Deconstructing and Improving Statistical Simulation in HLS

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

[†]Department of Electrical and Computer Engineering
The University of Texas at Austin
{belljr, ljohn}@ece.utexas.edu

[‡]Department of ELIS
Ghent University, Belgium
leeckhou@elis.ugent.be

Abstract

Statistical simulation systems can provide an accurate and efficient way to carry out early design studies for processors. One such system, HLS, has a rapid simulation capability, but our experiments demonstrate that several modeling improvements are possible. The front-end graph structure in HLS is hampered by workload modeling at the instruction level that reduces the accuracy of program simulation. The workload and processor models require significant changes to provide accurate results for a variety of benchmarks. We improve HLS by modeling the workload at the granularity of the basic block and by changing the processor model to more closely reflect components in modern microprocessors. The specific techniques improve HLS accuracy by a factor of 3.78 at the cost of increased storage and runtime requirements.

Our examination of HLS points to a pitfall for simulator developers: reliance on a single small set of benchmarks to qualify a simulation system. A simple regression model shows that the SPECint95 benchmarks, the original benchmarks used to calibrate HLS, have characteristics that yield to very simple modeling.

1. Introduction

To address the extremely long simulation times of modern processor designs, researchers have developed *statistical simulation* systems [2-5, 7, 8]. Statistical simulation uses workload statistics from specialized functional or trace-driven simulation to create a synthetic trace that is applied to a fast and flexible execution engine. In HLS [8], statistics are used to create a static control flow graph of a small number of statistically generated instructions. The graph is then walked and the instructions are simulated in a processor model. Since the number of instructions is small and their workload characteristics have been determined by a statistical distribution, the simulation converges to a result much faster than cycle-accurate simulations.

The workload statistics include *microarchitecture-independent* characteristics such as instruction mix and inter-instruction dependency frequencies. They also include *microarchitecture-dependent* statistics such as branch prediction accuracy and cache miss rates for specific branch predictor and cache configurations. These are used to model locality structures dynamically as the simulation proceeds.

Statistical simulation systems that correlate well with execution-driven simulators have been shown to exhibit good *relative* accuracy as microarchitecture changes are applied in design studies [3]. Studies have achieved average errors smaller than 5% on specific benchmark suites [4, 8]. In this study, we quantify the correlation of HLS over a range of benchmarks, from general-purpose applications to technical and scientific benchmarks, and streaming kernels. In addition to the SPEC95 benchmarks [12], we study single-precision versions of the STREAM and STREAM2 benchmarks [13]. On these benchmark suites, we find that HLS has an average error of 15.5%.

The purpose of this study is to investigate exactly why HLS is not more accurate. Simultaneously we work to improve HLS. We enhance the workload model by collecting information at the basic block level instead of at the instruction level, and we add more detail to the processor model. We find that the overall error decreases from 15.5% to 4.1%, a factor of 3.78. We use the same basic block simulation techniques as in [4], so the error is similar. However, in this study, we start with the HLS framework as a base and in-

crementally add modeling detail to uncover the additional complexity necessary to improve HLS. We quantify the cost of the improvements in terms of additional storage requirements.

A simple regression model indicates that CPI results for the SPECint95, the benchmarks originally used to calibrate HLS, can yield to very simple modeling. Our analysis points to a larger problem for simulator developers: using a small set of benchmarks, datasets and simulated instructions to calibrate a simulation system.

In the next section, we describe HLS. In Section 3, we describe various modeling problems that we found in HLS. In Section 4, we investigate improvements to the system. We quantify the costs of the improvements in Section 5, followed by conclusions and references.

2. HLS Overview

In the HLS system [8], machine-independent characteristics are analyzed using a modified version of the *sim-fast* functional simulator from the SimpleScalar release 2.0 toolset [1]. An instruction mix frequency distribution is generated that consists of the percentages of integer, float, load, store and branch instructions. The mean basic block size and standard deviation are also computed.

Also generated is the frequency distribution of the dependency distances between instructions for each input of the five instruction types. The benchmarks are executed for one billion cycles in *sim-outorder* [1]. *Sim-outorder* provides the IPC used to compare against the IPC obtained in HLS statistical simulation. It also computes the L1 I-cache and D-cache miss rates, the unified L2 cache rate, and the branch predictability. After the workload is characterized, HLS generates one hundred basic blocks using a normal random variable over the mean block size and standard deviation. A uniform random variable over the instruction mix distribution fills in the instructions of each basic block.

For each randomly generated instruction, a uniform random variable over the dependency distance distribution generates a dependency for each instruction input. An effort is made to make an instruction independent of a store within the current basic block, but if the dependency stretches beyond the limits of the current basic block, no change is made because the dynamic predecessor is not known.

The basic blocks are connected into a graph structure. Each branch has both a taken pointer and a not-taken pointer to other basic blocks. The percentage of backward branches, set statically to 15% in the code, determines whether the taken pointer is a backward branch or a forward branch. For backward or forward branches, a normal random variable over either the mean backward or forward jump distances (set statically to ten and three in the code, respectively) determines the taken target. Later, during simulation, normal random variables over the branch predictability obtained from the *sim-outorder* run determine dynamically if the branch is actually taken or not, and the corresponding branch target pointer is followed.

After the machine statistics are processed and the basic blocks are configured, the instruction graph is walked. As each instruction is encountered, it is simulated on a generalized superscalar execution model for ten thousand cycles. The IPC is averaged over twenty simulations. The generalized model contains fetch, dispatch, execution, completion, and writeback stages. Fetches are buffered up to the fetch width of the machine. Instructions are dispatched to issue queues in front of the execution units and executed as their dependencies are satisfied. Neither an issue width nor a commit width is specified in the processor model. In HLS, the procedure is to first calibrate the generalized processor model using a test workload; then a reference workload is executed on the model.

For loads, stores, and branches, the locality statistics determine the necessary delay before issue of dependent instructions. To provide comparison with the SimpleScalar *lsq*, loads and stores are serviced by a single queue. Parallel cache miss operations are provided through the two memory ports available to the load-store execution unit. As in SimpleScalar, stores execute in zero-time when they reach the tail of their issue queue and the execution unit is available.

3. Issues in HLS

In this section, we first describe the experimental setup and benchmarks used in our experiments, followed by our examination of HLS, including descriptions of several workload and processor modeling issues.

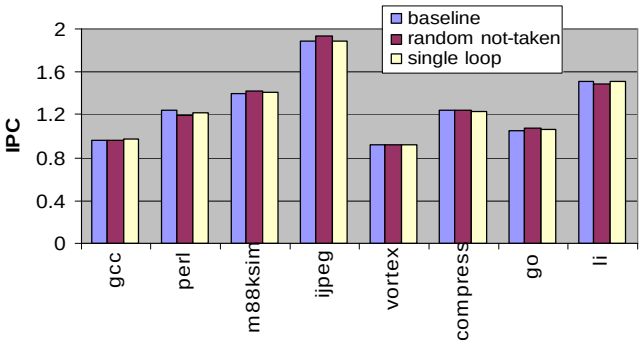

Figure 1: Effect of Graph Connectivity Changes

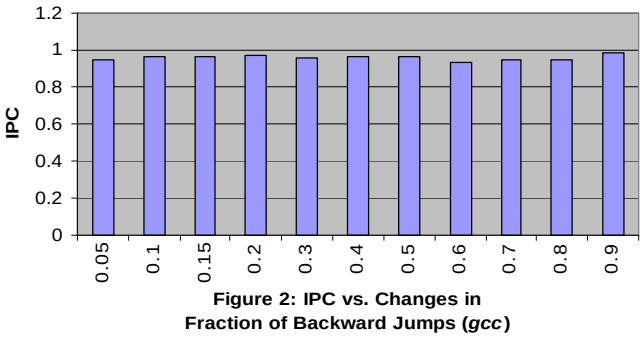

Figure 2: IPC vs. Changes in Fraction of Backward Jumps (*gcc*)

3.1. Experimental Setup and Benchmarks

For our experiments we follow the procedure in [8] using the software available at [9]. SimpleScalar and the statistical simulation software were compiled to target big-endian PISA binaries on an IBM Power3 p270. Using the default parameters in [8], *sim-outorder* was executed on the SPECint95 binaries found at [11] for up to one billion instructions of one reference input dataset, as in [8]. The modified *sim-fast* was executed on the input dataset for fifty billion instructions, to approximate complete program simulation.

In these experiments we use the SPEC CPU 95 integer benchmarks [12] for direct comparison with the original HLS results. We add the SPEC CPU 95 floating point benchmarks [12] and single-precision versions of the STREAM and STREAM2 benchmarks [6, 13]. We include this last suite of benchmarks because they are particularly challenging to statistical simulation systems. In Section 2.5 we discuss the characteristics of the STREAM benchmarks in more detail.

3.2. The HLS Graph Structure

We first examine the HLS front-end graph structure. We vary the percentages of backward branches, the backward branch jump distance, the forward branch jump distance, and the graph connections themselves.

Figure 1 shows the effect of varying the front-end graph connectivity. *Baseline* is the base HLS system running with the taken and not-taken branches connected as described in Section 2. *Random not-taken* is the base system with the not-taken target randomly selected from the configured basic blocks. *Single loop* is the base system with the taken and not-taken targets of each basic block both pointing to the next basic block in the sequence of basic blocks, with the last basic block pointing back to the first. The maximum error versus the base system is 3.6% for *perl* using the *random not-taken* strategy. This is well below the average HLS correlation error versus the SimpleScalar.

Figure 2 shows the IPC for *gcc* as the fraction of backward jumps changes. The hard-coded HLS default is 15% backward jumps. The maximum error versus that default is 2.8%. Figure 3 shows IPC as the backward and forward jump distances are changed from a default of ten and three, respectively. The maximum error versus either of those is 2.0%.

From these figures, it is apparent that the graph connectivity in HLS has no impact on simulation performance. Intuitively, HLS models the workload at the *granularity of the instruction.* All instructions in all basic blocks in the graph are generated identically. The instruction type and dependencies assigned to any slot in any basic block in the graph is randomly selected from the global instruction mix distribution, so the instruction found at any slot on a jump is just as likely to be found at any other slot.

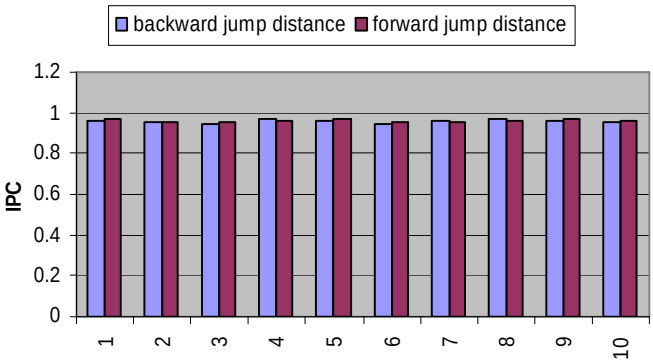

Figure 3: IPC vs. Changes in Backward/Forward Jump Distance (gcc)

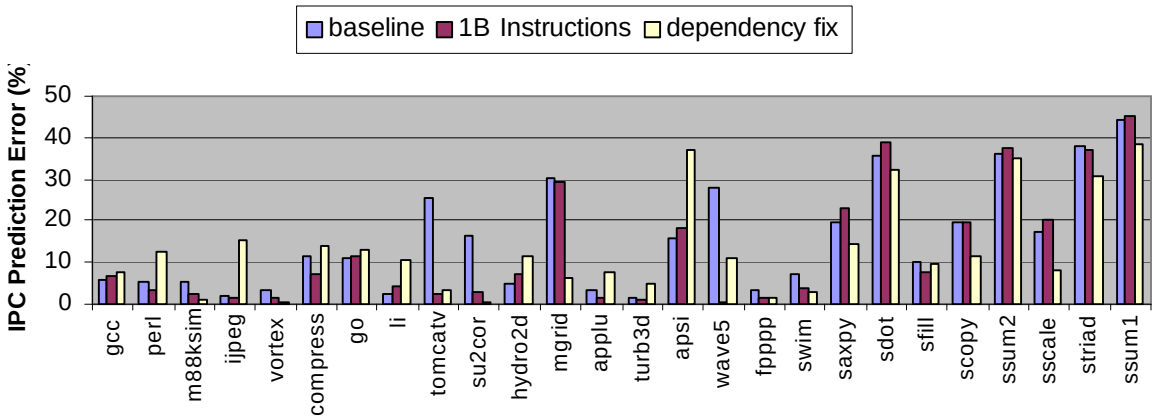

Figure 4: HLS Error as Modeling Changes

There is also a small probability that the random graph connectivity causes skewed results because the randomly selected *taken* targets can form a small loop of basic blocks, effectively pruning other parts of the graph from the simulation. This is not a major problem for HLS, in which all blocks are essentially the same, but it has implications for our improvements to HLS described below, so the single loop strategy is employed for the remainder of this paper.

3.3. The HLS Processor Model

In the HLS generalized execution model, there is no issue-width concept. The issue of instructions to the issue queues is instead limited by the queue size and dispatch window and, ultimately, by the fetch window. There is also no specific completion width in HLS, so the instruction completion rate is also front-end limited. These omissions are conducive to obtaining quick convergence to an average result for well-behaved benchmarks, but they make it difficult to correlate the system to SimpleScalar for a variety of benchmarks.

3.4. Modeling Workload Characteristics

Figures 4 and 5 show the IPC prediction error [4] over all benchmarks as workload modeling issues are incrementally addressed. The *baseline* run gives the HLS results out-of-the-box. While SPECint95 does well as in [8] with only 5.8% error, SPECfp95 has twice the correlation error. The STREAM loop error is more than four times worse at 27%. We were unable to achieve accurate results on all the benchmarks by recalibrating the generalized HLS processor model.

Recall that, in standard HLS, measuring microarchitecture-independent characteristics is carried out on the complete benchmark using *sim-fast*, whereas microarchitecture-dependent locality metrics are obtained only for the first one billion instructions using *sim-outorder*. It stands to reason that workload information and locality information should be collected over the same cycle ranges. The *1B Instructions* run gives results with *sim-fast* executing the same one billion instructions as *sim-outorder*. Not all benchmarks improve, but the error in SPECfp95 drops by half from 13.6% to 6.8%. Overall error decreases from 15.5% to 13.1%.

The modified *sim-fast* makes no distinction between memory instructions that carry out auto-increment or auto-decrement on the address register after memory access and those that do not. The HLS *sim-fast* code always assumes the modes are active. This causes the code to assume register dependencies that do not actually exist between memory access instructions, and it makes codes with significant numbers of load and store address

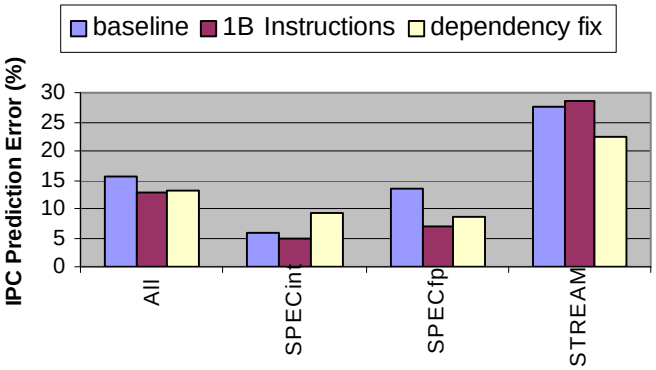

Figure 5: Overall HLS Error as Modeling Improves

Table 1: CPI Regression Analysis over 1B Instructions		
Benchmarks	**Targeted CPI**	**R^2**
SPECint	HLS	0.988
	SimpleScalar	0.970
SPECint and SPECfp	HLS	0.972
	SimpleScalar	0.895
SPECint, SPECfp and STREAM	HLS	0.757
	SimpleScalar	0.811

register dependencies, including the STREAM loops, appear to run slower. The *sim-fast* code was modified to check the instruction operand for the condition and mark dependencies accordingly, and the *dependency fix* bars in the figures give the results. The STREAM loops are improved, but the SPECint95 error increases from 4.8% to 9.3%. This is most likely due to the original calibration of the generalized HLS processor model in the presence of the modeling error.

Table 1 shows a simple regression analysis over the locality features taken from *sim-outorder* runs: branch mispredictability, L1 I-cache and D-cache miss rates, and L2 miss rate. The *targeted CPI* is the particular CPI targeted in the analysis, either SimpleScalar or the HLS result. The squared correlation coefficient, R^2, is a measure of the variability in the CPI that is predictable from the four features. The SPECint95 benchmarks always achieve high correlation, while the analysis over all benchmarks or even over SPECint95 together with SPECfp95 achieve lower correlation. This is an indication that a very simple processor model can potentially represent the CPI of the SPECint95 by emphasizing the performance of the locality features; but it can not as easily do the same over all three suites.

Table 2: The STREAM Loops		
Benchmark	**Equation**	**Loop Instructions**
saxpy	z[k] = z[k] + q * x[k]	10
sdot	q = q + z[k] * x[k]	9
sfill	z[k] = q	5
scopy	z[k] = x[k]	7
ssum2	q = q + x[k]	6
sscale	z[k] = q * x[k]	8
striad	z[k] = y[k] + q * x[k]	11
ssum1	z[k] = y[k] + x[k]	10

3.5. Loop Challenges

Table 2 shows single-precision versions of the STREAM benchmarks, including the loop equation and the number of instructions in the kernel loop when compiled with *gcc* using *-O*. The STREAM loops are strongly phased, and in fact have only a single phase.

Loops consist of one or a small number of tight iterations containing specific instruction sequences that are difficult for statistical simulation systems to model. Figure 6 shows one iteration of the *saxpy* loop (in the *PISA* language [1]). If the *mul.s* and *add.s* were switched in the random instruction generation process leaving the dependency relationships the same, the extra latency of the multi-cycle *mul.s* instruction is no longer hidden by the latency of the second *l.s*, leading to a generally longer execution time for the loop. A similar effect can be caused by changes in dependency relationships as the dependencies are statistically generated from a distribution.

Shorter runs can also occur. The *mul.s* has a dependency on the previous *l.s*. If the *l.s* is switched with the one-cycle *add.s*, keeping dependencies the same, the *mul.s* can dispatch much faster. While higher-order ILP distributions might work well for some loops, the results have been mixed and can actually lead to decreased accuracy for general-purpose programs [3].

4. Improving HLS

In this section, we focus on improving the processor and workload models to give more accurate simulation results.

4.1. Processor Model

It is difficult to correlate the generalized HLS processor model to SimpleScalar for all benchmarks. For this reason, we augmented HLS with a register-update-unit (RUU), an issue width

```
start:  addu $2, $3, $6
        l.s $f2, 0($2)
        mul.s $f2, $f4, $f2
        l.s $f0, 0($3)
        add.s $f2, $f2, $f0
        addiu $4, $4, 1
        slt $2, $5, $4
        s.s $f2, 0($3)
        addiu $3, $3, 4
        beq $2, $0, start
```

Figure 6: Disassembled *SAXPY* Loop

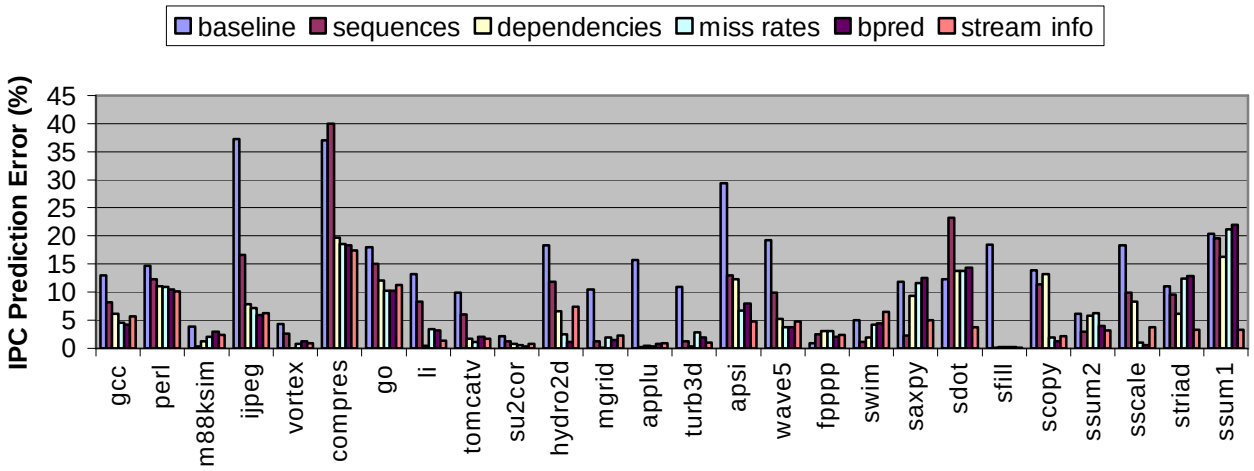

Figure 7: Improved HLS Error as Modeling Changes

and a completion width. We also rewrote the recurrent completion function to be non-recurrent and callable prior to execution, and we rewrote the execution unit to issue new instructions only after prior executing instructions have been serviced in the current cycle. We added code to differentiate long and short running integer and floating point instructions. To maintain efficiency, the locality structures are still modeled using the statistical parameters taken from *sim-outorder* runs.

We first run the benchmarks on the improved processor model using the same workload characteristics modeled in HLS, except that we generate one thousand basic blocks instead of one hundred, and we simulate for twenty thousand cycles in stead of ten thousand; so simulation time is about twice that in HLS. (The same changes in HLS do not decrease error.) The execution engine flow, delays and parameters are all chosen to match those in the SimpleScalar default configuration. The baseline system was validated by comparing *sim-outorder* traces, obtained from sections of the STREAM loops, to traces taken from the improved HLS assuming perfect caches and perfect branch predictability. The validation was simplified by the fact that the loops are comprised of only one phase.

Figure 7 gives the results for the individual benchmarks, and Figure 8 shows the average results per benchmark suite. The *baseline* run gives the improved system results using the default SimpleScalar parameters and using the global instruction mix, dependency information, and load and store miss rates. There are errors greater than 25% for particular benchmarks, such as *ijpeg*, *compress* and *apsi*. The overall error of 14.4% compares well with the 15.5% baseline error in HLS, but it is higher than the 13.1% error in shown in Figure 5 for HLS with improved workload modeling.

4.2. Workload Model

We also enhanced the workload model to reduce correlation errors. The analysis of the graph structure showed that modeling at the granularity of the instruction in HLS did not contribute to accuracy. In [7], the basic block size is the granule of simulation. However, this raises the possibility of *basic block size aliasing*, in which many blocks of the same size but very different instruction sequences and dependency relationships are combined.

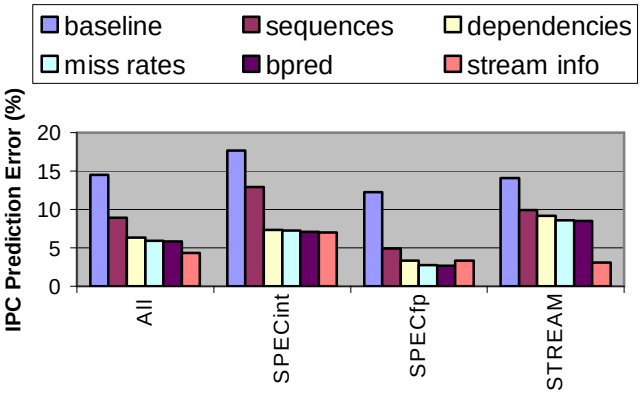

Figure 8: Improved HLS Average Error as Modeling Changes

4.2.1. Basic Block Modeling Granularity

Instead of risking reduced accuracy with block size aliasing, we model at the granularity of the basic block itself. The dynamic frequencies of all basic blocks are used as a probability distribution function for building the sequence of basic blocks in the graph. This is the same as the *k=0* modeling in the SMART-HLS system [4]. To capture cache and branch predictor statistics for the basic blocks, we use *sim-cache* augmented with the *sim-bpred* code.

In the *sequences* bars of Figures 7 and 8, the basic block instruction sequences are used, but the dependencies and locality statistics for each instruction in each basic block are still taken from the global statistics found for the entire benchmark. The overall correlation errors are reduced dramatically for the three classes of benchmarks. However, some benchmarks such as *compress* and *hydro2d*, and the STREAM loops, still show high correlation errors.

In the *dependencies* run, we include the use of dependency information for each basic block. In order to reduce the amount of information stored, we merge the dependencies into the smallest dependency relationship found in any basic block with the same instruction sequence, as in [4]. The average error is reduced significantly from 8.9% to 6.3%.

On investigation, it was found that the global miss rate calculations do not correspond to the miss rates from the viewpoint of the memory operations in a basic block. In the cache statistics, HLS pulls in the overall cache miss rate number from SimpleScalar, which includes writebacks to the L2. But for individual memory operations in a basic block, the part of the L2 miss rate due to writebacks should not be included in the miss rate. This is because the writebacks generally occur in parallel with the servicing of the miss so they do not contribute to the latency of the operation. This argues for either a global L2 miss rate calculation that does not include writebacks or the maintenance of miss rate information for each basic block. In addition, examination of the STREAM loops reveals that the miss rates for loads and stores are quite different. In *saxpy*, for example, both loads miss to the L1, but the store always hits. Because of these considerations, the L1 and L2 probabilistic miss rates for both loads and stores should be maintained local to each basic block.

The *miss rates* run includes this information. All benchmarks improve, but a few of the STREAM loops still have errors greater than 10%. The problem is that the STREAM loops need information concerning how the load and store misses, or *delayed hits*, overlap. In most cases load misses overlap, but the random cache miss variables often cause them not to overlap, leading to an underestimation of performance. Note that this is the reverse of the usual situation for statistical simulation in which critical paths are randomized to less critical paths, and performance is overestimated. An additional run, *bpred*, includes branch predictability local to each basic block. This helps a few benchmarks like *ijpeg* and *hydro2d*, but, as expected, the STREAM loops are unaffected.

One solution is to keep overlap statistics. This solves the delayed hits problem, but does not provide for the modeling of additional memory operation features. Instead, when the workload is characterized, we track one hundred L1 and L2 hit/miss indicators (i.e. if the memory operation was an L1 hit or miss or an L2 hit or miss) for the sequence of loads and stores in each basic block near the end of the one billion instruction simulation. Later, during statistical simulation, we use the stream indicators in order (but without pairing them to particular memory operations) to determine the miss characteristics of the stream as the loads and stores are encountered. This is a simplistic way to operate, since the stream hit/miss indicators are simply collected at the end of the run and are therefore not necessarily representative of the entire run. However, the technique may be applicable given the trend to identify and simulate program phases [10] in which stream information may change little. Still, simulating one billion instructions without regard to phase behavior, we expect the technique to help only the STREAM loops, and to negatively affect the others.

The *stream info* bars in Figure 7 show the results. As expected, the STREAM loops improve significantly. However, only a small amount of accuracy is lost for the others. This indicates that there is only one or a small number of phases in the first one billion instructions for most bench-

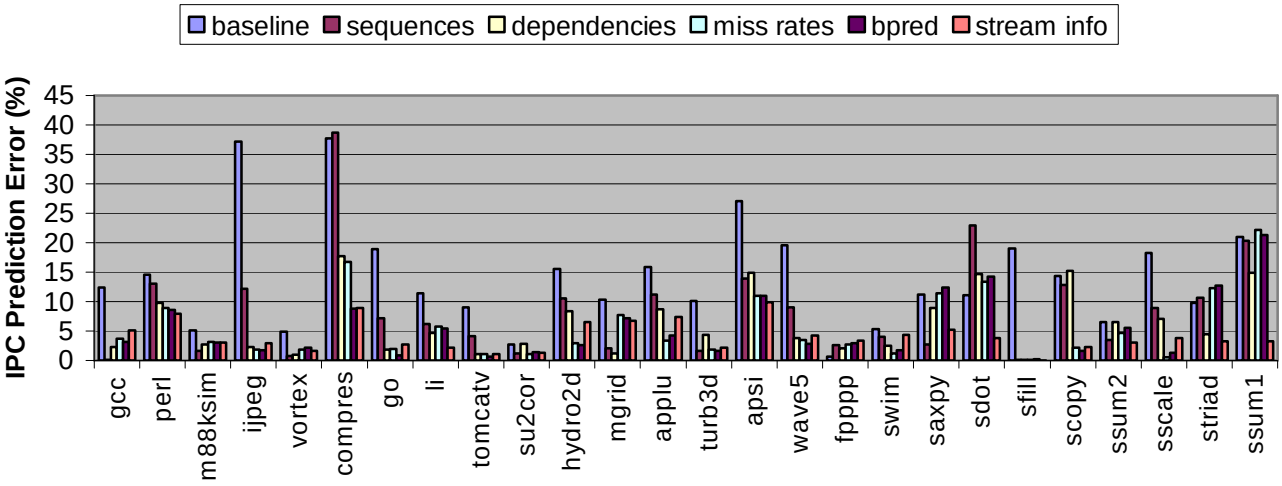

Figure 9: Improved HLS Error as Modeling Changes Using Basic Block Maps

marks, at least with respect to the load and store stream behavior.

4.2.2. Basic Block Maps

In the previous simulations, the basic blocks were not associated with each other in any way since a random variable over the frequency distribution of the blocks is used to pick the next basic block to be simulated. At branch execution time, a random variable based on the global branch predictability is used simply to indicate that a branch misprediction occurred when the branch was dispatched, causing additional delay penalty before the next instruction can be fetched, but that is not related to the successor block decision. This technique treats all blocks together as if no phases exist in which one area of the graph is favored over another at different times.

By associating particular basic blocks with each other in specific time intervals, for example during a program phase, it is expected that better simulation accuracy can be obtained for multi-phase programs. One way to do that is to specify the phases, the basic blocks executing in those phases, and the relative frequencies of the basic block executions during those phases. These three things together constitute a *basic block map*.

The phase identification requires knowledge of when the relative frequencies of the basic blocks change. The identification of phases at a coarse granularity can be carried out using a phase identification program such as SimPoint [10]. It can also be developed dynamically during simulation by walking a representation of the control flow graph of the program. A system to do that for the SPEC2000 benchmarks is presented in [4]. Since the phase identification is carried out continuously during simulation, the possibility exists of not only detecting the coarse-grained phases, but also the *micro-phases*, or small shifts in relative block frequencies, that must be identified in order to achieve good accuracy using statistical simulation.

Following [4], we annotate each basic block with a list of pointers to its successor blocks along with the probabilities of accessing each successor. By walking the basic blocks as in the previous section, but using a random variable over the successor probabilities to pick the successor, the program phase behavior is uncovered. We simulate all strategies as before.

Figures 9 and 10 show the basic block map results. The overall error using all techniques is improved only a little from 4.35% to 4.11%, a

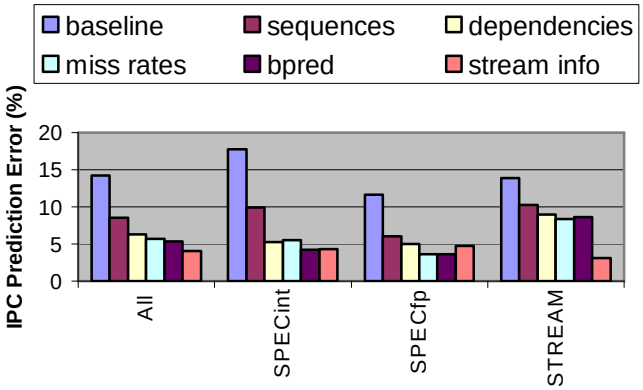

Figure 10: Improved HLS Average Error as Modeling Changes Using Basic Block Maps

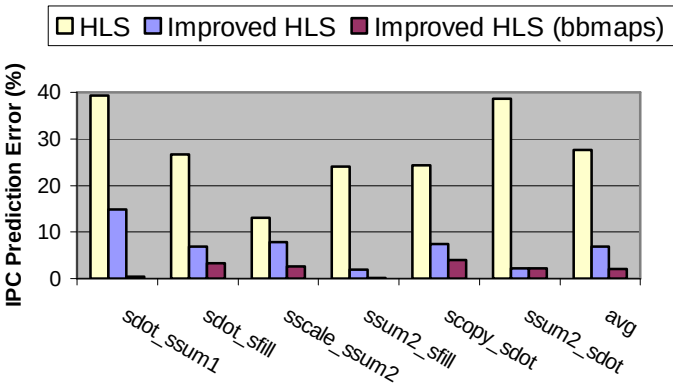

Figure 11: HLS and Improved HLS on Two-Phase Benchmarks

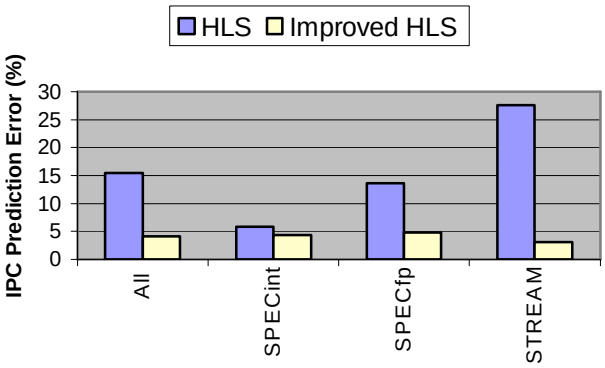

Figure 12: HLS vs. Improved HLS with Basic Block Maps

5.5% decrease. SPECint95 is improved from 6.9% to 4.3%, or 38% on average. The STREAM loops are unchanged since they consist of a single phase, and there is no advantage in using basic block maps in that case. The SPECfp95 show an increase in error from 3.3% to 4.7%. Part of this is due to the negative effects of using stream information, which cause a jump up from 3.6% error for SPECfp95. The low overall improvement agrees with the results found in the last subsection, in which stream information, which should be phase dependent, causes little adverse reaction. Coupled with increased variance by simulating only twenty thousand cycles, the result is not surprising. Improvements are also limited by errors in the graph structure, including the merge of dependencies explained earlier.

Basic block maps demonstrate improvement on programs with a number of strong phases. To demonstrate the effectiveness of the technique, several benchmarks are created using combinations of the STREAM loops. Figure 11 shows, for example, that a simple code created from the concatenation of *sdot* and *ssum1* has correlation errors of 39.4% and 14.8% in HLS and the improved HLS without basic block maps, respectively. In the improved HLS without basic block maps, given that 50% of the blocks are equivalent to *sdot* blocks, and 50% are equivalent to *ssum1* blocks, the resulting sequence of basic blocks is a jumble of both. The behavior of the resulting simulations tends to be pessimistic with long-latency L2 cache misses forming a critical chain in the dispatch window. When the basic block map technique is applied, the error shrinks to 0.4% because the sequence of simulated basic blocks is more accurate.

Figure 12 and 13 compare HLS to HLS with basic block maps running with all optimizations. The improvements show a 4.1% average error, which is 3.78 times more accurate than the original HLS at 15.5% error.

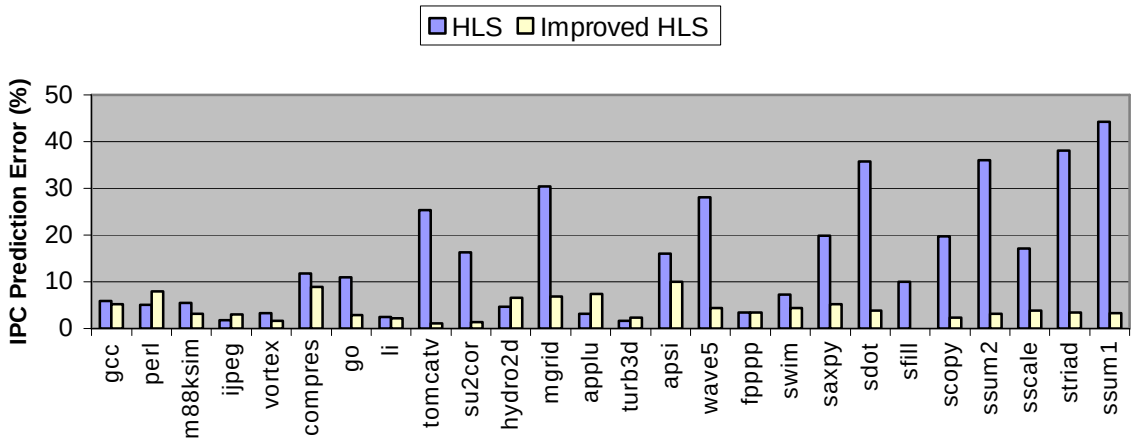

Figure 13: HLS vs. Improved HLS with Basic Block Maps

5. Implementation Costs

Table 3 shows the cost of the improvements in bytes as a function of the number of basic blocks (NBB), the average length of the basic blocks (LBB), the average number of loads and stores in the basic block (NLS), the average number of successors in the basic blocks (SBB), and the amount of stream data used (NSD). NSD is NLS x 100 = 4.71 x 100 = 471 in our runs. Table 4 shows the error reduction as the average reduction in correlation error as each technique augments the previous technique.

There are only five instruction types, so we use four bits to represent each. There are two dependencies per instruction, each of which is limited to within 255; so two bytes of storage per instruction are needed. We maintain both load and store miss rates for the L1 and L2 caches; so four floats are needed. For basic block maps, the successor pointer and frequency are maintained in in a 32-bit address and a float.

Clearly, including detailed stream data is inefficient on average compared to using the other techniques, but future work, including phase identification techniques, can seek to reduce the amount of data being collected.

6. Conclusions

Statistical simulation can provide an accurate and efficient simulation capability. In the HLS system, we identified several issues related to workload and processor modeling that affect simulation accuracy negatively.

One workload modeling issue is that the front-end graph structure of HLS operates at the

Table 3: Benchmark Information

Name	Number of basic blocks	Average Block Length	Average Ld St per Block	Average Number of Successors
gcc	2714	12.74	6.07	2.19
perl	575	9.39	4.93	1.82
m88ksim	398	10.90	4.7	1.86
ijpeg	661	13.09	6.03	1.76
vortex	1134	14.38	8.53	1.64
compress	151	8.30	3.4	1.94
go	1732	15.17	5.01	2.26
li	318	8.74	4.42	1.96
tomcatv	258	8.91	3.9	1.9
su2cor	406	9.58	3.84	1.76
hydro2d	646	11.91	3.99	1.81
mgrid	450	12.41	4.74	2.02
applu	552	25.24	8.21	1.87
turb3d	496	12.57	4.92	1.77
apsi	1010	17.94	8.45	1.65
wave5	507	9.89	3.96	1.86
fpppp	452	18.94	8.59	1.77
swim	419	12.44	4.66	1.91
saxpy	177	9.01	3.55	2.12
sdot	109	8.58	3.92	2.3
sfill	177	8.94	3.53	2.12
scopy	177	8.97	3.54	2.12
ssum2	109	8.50	3.89	2.3
sscale	177	8.98	3.54	2.12
striad	177	9.03	3.55	2.12
ssum1	177	9.02	3.55	2.12
Average	**524.4**	**10.7**	**4.71**	**1.89**

Table 4: Implementation Costs

Technique	Cost Formula (Bytes)	Avg. Cost Per Benchmark (Bytes)	Percent Error Reduction	Cost Per Percent Error Reduction (Bytes)	~Storage per Block
Cumulative Frequencies	NBB x 4	2098	42.7%	115	1 Float
Sequences	NBB x LBB x ½	2806			6 Bytes
Dependencies	NBB x LBB x 2 x 1	11222	25.4%	442	22 Bytes
Miss Rates	NBB x 4 x 4	4195	6.5%	645	4 Floats
Branch Predictability	NBB x 4	2098	2.3%	912	1 Float
Stream Info	NBB x NSD x ¼	61701	25.0%	2468	118 Bytes
Basic Block Maps	NBB x SBB x 2 x 4	7929	5.3%	1496	4 Floats
Overall		92049	73.5%	1252	186 Bytes

granularity of the instruction and contributes little to the performance of the system. The HLS processor model does not implement a specific issue width or a commit width, making calibration to a detailed processor simulator such as SimpleScalar difficult.

To meet these challenges, we model the workload at the *granularity of the basic block* and recode the processor model to decrease error. We find that IPC prediction error can be reduced from 15.5% to 4.1%. We quantify the cost of the improvements in terms of increased storage requirements and find that less than 100K bytes on average are needed per benchmark to achieve the maximum error reduction. Runtime is approximately twice that of HLS.

A simple regression analysis shows that the SPECint95 workload is susceptible to very simple processor models. Our results point to a major pitfall for simulator developers: reliance on a small set of benchmarks, datasets and simulated instructions to qualify a simulation system.

Aknowledgements

The authors would like to thank the anonymous reviewers for their feedback. Rob Bell is supported by the IBM Graduate Work Study program and the Server and Technology Division of IBM. Lieven Eeckhout is a 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] D. C. Burger and T. M. Austin, “The SimpleScalar Toolset,” Computer Architecture News, 1997.

[2] R. Carl and J. E. Smith, "Modeling Superscalar Processors Via Statistical Simulation,” Workshop on Performance Analysis and Its Impact on Design, June 1998.

[3] L. Eeckhout, S. Nussbaum, J. E. Smith and K. De Bosschere, “Statistical Simulation: Adding Efficiency to the Computer Designer’s Toolbox,” IEEE Micro, Vol. 23 No. 5, Sept/Oct 2003, pp. 26-38.

[4] L. Eeckhout, R. H. Bell Jr., B. Stougie, K. De Bosschere and L. K. John, “Improved Control Flow in Statistical Simulation for Accurate and Efficient Processor Design Studies,” Proceedings of the International Symposium on Computer Architecture, June 2004, to appear.

[5] C. P. Joshi, A. Kumar and M. Balakrishnan, "A New Performance Evaluation Approach for System Level Design Space Exploration," IEEE International Symposium on System Synthesis, October 2002, pp. 180-185.

[6] J. D. McCalpin, “Memory Bandwidth and Machine Balance in Current High Performance Computers,” IEEE Technical Committee on Computer Architecture Newsletter, December 1995.

[7] S. Nussbaum and J. E. Smith, "Modeling Superscalar Processors Via Statistical Simulation," Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, September 2001, pp. 15-24.

[8] M. Oskin, F.T.Chong and M. Farrens, "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Design," Proceedings of the 27th Annual International Symposium on Computer Architecture, June 2000, pp. 71-82.

[9]http://www.cs.washington.edu/homes/okskin/tools.html

[10] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, “Automatically Characterizing Large Scale Program Behavior,” Proceedings of the International Conference on Architected Support for Programming Languages and Operating Systems, October 2002, pp. 45-57.

[11]http://www.cs.wisc.edu/~mscalar/simplescalar.html

[12]http://www.spec.org

[13]http://www.cs.virginia.edu/stream/ref.html