Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies

Lieven Eeckhout[†] Robert H. Bell Jr.[‡] Bastiaan Stougie[†] Koen De Bosschere[†] Lizy K. John[‡]

[†]ELIS Department
Ghent University, Belgium
{leeckhou,bastiaan,kdb}@elis.ugent.be

[‡]ECE Department
The University of Texas at Austin
{robbell,ljohn}@ece.utexas.edu

Abstract

Designing a new microprocessor is extremely time-consuming. One of the contributing reasons is that computer designers rely heavily on detailed architectural simulations, which are very time-consuming. Recent work has focused on statistical simulation to address this issue. The basic idea of statistical simulation is to measure characteristics during program execution, generate a synthetic trace with those characteristics and then simulate the synthetic trace. The statistically generated synthetic trace is orders of magnitude smaller than the original program sequence and hence results in significantly faster simulation.

This paper makes the following contributions to the statistical simulation methodology. First, we propose the use of a statistical flow graph to characterize the control flow of a program execution. Second, we model delayed update of branch predictors while profiling program execution characteristics. Experimental results show that statistical simulation using this improved control flow modeling attains significantly better accuracy than the previously proposed HLS system. We evaluate both the absolute and the relative accuracy of our approach for power/performance modeling of superscalar microarchitectures. The results show that our statistical simulation framework can be used to efficiently explore processor design spaces.

1. Introduction

Designing a new microprocessor is both complex and time-consuming (taking up to 7 years [19]). Computer designers rely heavily on detailed architectural simulators to identify the optimal design in a large design space under a number of constraints such as chip area, power budget, etc. These architectural simulation tools are at least a factor of a thousand slower than native hardware execution. Another issue that contributes to the long simulation time is the use of real-world applications as benchmarks and the ever-increasing number of dynamic instructions that need to be simulated. The increasing performance of current microprocessor systems coupled with the increasing complexity of current computer applications means that the dynamic instruction count must be increased proportionally to simulate a respectable time slice of a real system. For example, some benchmarks in the SPEC CPU2000 benchmark suite have a dynamic instruction count that is greater than 500 billion instructions [12]. Since several benchmarks may need to be simulated and various design points evaluated, the consequences are an impractically long simulation time and an undesirably long time-to-market.

Researchers have proposed several techniques to shorten the total simulation time such as sampling [6, 25, 29], reduced input sets [18] and analytical modeling [7, 20, 27]. Over the last few years, interest has grown in yet another approach, namely statistical simulation [5, 8, 9, 10, 21, 22, 23, 24]. The basic idea of statistical simulation is simple: measure a well-chosen set of program characteristics during execution, generate a synthetic trace with those characteristics and simulate the synthetic trace. If the set of characteristics reflects the key properties of the program's behavior, accurate performance/power predictions can be made. The statistically generated synthetic trace is several orders of magnitude smaller than the original program execution, and hence simulation finishes very quickly. The goal of statistical simulation is not to replace detailed simulation but to be a useful complement. Statistical simulation can be used to identify a region of interest in a large design space that can, in turn, be further analyzed through slower but more detailed architectural simulations.

In this paper, we present an improved statistical simulation framework that extends previous work with two major contributions. First, we propose the use of a statistical flow graph to characterize the control flow of a program's execution. Control flow behavior is characterized by modeling sequences of basic blocks along with their mutual transition probabilities and execution characteris-

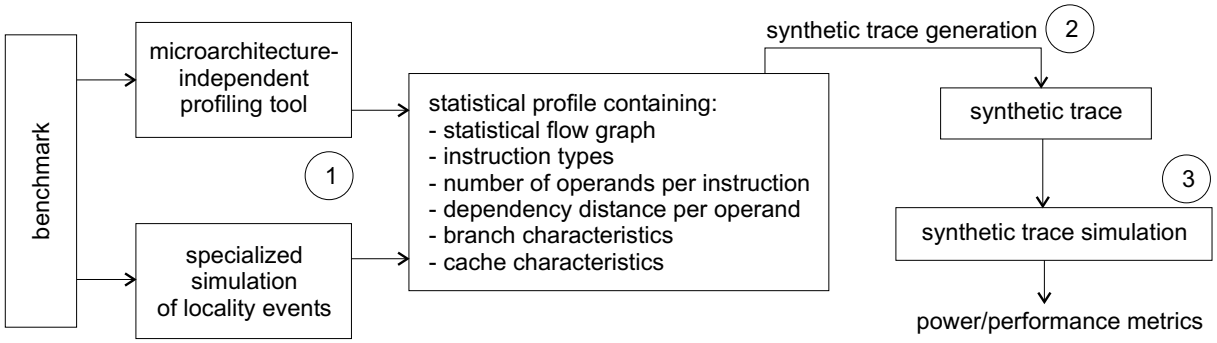

Figure 1. Statistical simulation: general framework.

tics. This statistical flow graph combines the graph representation proposed in the SMART technique by Iyengar *et al.* [14, 15] with previously proposed statistical simulation frameworks [5, 8, 9, 10, 22, 23, 24]. This combines the major benefit of SMART, workload modeling accuracy, with the major benefits of statistical simulation, simplicity and rapid convergence. Second, we show that it is important to consider delayed update when characterizing the branch behavior. This improved statistical simulation framework is extensively evaluated by considering both absolute and relative accuracy in modeling the performance and energy consumption of superscalar microarchitectures. We report an average error of 6.6% and 4% for predicting performance and energy, respectively, on an 8-way superscalar out-of-order processor using SPECint2000 benchmarks. We also show that our framework is significantly more accurate than the previously proposed HLS framework. In addition, we demonstrate that the error when predicting relative performance/power trends is generally less than 3%. As a consequence, we conclude that statistical simulation is a useful tool for accurately and efficiently exploring processor design spaces.

This paper is organized as follows. Section 2 presents our statistical simulation framework: the use of the statistical flow graph is discussed and our branch profiling approach using delayed update is proposed. Section 3 discusses our experimental setup which is used in Section 4 during the evaluation. Related work and how it differs from this work is discussed in Section 5. Finally, we conclude in Section 6.

2. Statistical simulation

Statistical simulation consists of three steps as shown in Figure 1. In the first step, a collection of program execution characteristics is measured. Subsequently, this *statistical profile* is used to generate a *synthetic trace*. In the final step, this synthetic trace is simulated on a trace-driven simulator. In the following subsections, we discuss all three steps.

2.1. Statistical profiling

In our statistical profiles, we make a distinction between microarchitecture-independent characteristics and microarchitecture-dependent characteristics. This will be discussed in the following two subsections. In the final subsection, we discuss how to improve the microarchitecture-dependent branch characteristics.

2.1.1. Microarchitecture-independent characteristics. During statistical profiling we build a *statistical flow graph (SFG)*. To clarify how this is done, we refer to Figure 2 in which first- ($k = 1$) and second-order ($k = 2$) SFGs are shown for an example basic block sequence 'AABAABCABC'. Each node in the graph represents the history of the preceding basic block(s) as its state. This is shown with the labels 'A', 'B' and 'C' in the first-order SFG and labels 'AA', 'AB', 'BA', 'BC' and 'CA' in the second-order SFG. The numericals in each node show the *occurrences* or the number of times the history of preceding basic block(s) appears in the basic block stream. The labels and the percentages next to the edges represent the current basic block and the transition probabilities between the nodes $Prob[B_n|B_{n-1}, \ldots, B_{n-k}]$, with k being the order of the SFG. Note that during statistical profiling only one SFG is built for one specific value of k. In the evaluation section of this paper, we will evaluate the importance of the chosen value of k. For comparison, we will also consider $k = 0$ or no edges in the graph.

For each basic block in the SFG we record the instruction types of each instruction. We classify the instruction types into 12 classes according to their semantics: load, store, integer conditional branch, floating-point conditional branch, indirect branch, integer alu, integer multiply, integer divide, floating-point alu, floating-point multiply, floating-point divide and floating-point square root. For each instruction, we record the number of source operands. Note that some instruction types, although classified within the same instruction class, may have a different number of source

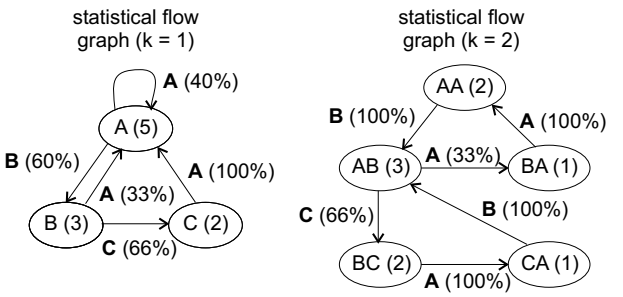

Figure 2. Example first-order (k=1) and second-order (k=2) SFGs corresponding to the basic block sequence 'AABAABCABC'.

operands. For each operand we also record the dependency distance which is the number of dynamically executed instructions between the production of a register value (register write) and the consumption of it (register read). We only consider read-after-write (RAW) dependencies since our focus is on out-of-order architectures in which write-after-write (WAW) and write-after-read (WAR) dependencies are dynamically removed through register renaming as long as enough physical registers are available. Although not done in this paper, this approach could be extended to also include WAW and WAR dependencies to account for a limited number of physical registers or in-order execution. Note that recording the dependency distance requires storing a distribution since multiple dynamic versions of the same static instruction could result in multiple dependency distances. In theory, this distribution could be very large due to large dependency distances; in practice, we can limit this distribution. This however limits the number of in-flight instructions that can be modeled during synthetic trace simulation. In our study, we limit the dependency distribution to 512 which still allows the modeling of a wide range of current and near-future microprocessors. More formally, the distribution of the dependency distance of the p-th operand of the o-th instruction in basic block B_n given its basic block history $B_{n-1}, \ldots, B_{n-k}$ can be expressed as follows: $Prob[D_{n,o,p}|B_n, B_{n-1}, \ldots, B_{n-k}]$.

Note that these characteristics are independent of any microarchitecture-specific organization. In other words, these characteristics do not rely on assumptions related to issue width, window size, etc. They are therefore called microarchitecture-independent characteristics.

2.1.2. Microarchitecture-dependent characteristics. In addition to the above characteristics we also measure a number of characteristics that are related to locality events, specifically the branch behavior and the cache behavior. The *branch characteristics* consist of three probabilities:

- the probability of a taken branch, which will be used to limit the number of taken branches that are fetched per clock cycle;
- the probability of a fetch redirection, which corresponds to a target misprediction (BTB miss) in conjunction with a correct taken/not-taken prediction for conditional branches; and
- the probability of a branch misprediction, which accounts for BTB misses for indirect branches and taken/not-taken mispredictions for conditional branches.

The *cache characteristics* consist of the following six probabilities: (i) the L1 I-cache miss rate, (ii) the L2 cache miss rate due to instructions only[1], (iii) the L1 D-cache miss rate, (iv) the L2 cache miss rate due to data accesses only, (v) the I-TLB miss rate and (vi) the D-TLB miss rate.

It is important to note that these characteristics are annotated to the corresponding edges in the SFG. Therefore branch characteristics are recorded for a particular branch with its history of preceding basic blocks. The same branch with a different history is stored separately. The same applies for the cache characteristics.

Note that characteristics related to locality events, such as branch and cache characteristics, are hard to model in a microarchitecture-independent way. Therefore we take a pragmatic approach and use characteristics for specific branch predictors and specific cache configurations. In our framework, we use functional simulation extended with branch predictors and cache structures to compute these locality events. Our tools are extended versions of SimpleScalar's `sim-bpred` and `sim-cache` [1]. Note that although this approach requires the simulation of the complete program execution for specific branch predictors and specific cache structures, this does not limit its applicability. Indeed, a number of tools exist that measure a wide range of these structures in parallel, e.g., the `cheetah` simulator [28] which is a single-pass multiple-configuration cache simulator.

Statistical profiling can be carried out using trace-driven tools operating on an execution trace that is stored on a disk. However, in cases where storing a large trace is impractical, an execution-driven tool can be used to measure the characteristics during functional simulation. We take the latter option in this paper.

2.1.3. Improving the branch characteristics. The tools that are used to measure the statistical profiles operate on an instruction-per-instruction basis. In particular, during the computation of the branch characteristics, the outcome of

1 We assume a unified L2 cache. However, we make a distinction between L2 cache misses due to instructions and due to data.

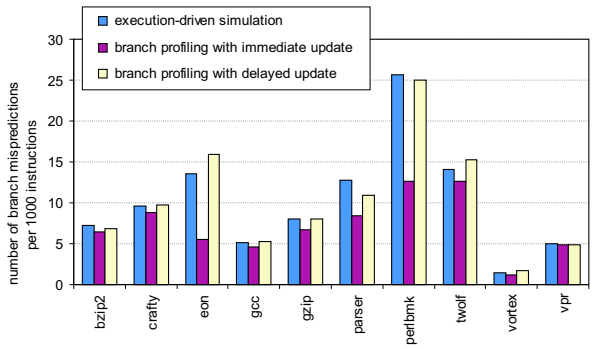

Figure 3. Number of branch mispredictions per 1,000 instructions under three scenarios: (i) execution-driven simulation, (ii) branch profiling with immediate update, and (iii) branch profiling with delayed update.

the previous branch is updated before the branch predictor is accessed for the current branch (immediate update). In pipelined architectures, however, this situation rarely occurs. Instead, multiple lookups to the branch predictor often occur between the lookup and the update of one particular branch. This is well known in the literature as *delayed update*. In a conservative microarchitecture the update occurs at commit time (at the end of the pipeline) whereas the lookup occurs at the beginning of the pipeline by the fetch engine. Delayed update can have a significant impact on overall performance. Therefore computer architects have proposed speculative update of branch predictors [11, 16, 26] with the predicted branch outcome instead of the resolved outcome. Speculative update can yield significant performance improvements because the branch predictor is updated earlier in the pipeline, for example at writeback time or at dispatch time. Note that speculative update requires a repair mechanism to recover from corrupted state due to mispredictions. In this paper, we assume the most aggressive speculative update mechanism available in SimpleScalar[2], namely at dispatch time, i.e., when instructions from the instruction fetch queue are inserted into the register update unit. It is interesting to note that speculative update mechanisms have been implemented in commercial microprocessors, for example in the Alpha 21264 [17].

Delayed update, even when using a speculative update mechanism, can have a significant impact on overall performance when modeling microprocessor performance. Therefore we propose a branch profiling approach that takes into account delayed update. This is done using a FIFO buffer in which lookup and update occur at the head and at the tail of the FIFO, respectively. The branch prediction lookups that are made when instructions enter the FIFO are based on 'stale' state that lacks updated information from branch instructions still residing in the FIFO. At each step of the algorithm, an instruction is inserted into the FIFO and removed from the FIFO. A branch predictor lookup occurs when a branch instruction enters the FIFO; an update occurs when a branch instruction leaves the FIFO. If a branch is mispredicted—this is detected upon removal—the instructions residing in the FIFO are squashed and new instructions are inserted until the FIFO is completely filled. As mentioned above, we assume speculative update at dispatch time. Therefore a natural choice for the size of the FIFO is the size of the instruction fetch queue. If other update mechanisms are used, such as speculative update at write-back time or non-speculative update at commit time, appropriate sizes should be chosen for the FIFO buffer.

To evaluate the benefits of this branch profiling approach, we refer to Figure 3 which shows the number of branch mispredictions per 1,000 instructions under various scenarios: (i) execution-driven simulation using SimpleScalar's `sim-outorder` simulator while assuming delayed update at dispatch time[3], (ii) branch profiling with immediate update after lookup, and (iii) the newly proposed branch profiling approach with delayed update. This graph shows that the new approach closely resembles the behavior that is observed during execution-driven simulation. In the evaluation section of this paper, we will show that this significantly improves the accuracy of statistical simulation.

2.2. Synthetic trace generation

Once a statistical profile is computed, we generate a synthetic trace that is a factor R smaller than the original program execution. R is defined as the *synthetic trace reduction factor*; typical values range from 1,000 to 100,000. Before applying our synthetic trace generation algorithm, we first generate a *reduced statistical flow graph*. This reduced SFG differs from the original SFG in that the occurrences of each node are divided by the synthetic trace reduction factor R. In other words, the occurrences in the reduced SFG N_i are a fraction R of the original occurrences M_i for all nodes i: $N_i = \lfloor \frac{M_i}{R} \rfloor$. Subsequently, we remove all nodes for which N_i equals zero. Along with this removal, we also remove all incoming and outgoing edges. In doing so, we obtain a reduced statistical flow graph that is no longer fully interconnected. However, the interconnection is still strong enough to allow for accurate performance predictions. Once the reduced statistical flow graph is computed, the synthetic trace is generated using the following algorithm.

2 The simulation environment that is used in this paper is SimpleScalar, see section 3.

3 See section 3 for details on the experimental setup concerning the processor configuration as well as the benchmarks.

1. If the occurrences of each of the nodes in the reduced statistical flow graph are zero, terminate the algorithm. Otherwise, generate a random number in the interval [0,1] and use this value to point to a particular node in the reduced statistical flow graph. Pointing to a node is not done in a uniform way but using a cumulative distribution function built up by the occurrence of each node. In other words, a node with a higher occurrence will be more likely to be selected than a node with a smaller occurrence.
2. Decrement the occurrence of the selected node reflecting the fact that this node has been accessed. Determine the current basic block corresponding to the node.
3. Assign the instruction types and the number of source operands of each of the instructions in the basic block.
4. For each source operand, determine its dependency distance. This is done using random number generation on the cumulative dependency distance distribution. Therefore an instruction x is made dependent on a preceding instruction $x - \delta$ with δ the dependency distance. Note that we do not generate dependencies that are produced by branches or stores since those types of instructions do not have a destination operand. This is achieved by trying a number of times until a dependency is generated that is not supposedly generated by a branch or a store. If after a maximum number of times (in our case 1,000 times) still no valid dependency is created, the dependency is simply squashed.
5. For each load in the synthetic trace, determine whether this load will cause a D-TLB hit/miss, an L1 D-cache hit/miss and in case of an L1 D-cache miss whether this load will cause an L2 cache hit/miss.
6. For the branch terminating the basic block, determine whether this is a taken branch and whether this branch is correctly predicted, results in a fetch redirection or is a branch misprediction.
7. For each instruction, determine whether this instruction will cause an I-TLB hit/miss, an L1 I-cache hit/miss, and, in case of an L1 cache miss, whether this instruction will result in an L2 cache miss.
8. Output the synthetically generated instructions along with their characteristics.
9. If the current node in the reduced statistical flow graph does not have outgoing edges, go to step 1, otherwise proceed. Generate a random number in the interval [0,1] and use this value to point a particular outgoing edge. This is done using a cumulative distribution built up by the transition probabilities of the outgoing edges. Use this outgoing edge to point to a particular node. Go to step 2.

2.3. Synthetic trace simulation

The trace-driven simulation of the synthetic trace is very similar to the trace-driven simulation of real program traces. In particular, for this paper, the synthetic trace simulator is a modified version of SimpleScalar's `sim-outorder` simulator in which a synthetic trace is fed into the simulator. The synthetic trace simulator does not need to model branch predictors nor caches. However, special actions are needed during synthetic trace simulation for the following cases.

- When a branch is mispredicted in an execution-driven simulator, instructions from an incorrect path are fetched and executed. When the branch is executed, it is determined whether the branch was mispredicted. In case of a misprediction, the instructions down the pipeline need to be squashed. A similar scenario is implemented in the synthetic trace simulator: when a mispredicted branch is fetched, the pipeline is filled with instructions from the synthetic trace as if they were from the incorrect path; this is to model resource contention. When the branch gets executed, the synthetic instructions down the pipeline are squashed and synthetic instructions are fetched as if they were from the correct path.
- For a load miss, the latency will be determined by whether this load is an L1 D-cache hit, an L1 D-cache miss, an L2 cache miss, or a D-TLB miss. For example, in case of an L2 miss, the access latency to main memory is assigned.
- In case of an I-cache miss, the fetch engine stops fetching for a number of cycles. The number of cycles is determined by whether the instruction causes an L1 I-cache miss, an L2 cache miss or a D-TLB miss.

The most important difference between the synthetic trace simulator and the reference execution-driven simulator, other than the fact that the former operates on synthetic traces, is that the synthetic trace simulator does not take into account instructions along misspeculated paths when accessing the caches. This can potentially have an impact on the performance prediction accuracy [2].

3. Experimental setup

The SPEC CINT2000 benchmarks[4] that are used in the evaluation of this paper are listed in Table 1. We have used the Alpha binaries from the SimpleScalar website.[5] The second column shows the inputs that were used

4 `http://www.spec.org`
5 `http://www.simplescalar.com`

instruction cache	8KB, 2-way set-associative, 32-byte block, 1 cycle access latency
data cache	16KB, 4-way set-associative, 32-byte block, 2 cycles access latency
unifi ed L2 cache	1MB, 4-way set-associative, 64-byte block, 20 cycles access latency
I-TLB and D-TLB	32-entry 8-way set-associative with 4KB pages
memory	150 cycle round trip access
branch predictor	8K-entry hybrid predictor selecting between an 8K-entry bimodal predictor and a two-level (8K x 8K) local branch predictor xor-ing the local history with the branch's PC, 512-entry 4-way set-associative BTB and 64-entry RAS
speculative update	at dispatch time
branch misprediction penalty	14 cycles
IFQ	32-entry instruction fetch queue
RUU and LSQ	128 entries and 32 entries, respectively
processor width	8 issue width, 8 decode width (fetch speed = 2), 8 commit width
functional units	8 integer ALUs, 4 load/store units, 2 fp adders, 2 integer and 2 fp mult/div units

Table 2. Baseline configuration.

benchmark	input	simpoints (weight)	IPC
bzip2	program	5 (20%), 6 (32%), 8 (48%)	1.83
crafty	ref	8 (100%)	0.51
eon	rushmeier	2 (100%)	0.81
gcc	integrate	9 (18%), 12 (4%), 17 (22%), 33 (9%), 53 (5%), 62 (18%), 88 (14%), 107 (6%)	1.37
gzip	graphic	4 (100%)	1.94
parser	ref	5 (55%), 13 (45%)	1.03
perlbmk	makerand	2 (100%)	0.97
twolf	ref	10 (100%)	0.64
vortex	lendian2	58 (100%)	1.11
vpr	route	72 (100%)	0.69

Table 1. The SPEC CINT2000 benchmarks used in this paper, their inputs, their simulation points with their corresponding weights, and the IPC for the baseline configuration.

for each benchmark. All these inputs are reference inputs. The third column shows the simulation points provided by SimPoint [25] along with their weights.[6] These simulation points are representative samples of 100M instructions. The main reason why we used these simulation points instead of the complete benchmark run is to limit the total simulation time. As will become clear in the evaluation section, a large number of simulations were run using detailed execution-driven simulation to validate the accuracy of the proposed statistical simulation approach. Running larger samples or complete benchmarks would have been too time-consuming. Note that this is exactly the problem we are addressing through statistical simulation. However, in section 4.4 we will evaluate whether statistical simulation is also accurate for larger sample sizes (1B and 10B instruction samples).

The baseline processor configuration is detailed in Table 2. We have used SimpleScalar/Alpha v3.0 [1]. The fourth column in Table 1 shows the baseline IPC over the SimPoint simulation points. For estimating the on-chip power consumption per cycle, we have used Wattch v1.02 [4] assuming a 0.18 μm-technology and a 1.2GHz clock frequency. We assume a base activity factor of 0.5 or random switching activity for single-ended array bitlines. Further, the most aggressive clock gating mechanism (`cc3`) is considered: a unit that is unused consumes 10% of its max power and a unit that is only used for a fraction x only consumes a fraction x of its max power.

4. Evaluation

In the evaluation of this statistical simulation approach we consider the following factors: (i) the simulation speed, (ii) the order k of the statistical flow graph, (iii) the usefulness of delayed update during branch profiling, (iv) the absolute accuracy for modeling performance and power consumption, (v) a comparison with HLS [23], (vi) modeling program phases and a comparison with SimPoint [25], (vii) the relative accuracy as a function of various architectural parameters, and (viii) the applicability for efficiently exploring huge design spaces.

4.1. Simulation speed

Due to the statistical nature of this technique, performance metrics converge to 'steady-state' values. To quantify the simulation speed of the statistical simulation approach we calculate the coefficient of variation (CoV) of the IPC as a function of the number of synthetic instructions. The CoV is defined as the standard deviation divided by the mean of the IPC over a given number of synthetic traces, in our case 20. The variation that is observed is due to the different random seeds that were used for the various synthetic traces. We clearly observe that the CoV decreases for longer synthetic traces and that small CoVs are obtained for small synthetic traces, e.g., 4% for 100K, 2% for 200K, 1.5% for 500K and 1% for 1M synthetic instructions. From these data we can conclude that statistical simulation is significantly faster than execution-driven simula-

6 `http://www.cs.ucsd.edu/~calder/simpoint` now provides new simulation points.

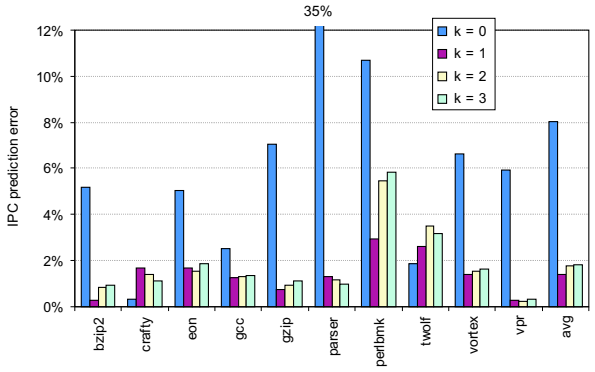

Figure 4. Evaluating the order k of the SFG; perfect caches and perfect branch prediction are assumed.

tion. In our setup (with 100M-instruction reference samples), we achieve a speedup of 100X to 1,000X. If larger instruction streams are considered (in section 4.4 we consider 10B instructions), even higher speedups are obtained: 10,000X to 100,000X.

4.2. Absolute accuracy

This section evaluates the absolute accuracy of the statistical simulation approach proposed in this paper. The absolute prediction error for a metric M is defined as

$$AE_M = \frac{|M_{SS} - M_{EDS}|}{M_{EDS}}$$

with M_{SS} and M_{EDS} computed through statistical simulation (SS) and execution-driven simulation (EDS), respectively. The metrics can be IPC (instructions retired per cycle) or EPC (energy consumption per cycle). We will use the absolute accuracy to evaluate the importance of using a statistical flow graph in our statistical profile. Subsequently, we will evaluate the importance of considering delayed updates during branch profiling. In the final subsection, we will evaluate the absolute accuracy of our method in estimating overall power/performance metrics.

4.2.1. Evaluating the statistical flow graph. Recall from section 2 that the order k of the SFG is yet to be defined. Figure 4 presents IPC prediction errors for various values of k under the assumption of perfect caches (each access is a hit) and perfect branch prediction (each branch is correctly predicted). These data show that $k = 0$ can result in large IPC prediction errors (up to 35%); if $k \geq 1$, the IPC predictions are significantly more accurate (less than 2% on average). Since $k = 1$ leads to predictions that are as accurate as $k = 2$ and $k = 3$, we will use $k = 1$ for the remainder of this paper. Table 3 presents the total number of nodes in the SFG as a function of its order k.

	$k=0$	$k=1$	$k=2$	$k=3$
bzip2	675	945	1,314	1,799
crafty	1,534	2,579	3,983	5,732
eon	466	645	836	1,028
gcc	30,834	43,157	57,031	71,879
gzip	291	434	632	863
parser	2,483	3,711	5,266	7,140
perlbmk	473	549	623	693
twolf	414	594	809	1,082
vortex	4,221	5,209	6,193	7,161
vpr	149	184	220	261

Table 3. The number of nodes in the SFG.

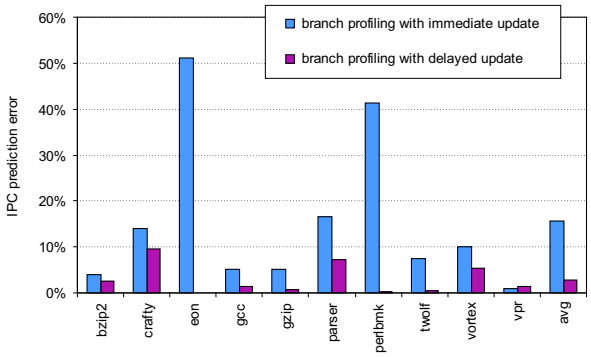

Figure 5. Evaluating the importance of modeling delayed update during branch profiling; perfect caches are assumed.

4.2.2. Evaluating branch profiling with delayed update. In section 2.1.3, we proposed a delayed update branch profiling technique. Figure 5 shows that modeling delayed update during branch profiling improves the IPC prediction accuracy. The benchmarks that benefit most are eon and perlbmk. Not surprisingly, these benchmarks showed the largest discrepancies in the number of branch mispredictions between execution-driven simulation and branch profiling with immediate update, as shown in Figure 3. Branch profiling with delayed update will be used for the remainder of this paper.

4.2.3. Overall power/performance prediction error. The left graph of Figure 6 presents IPC numbers obtained using our enhanced statistical simulation approach. For the baseline configuration, the average IPC prediction error is 6.6%; the maximum error is observed for parser (14.2%).

When the synthetic trace simulator is extended with an architectural power estimation tool, power consumption can be estimated using statistical simulation [9, 24]. The right graph of Figure 6 shows that statistical simulation estimates energy consumption per cycle (EPC) accurately. The average error is 4%; the largest error is observed for bzip2

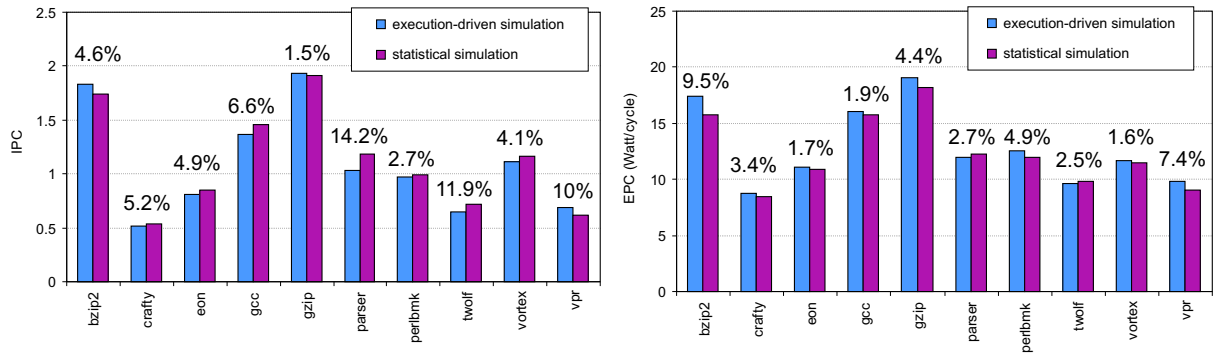

Figure 6. Execution-driven simulation versus statistical simulation for estimating IPC (on the left) and EPC (on the right).

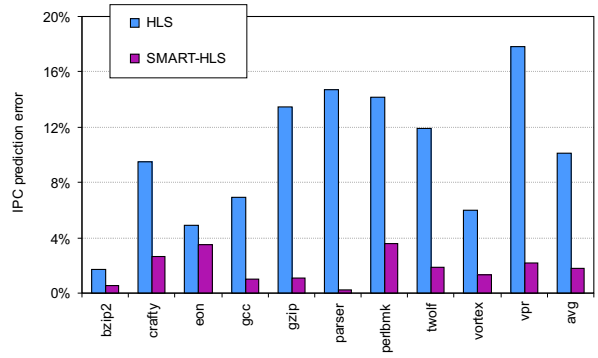

Figure 7. Comparing HLS to SMART-HLS, the statistical simulation framework presented in this paper.

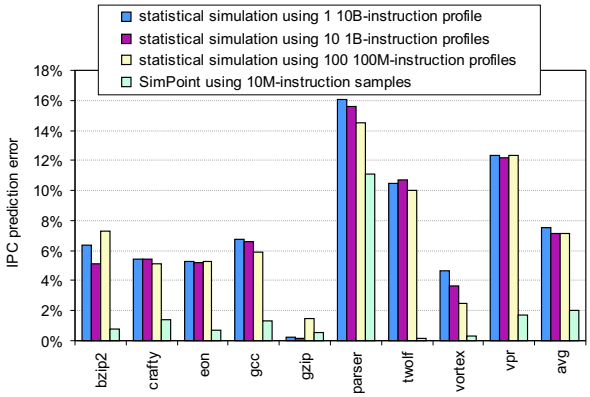

Figure 8. Evaluating the impact of modeling program phases and comparison with SimPoint.

(9.5%).

We have also considered the energy-delay product (EDP), which is an energy-efficiency metric that combines energy consumption with performance. EDP is defined as follows [3]: $EDP = EPC \cdot CPI^2 = EPC \cdot \frac{1}{IPC^2}$. The average EDP prediction error using statistical simulation is 11%; the largest error is observed for parser and twolf: 21% and 18%, respectively. Not surprisingly, these are the benchmarks with the highest IPC prediction errors, as shown in Figure 6 on the left.

4.3. Comparison with HLS

We now compare our statistical simulation framework to HLS as proposed by Oskin *et al.* [23]. The HLS synthetic trace simulator models an out-of-order architecture that is a simplification of SimpleScalar's model. HLS models the workload as a front-end graph structure, but the instructions in the graph are generated randomly from an instruction mix distribution without regard to the instruction sequences found in particular basic blocks. This is in contrast to the SFG proposed in the present work. The generalized HLS model was calibrated to match SimpleScalar's out-of-order simulator for one particular processor configuration, i.e., SimpleScalar's baseline configuration, as given in [23]. To allow for a fair comparison between HLS and the framework presented in this paper we have used SimpleScalar's baseline configuration instead of the configuration from Table 2. Figure 7 clearly shows that our framework, called SMART-HLS, is more accurate than HLS with an average error of 1.8% versus 10.1%.

4.4. Modeling program phases

It is well known that a computer program goes through various phases of execution [25]. In this section, we evaluate whether measuring separate statistical profiles and gen-

erating separate synthetic traces for each of these program phases yields more accurate performance predictions. For these experiments, we consider 10B instructions for each benchmark as our reference streams after skipping the first 1B instructions.[7] We consider the following scenarios:

- We apply statistical simulation over the complete reference stream, i.e., we generate one statistical profile and one synthetic trace to characterize the 10B instructions.
- We apply statistical simulation over each sample of 1B instructions. So, in total we have ten statistical profiles and ten synthetic traces. These ten synthetic traces are simulated and the performance metrics are averaged.
- We run statistical simulation on one hundred 100M-instruction samples.
- We use the SimPoint software [25] to compute representative 10M-instruction intervals from these 10B instruction streams. These 10M-instruction samples are then simulated through execution-driven simulation.

We draw a number of interesting conclusions from the results presented in Figure 8. First, applying statistical simulation to smaller samples only slightly improves accuracy, e.g., compare statistical simulation over one hundred 100M-instruction samples vs. statistical simulation over one 10B-instruction sample. Second, SimPoint is more accurate than statistical simulation. The average errors for SimPoint and statistical simulation are 2% and 7.2%, respectively. However, the number of simulated instructions for SimPoint is significantly larger than for statistical simulation. SimPoint simulates 20 million (crafty) to 300 million (gcc) instructions whereas statistical simulation only requires 1 million instructions *at the most*. In addition, SimPoint employs execution-driven simulation which is slower than synthetic trace simulation since the latter does not model caches nor branch predictors. In contrast to SimPoint however, statistical simulation needs to compute a new statistical profile when the cache or branch predictor is changed during a design space exploration. Nevertheless, statistical simulation will be much faster than SimPoint.

4.5. Relative accuracy

In prior sections we only considered absolute power/performance prediction accuracy, i.e., the error in one single design point. For a computer architect, relative accuracy or the ability to accurately predict a performance trend, is often more important. Indeed, the sensitivity of power and performance to a particular architectural parameter can help the designer identify the (near) optimal design point, e.g., on the 'knee' of the performance curve, or where performance begins to saturate as a function of a given architectural parameter. To evaluate statistical simulation in this perspective we have measured the relative accuracy as a function of five architectural parameters: window size, processor width, instruction fetch queue size, branch predictor size and cache size. The relative prediction error for a metric M when moving from design point A to design point B is defined as

$$RE_M = \frac{|M_{B,SS}/M_{A,SS} - M_{B,EDS}/M_{A,EDS}|}{M_{B,EDS}/M_{A,EDS}}.$$

Table 4 shows the relative prediction errors averaged over the various benchmarks. This table not only presents sensitivity of IPC and EPC to a given architectural parameter, but also sensitivity of other metrics, such as the RUU occupancy, the LSQ occupancy, the IFQ occupancy, the fetch unit's energy consumption, the dispatch unit's energy consumption, etc. An accurate estimate of those trends is particularly relevant for a designer who wants to ensure that the various parameters are tuned properly to optimize performance. The results in Table 4 show that the average relative prediction errors are generally smaller than 3%.

4.6. Design space exploration

Statistical simulation can be used to efficiently explore large design spaces. In spite of the absolute errors obtained when estimating EDP (see section 4.2.3), a region of energy-efficient designs can be identified through statistical simulation. To demonstrate this we have set up the following experiment. We computed the energy-delay product for a large design space using statistical simulation by varying the size of the RUU (8,16,32,48,64,96,128), the size of the LSQ[8] (4,8,16,24,32,48,64), the decode width (2,4,6,8), the issue width (2,4,6,8) and the commit width (2,4,6,8). The total number of design points in this experiment is 1,792. These 1,792 design points are all evaluated through statistical simulation and the design point with optimal EDP is identified. To verify that statistical simulation indeed identifies a region of optimal design points, we have computed the EDP for the design points that were in a 3% range of the optimal design point. For 7 out of the 10 benchmarks, statistical simulation indeed identified the optimal design. For the remaining three benchmarks, statistical simulation identified a design that is in a very short range to the optimal design: gzip (0.03%), eon (1.03%) and vpr (1.24%).

7 perlbmk was excluded from these experiments because we had problems simulating it for such a large instruction count.

8 We limit the LSQ size not to be larger than the RUU size.

Sensitivity to window size (the RUU size is varied from 8 to 128; the LSQ size is half the RUU size)						
	8 → 16	16 → 32	32 → 48	48 → 64	64 → 96	96 → 128
IPC	1.0%	1.7%	1.2%	0.7%	0.6%	1.3%
RUU occupancy	0.4%	1.8%	2.3%	1.9%	3.6%	3.2%
LSQ occupancy	0.6%	1.9%	2.3%	2.0%	3.7%	2.9%
EPC	0.6%	1.0%	0.6%	0.6%	1.0%	0.7%
RUU power consumption	0.7%	1.3%	0.8%	0.8%	0.8%	0.9%
LSQ power consumption	0.4%	0.7%	0.4%	0.3%	0.6%	0.5%

Sensitivity to processor width (decode width = issue width = commit width)			
	2 → 4	4 → 6	6 → 8
IPC	1.7%	1.2%	0.8%
execution bandwidth	1.5%	2.1%	1.6%
EPC	1.6%	1.1%	0.4%
fetch unit power consumption	0.8%	0.7%	0.4%
dispatch unit power consumption	1.1%	1.6%	1.1%
issue unit power consumption	1.6%	1.3%	0.5%

Sensitivity to the instruction fetch queue (IFQ) size			
	4 → 8	8 → 16	16 → 32
IPC	1.3%	0.8%	0.9%
EPC	0.9%	1.1%	0.5%
IFQ occupancy	3.2%	5.0%	6.4%

Sensitivity to the branch predictor size				
	base ÷ 4 → base ÷ 2	base ÷ 2 → base	base → base · 2	base · 2 → base · 4
IPC	0.5%	0.5%	0.7%	0.4%
EPC	0.5%	0.5%	0.5%	0.6%
RUU occupancy	0.8%	0.7%	0.7%	0.6%
RUU power consumption	0.4%	0.4%	0.6%	0.3%
LSQ occupancy	0.8%	0.5%	0.7%	0.4%
LSQ power consumption	0.2%	0.2%	0.3%	0.2%
IFQ occupancy	0.6%	0.6%	0.8%	0.6%
fetch unit power consumption	0.4%	0.3%	0.5%	0.5%
branch predictor power consumption	0.3%	1.4%	1.2%	0.2%

Sensitivity to the cache confi guration size				
	base ÷ 4 → base ÷ 2	base ÷ 2 → base	base → base · 2	base · 2 → base · 4
IPC	2.2%	1.4%	3.3%	2.6%
EPC	1.3%	1.3%	1.7%	4.0%
RUU occupancy	4.6%	1.6%	3.6%	2.0%
RUU power consumption	1.3%	1.0%	1.5%	1.1%
LSQ occupancy	3.9%	2.0%	3.2%	3.9%
LSQ power consumption	0.7%	0.6%	1.0%	0.7%
IFQ occupancy	5.6%	7.3%	8.9%	8.5%
fetch unit power consumption	1.0%	0.7%	0.9%	1.2%
I-cache power consumption	1.5%	1.2%	2.1%	2.4%
D-cache power consumption	6.8%	7.2%	9.3%	6.0%
L2 cache power consumption	0.4%	0.2%	0.4%	0.3%

Table 4. Relative error of statistical simulation as a function of window size, processor width, instruction fetch queue size, branch predictor size and cache size.

5. Related work

Noonburg and Shen [21] present a framework that models the execution of a program on a particular architecture as a Markov chain, in which the state space is determined by the microarchitecture and in which the transition probabilities are determined by the program execution. This approach was evaluated for in-order architectures. Extending it for wide-resource out-of-order architectures would result in a far too complex Markov chain.

Hsieh and Pedram [13] present a technique to estimate performance and power consumption of a microarchitecture by measuring a characteristic profile of a program execution, synthesizing a fully functional program from it, and simulating this synthetic program on an execution-driven simulator. The main disadvantage of their approach is the fact that no distinction is made between microarchitecture-dependent and microarchitecture-independent characteristics. All characteristics are microarchitecture-dependent, which makes this technique unusable for design space explorations.

Iyengar *et al.* [15] present SMART to generate representative synthetic traces based on the concept of a fully qualified basic block. A fully qualified basic block is a basic block together with its *context*. The context of a basic block is determined by its n preceding qualified basic blocks—a qualified basic block is a basic block together with the branching history (of length k) of its preceding branch. This work was later extended in [14] to account for cache behavior. In this extended work the focus was shifted from fully qualified basic blocks to fully qualified

instructions. The context of a fully qualified instruction is then determined by n singly qualified instructions. A singly qualified instruction is an instruction annotated with its instruction type, its I-cache behavior, and, if applicable, its D-cache behavior and its branch behavior. Therefore a distinction is made between two fully qualified instructions having the same preceding instructions, except that, in one case, a preceding instruction missed in the cache, whereas in the other case it did not. Obviously, collecting all these fully qualified instructions during profiling results in a huge amount of data to be stored in memory. For some benchmarks, the authors report that the amount of memory that is needed can exceed the available memory in a machine, so that some information needs to be discarded from the graph. The statistical simulation framework presented in this paper shares the concept of using a context by qualifying a basic block with its preceding basic blocks. However, the statistical flow graph that is built for this purpose is both simpler and smaller than the fully qualified one used in SMART. In addition, we have found that qualifying with one single basic block is sufficient. Another interesting difference between SMART and the framework presented here is the fact that SMART generates memory addresses during synthetic trace generation. We simply assign hits and misses.

In recent years, a number of papers [5, 8, 9, 10, 22, 23, 24] have been published that are built around (slightly different forms of) the general statistical simulation framework presented in Figure 1. We identify one major difference between these approaches and the present work related to the degree of correlation in the statistical profile. The simplest way to build a statistical profile is to assume that all characteristics are independent from each other [5, 8, 9, 10], which results in the smallest statistical profile and the fastest convergence time but potentially the largest prediction errors. In HLS, Oskin *et al.* [23, 24] generate one hundred basic blocks of a size determined by a normal distribution over the average size found in the original workload. The basic block branch predictabilities are statistically generated from the overall branch predictability obtained from the original workload. Instructions are assigned to the basic blocks randomly based on the overall instruction mix distribution, in contrast to the *basic block modeling granularity* of the SFG. As in the present work, the HLS synthetic trace generator then walks through the graph of instructions. Nussbaum and Smith [22] propose to correlate various characteristics such as the instruction types, the dependencies, the cache behavior and the branch behavior to the size of the basic block. Using the size of the basic block to correlate statistics raises the possibility of *basic block size aliasing*, in which statistical distributions from basic blocks with very different characteristics are combined and reduce simulation accuracy. In a SFG, all characteristics are correlated to the basic block itself, not just its size. Moreover, we correlate basic blocks on previously executed basic blocks by using higher order ($k \geq 1$) SFGs, i.e., basic blocks with a different history of executed basic blocks are characterized separately.

Intuitively, the framework presented in this paper combines SMART with previously proposed statistical simulation approaches to combine the benefits and to eliminate the drawbacks of both techniques. The major benefit gained from SMART is the accurate modeling of instruction sequences and their dependencies; this is achieved by considering basic blocks along with their context, i.e., we statistically model at the granularity of the basic block. The drawback that is eliminated from SMART is the explosion of state (and thus memory) that is needed to keep track of all the qualified instructions. The major benefit that is gained from statistical simulation is its simplicity. The major drawback that is eliminated from previously proposed statistical simulation approaches is their inability to accurately model instruction sequences and their dependencies.

6. Conclusion

Architectural simulations are extremely time-consuming and often impact the time-to-market of newly designed microprocessors. One possible approach to this problem is to use statistical simulation as an accurate and efficient complement to detailed simulation. The statistical simulation approach presented in this paper has two major contributions. First, the use of the statistical flow graph (SFG) for statistical simulation combines the benefits of the previously proposed graph representation in SMART—accuracy—with features from previously proposed statistical simulation frameworks—simplicity and rapid convergence. Second, we have shown that it is important to model delayed update of branch predictors during statistical profiling. This improved statistical simulation framework was extensively evaluated. First, we show that statistical simulation is indeed a fast simulation technique, i.e., the synthetic traces can be very short (100K to 1M instructions). Second, our measurements show that the performance and energy consumption of an 8-issue out-of-order superscalar architecture for SPECint2000 benchmarks can be predicted with an average error of only 6.6% and 4%, respectively. Third, we show that our approach is significantly more accurate than the previously proposed HLS statistical simulation framework. A comparison with the SimPoint sampling technique shows that SimPoint is more accurate, but that statistical simulation is faster. We also show that relative accuracy, the ability to predict performance trends, using statistical simulation is very high; the relative error is generally below 3%. And finally, we show that statistical simulation can be used to identify energy-efficient microarchitectures in a large design space.

Acknowledgements

The authors would like to thank Brad Calder, Jim Smith and the anonymous reviewers for their detailed feedback. Lieven Eeckhout is Postdoctoral Fellow of the Fund for Scientific Research—Flanders (Belgium) (F.W.O. Vlaanderen). This research is also partially supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), by Ghent University, by the United States National Science Foundation under grant number 0113105, and by IBM, Intel, and AMD Corporations.

References

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

[2] C. Bechem, J. Combs, N. Utamaphetai, B. Black, R. D. S. Blanton, and J. P. Shen. An integrated functional performance simulator. *IEEE Micro*, 19(3):26–35, May/June 1999.

[3] D. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, November/December 2000.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA-27*, pages 83–94, June 2000.

[5] R. Carl and J. E. Smith. Modeling superscalar processors via statistical simulation. In *PAID-98, in conjunction with ISCA-25*, June 1998.

[6] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *ICCD-96*, pages 468–477, Oct. 1996.

[7] P. K. Dubey, G. B. Adams III, and M. J. Flynn. Instruction window size trade-offs and characterization of program parallelism. *IEEE Transactions on Computers*, 43(4):431–442, Apr. 1994.

[8] L. Eeckhout and K. De Bosschere. Early design phase power/performance modeling through statistical simulation. In *ISPASS-2001*, pages 10–17, Nov. 2001.

[9] L. Eeckhout and K. De Bosschere. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *PACT-2001*, pages 25–34, Sept. 2001.

[10] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. De Bosschere. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, Sept/Oct 2003.

[11] E. Hao, P.-Y. Chang, and Y. N. Patt. The effect of speculatively updating branch history on branch prediction accuracy, revisited. In *MICRO-27*, pages 228–232, Nov. 1994.

[12] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[13] C. Hsieh and M. Pedram. Micro-processor power estimation using profile-driven program synthesis. *IEEE TCAD*, 17(11):1080–1089, Nov. 1998.

[14] V. S. Iyengar and L. H. Trevillyan. Evaluation and generation of reduced traces for benchmarks. Technical Report RC 20610, IBM Research Division, T. J. Watson Research Center, Oct. 1996.

[15] V. S. Iyengar, L. H. Trevillyan, and P. Bose. Representative traces for processor models with infinite cache. In *HPCA-2*, pages 62–73, Feb. 1996.

[16] S. Jourdan, J. Stark, T.-H. Hsing, and Y. N. Patt. Recovery requirements of branch prediction storage structures in the presence of mispredicted-path execution. *International Journal of Parallel Programming*, 25(5):363–383, Oct. 1997.

[17] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 microprocessor architecture. In *ICCD-98*, pages 90–95, Oct. 1998.

[18] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1(2):10–13, June 2002.

[19] S. S. Mukherjee, S. V. Adve, T. Austin, J. Emer, and P. S. Magnusson. Performance simulation tools: Guest editors' introduction. *IEEE Computer, Special Issue on High Performance Simulators*, 35(2):38–39, Feb. 2002.

[20] D. B. Noonburg and J. P. Shen. Theoretical modeling of superscalar processor performance. In *MICRO-27*, pages 52–62, Nov. 1994.

[21] D. B. Noonburg and J. P. Shen. A framework for statistical modeling of superscalar processor performance. In *HPCA-3*, pages 298–309, Feb. 1997.

[22] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *PACT-2001*, pages 15–24, Sept. 2001.

[23] M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor design. In *ISCA-27*, pages 71–82, June 2000.

[24] R. Rao, M. H. Oskin, and F. T. Chong. HLSPOWER: Hybrid Statistical Modeling of the Superscalar Power-Performance Design Space. In *HiPC-9*, pages 620–629, Dec. 2002.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X*, pages 45–57, Oct. 2002.

[26] K. Skadron, M. Martonosi, and D. W. Clark. Speculative updates of local and global branch history: A quantitative analysis. *Journal of Instruction-Level Parallelism*, 2, Jan. 2000. `http://www.jilp.org/vol2`.

[27] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic evaluation of shared-memory systems with ILP processors. In *ISCA-25*, pages 380–391, June 1998.

[28] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *SIGMETRICS'93*, pages 24–35, 1993.

[29] R. E. Wunderlich, T. F. Wenish, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA-30*, June 2003.