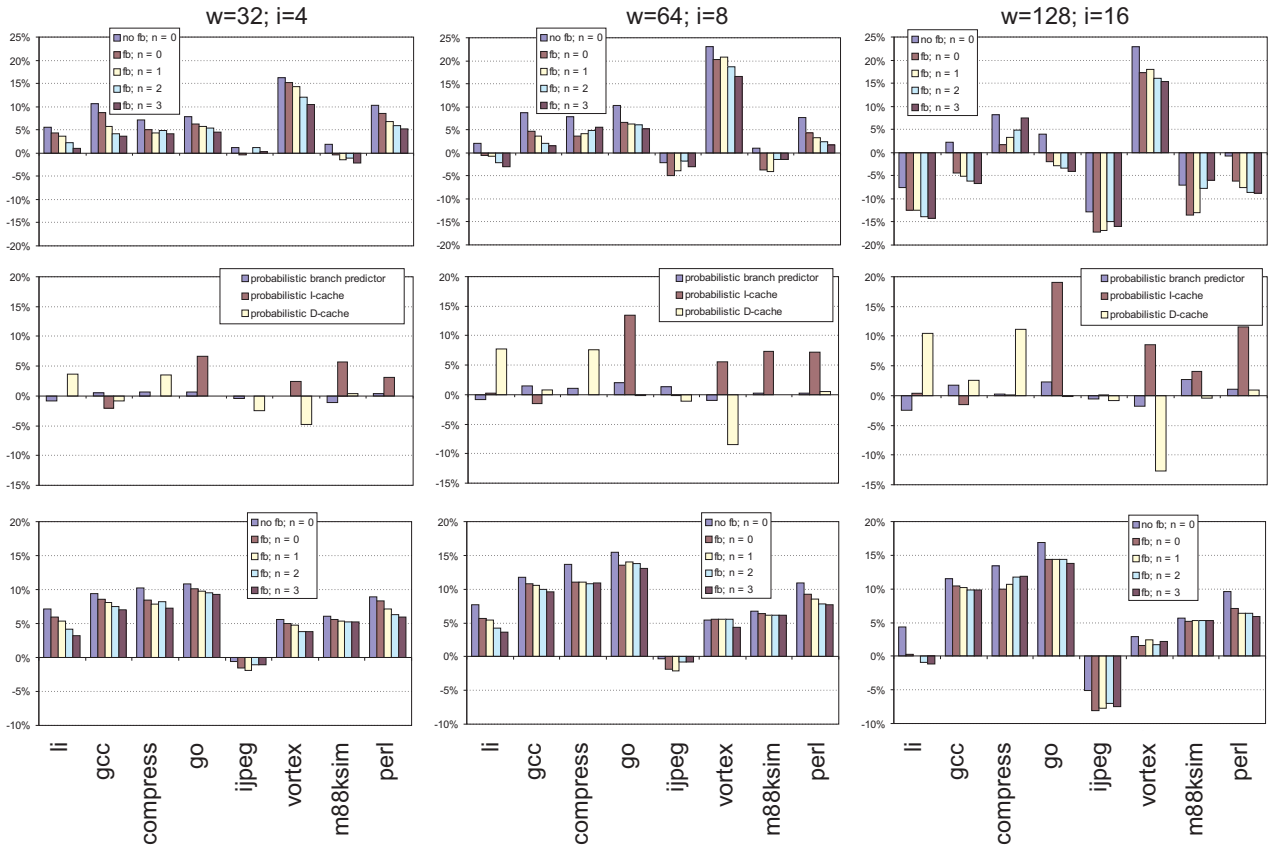


Figure 3: Performance prediction accuracy: the relative IPC error $\Delta_{IPC}$ between the IPC value of the synthetic trace and the corresponding real trace. The first row: synthetic trace with perfect branch predictor and perfect caches; second row: real trace with probabilistic branch predictor and probabilistic caches; third row: synthetic trace with probabilistic branch predictor and probabilistic caches.

Carl and Smith [2] proposed a hybrid approach, where a synthetic instruction trace was generated based on execution statistics and fed into a trace-driven simulator, as is done here. The work presented here is a continuation of the work initiated by Carl and Smith [2] by suggesting several improvements: incorporating memory dependencies, using more detailed statistical profiles and guaranteeing syntactical correctness of the synthetic traces. As a result, the performance predictions reported in this paper are far more accurate than those reported in [2].

## VII. CONCLUSION

In this paper, it was shown how the performance prediction accuracy of statistical modeling can be increased by enhancing the statistical profiles and by guaranteeing syntactically correct synthetic traces. As a result, the relative prediction errors reported here range from -8% to 14% over the SPECint95 benchmarks for a 16-issue out-of-order processor with an instruction window of 128 instructions, which are far more accurate predictions than those reported by Carl and Smith in [2].

## REFERENCES

[1] B. Black and J. P. Shen. Calibration of microprocessor performance models. *IEEE Computer*, 31(5):59–65, May 1998.

[2] R. Carl and J. E. Smith. Modeling superscalar processors via statistical simulation. In *Workshop on Performance Analysis and its Impact on Design (PAID), held in conjunction with the 25th Annual International Symposium on Computer Architecture (ISCA)*, June 1998.

[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 International Conference on Computer Design (ICCD)*, pages 468–477, October 1996.

[4] M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 22nd Annual International Symposium on Microarchitecture (MICRO)*, pages 236–245, December 1992.

[5] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, third edition, 2003.

[6] S. McFarling. Combining branch predictors. Technical Report WRL TN-36, Digital Western Research Laboratory, June 1993.

[7] D. B. Noonburg and J. P. Shen. A framework for statistical modeling of superscalar processor performance. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture (HPCA)*, pages 298–309, February 1997.

[8] J. Voldman, B. Mandelbrot, L. W. Hoevel, J. Knight, and P. Rosenfeld. Fractal nature of software-cache interaction. *IBM Journal of Research and Development*, 27(2):164–170, March 1983.