Basic Block Simulation Granularity, Basic Block Maps, and Benchmark Synthesis Using Statistical Simulation

Robert H. Bell, Jr. Lizy Kurian John
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712
{belljr,ljohn}@ece.utexas.edu

1. Abstract

We show that accurate statistical simulation of technical loops requires more information than previously modelled. We show that simulation of code blocks with a granularity on the order of the basic block improves accuracy for technical loops and does not decrease accuracy for other workloads.

Higher-level basic block maps are proposed to model program phases and achieve higher accuracy. The basic block maps, implemented as graph structures, represent an actual workload and achieve good correlation with a cycle-accurate simulator.

We show how to synthesize c-code benchmarks from basic block maps. To do this, memory accesses are modelled as streams in the synthetic benchmark. Correlation results for synthetic technical loops are given. Ideas for synthesizing more accurate statistical benchmarks are discussed.

The system provides a flexible framework for future investigations of statistical simulation and benchmark synthesis.

2. Introduction

The design of modern superscalar microprocessors has been complicated by the presence of hundreds of millions of available transistors on chip [DIEF99][TEND02] and sophisticated microarchitectural techniques that use those transistors to enhance performance beyond the gains provided by technology feature size shrinks and the corresponding processor clock frequency increases [AGAR00]. Microarchitectures have evolved from scalar in-order execution processors to complex superscalar out-of-order processors involving speculative fetch and execution, branch prediction, and value prediction [JOHN90][SMIT95] [OSKI00]. In addition, the available transistor count has allowed elements of the cache hierarchy and memory subsystem to reside on-chip to reduce communication costs [TEND02]. In the face of anticipated shrink limits to CMOS technology, third-generation microarchitectures using dynamic processor reconfiguration and elements of dataflow execution are being considered [SANK03].

The growing microarchitectural complexity necessitates the use of several simulation systems written at multiple levels of machine abstraction. Near the end of the design cycle, cycle-accurate logic simulation and detailed circuit simulation are required for functional verification. At the mid-point in the design cycle, an increasingly accurate performance simulation system can be used to model and assess changes to the microarchitecture over the workload space being considered.

In the very early high-level design phases, however, a simple, fast and accurate performance simulation capability is needed [EECK03]. The simulator must be simple enough to be developed and modified quickly for the design-under-study, fast enough to evaluate large design spaces for performance and power, and yet accurate enough to model the specific workloads of concern such that the study results can be relied on for sometimes momentous early design decisions.

High-level statistical simulation systems use both machine-independent and machine-dependent information from execution-driven simulations to generate statistics that are then applied to a fast and flexible execution engine [JOHN99][EECK03]. In [NUSS01][EECK03], machine independent statistics such as instruction operation mix and average instruction dependencies from specific workloads are used to create workload traces which are input to an execution engine. Machine dependent information such as branch prediction accuracy and cache miss ratios for the specific workloads are used to dynamically create misspredictions and cache misses as the execution engine proceeds.

In [OSKI00], workload statistics are used to create a static graph of a small number of instructions, which allows much faster convergence than cycle-accurate simulations.

The execution engine typically models the stages in a superscalar out-of-order execution machine including fetch, dispatch, execution, and completion. Cache misses are modelled as additional latency to complete an instruction. No actual addresses or data are used, making simulation extremely fast. Specialized workload features such as load-hit-store address collisions or data information (for value prediction studies) must also be modelled statistically. In [JOSH02], the execution engine is modelled as a series of delays, and additional statistics facilitate the modelling of read and write buffers in a multiprocessor system.

Prior studies have shown that statistical simulation systems that correlate well with execution-driven simulators also exhibit good "relative" accuracy, meaning that the delta in machine performance given a microarchitectural change in a statistical performance simulation is reflective of the delta found using execution-driven simulation [EECK03]. Additional work is needed to quantify the relative accuracy of somewhat more inaccurate high-level statistical models, such as those typically available in the early phases of a design.

Prior studies using statistical simulation systems have focused on achieving correlation with execution-driven simulators on benchmark suites like SPECint95 [OSKI00][EECK03], SPECfp95 [EECK03][NUSS01], the IBS traces [EECK03], or specific parallel workloads [NUSS02][JOSH02]. Benchmark correlation accuracies within 15% have been achieved in those studies. The correlated statistical systems are then used to study pipeline tradeoffs such as issue width and instruction window size versus IPC or the energy-delay product [EECK03], value prediction [OSKI00], or instruction throughput in a multiprocessor system [NUSS02][JOSH02].

When evaluating high-end processor designs, usually several classes of benchmarks are used. It is often important that general-purpose, transaction-oriented, and standard benchmark suites such as SPECint perform well. For some designs it is equally important that scientific and technical workloads perform well. SPECfp and technical loops such as [STRE03] have been used to quantify the latency and bandwidth capabilities of machines at various levels of cache hierarchy and are often used as key indicators of general technical and scientific workload performance. The demands of specific technical loops have also motivated the design of many novel microarchitectural innovations in the area of data prefetching.

In this study, we focus on the challenges to statistical simulation systems posed by technical workloads. Technical workloads often rely heavily on particular kernel loops. While loops such as SAXPY are easy to understand and characterize conceptually, it is difficult to develop a unified statistical simulation system that can accurately evaluate them together with general-purpose workloads. In the next section, we outline the specific challenges and sensitivities posed by technical loops. We then describe HLS and its graph structure and give results for several benchmark suites. We compare HLS to a system augmented to handle technical loops, called S-HLS, and describe techniques to reduce the amount of information needed to achieve the desired level of simulation accuracy.

The nature of the challenges posed by a variety of workloads leads us to argue that the necessary granularity of simulation in statistical simulation is at the basic block level, not at the instruction level. To further enhance accuracy over program phases, we propose a higher-level grouping of basic blocks, called *basic block maps*. Such constructs can be built easily from the basic blocks that make up the front-end graph structure in S-HLS. It it more difficult to identify and use such mappings in trace-based systems such as [NUSS01][EECK03] because the workload is not initially represented as a graph structure.

It has always been challenging to create synthetic benchmarks that are representative of real workloads [WONG88]. The graph structure in S-HLS, and, at a higher-level, basic block maps, provide a natural way to generate real programs from a representation of the program that converges quickly. The challenges to benchmark synthesis are substantial and include the necessity of generating constructs that reproduce branch predictor and cache locality behavior. In [WONG88], the LRU hit function is described mathematically and a method, called *replication*, is described which can generate a program with a specific cache locality behavior using other programs. It is not claimed that the resulting instruction mix matches the mix of the original program, however.

An area related to benchmark synthesis is circuit synthesis. Circuit synthesis transforms low-level circuit netlists to similar but different low-level netlists. In [LEEC98], simulated annealing is used to generate new circuits from the characteristics of a circuit to prevent over-tuning in synthesis systems. In [VERP00], circuits are cloned using Rent models or mutated using wiring-signature or functional-perturbation invariant methods.

In this report. we show how to automatically synthesize representative executable benchmarks from a set of high-level workload characteristics obtained from statistical performance simulation. We give correlation results for the technical loops.

3. Technical Loops Revisited

Technical workloads have several characteristics that make it difficult for a statistical simulation system to correlate their performance with the performance of execution-driven simulators. They may often consist of one or more tight loops containing specific instruction sequences. Figure 1 shows one iteration of the SAXPY loop, disassembled

```
00400258  addu $v0[2],$v1[3],$a2[6]
00400260  l.s $f2,0($v0[2])
00400268 mul.s $f2,$f4,$f2
00400270  l.s $f0,0($v1[3])
00400278 add.s $f2,$f2,$f0
00400280  addiu $a0[4],$a0[4],1
00400288  slt $v0[2],$a1[5],$a0[4]
00400290  s.s $f2,0($v1[3])
00400298  addiu $v1[3],$v1[3],4
004002a0  beq $v0[2],$zero[0],00400258
```

Figure 1: Optimized SAXPY loop iteration

from an optimized *gcc* compile targeting the PISA machine language [BURG97]. A statistical simulation system would first generate a basic block size that would be equal to 10 instructions more or less. The system would then use an instruction type distribution to generate a random stream of instructions weighted by the frequency of the particular instruction. If the *mul.s* and *add.s* were switched in this process, with the dependency relationships remaining fixed (albeit with different corresponding instruction types), 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 overall execution time for the loop. Even if the instruction ordering was equivalent to the ordering in the figure, the dependency relationships generated from a statistical distribution could cause latencies from two misses for the load instructions not to overlap, causing generally longer iteration execution times.

Since dependency relationships are statistically generated, it is possible for longer running instructions to be paired with shorter running instructions, leading to decreased execution times. In Figure 1, the *mul.s* has a dependency on the previous *l.s*. If the statistical trace switches the load with a one-cycle *addiu*, the *mul.s* will dispatch much more quickly. For dispatch windows of 16 or more, this can lead to significant correlation errors.

In most statistical simulation systems, the program characteristics are used to randomly generate many basic blocks with instruction types and dependencies also generated randomly from a distribution [OSKI00][EECK03] [NUSS01]. This random mix is anathema to the performance simulation of technical loops, which are very sensitive to the order and exact dependency relationships of a few instructions [NUSS01]. The specific instruction sequences must be modelled with additional machine-independent information.

We seek modelling techniques that work well for both general purpose programs and technical loops. While higher-order ILP distributions might work well for technical loops, the results using that technique have been mixed in some cases and can actually lead to decreased accuracy for general purpose programs [EECK03]. In [NUSS01], the basic block size is the granule of simulation. We show below that that technique leads to a high probability of merging basic blocks with different functionality.

Table 1 shows the technical loops that we will use in this report. The third column gives the number of instructions in the kernel loop when compiled with *gcc* version 2.95.3 using *-O*. The last column gives the total amount of data needed to characterize the workload if loads and hits were written as separate streaming data items as the workload progresses, assuming 4 bytes accesses to 32 byte L1 cache lines and 64 byte L2 cache lines.

Table 1: Technical Loops

benchmark	equation	loop instructions	Unique Stream Data
saxpy	z[k] = z[k] + q * x[k]	10	16 x 3
sdot	q = q + z[k] * x[k]	9	16 x 2
sfill	z[k] = q	5	1
scopy	z[k] = x[k]	7	16 x 2
ssum2	q = q + x[k]	6	16 x 1
sscale	z[k] = q * x[k]	8	16 x 2
striad	z[k] = y[k] + q * x[k]	11	16 x 3
ssum1	z[k] = y[k] + x[k]	10	16 x 3

4. Statistical Simulation using HLS

One statistical simulation system is HLS, available at [OSKI03], also described in [OSKI00]. In HLS, the machine independent characteristics are analyzed using a modified version of Sim-Fast [BURG97]. This gives an instruction mix distribution consisting of the percentages of integer, float, load, store and branch instructions. No other differentiation among the types of each of those groups is analyzed. Also calculated are the basic block mean size and standard deviation.

In addition, the frequency distribution of the dependency distances between instructions for each input of the five instruction types is given. For loads, only one input has a frequency distribution. Each distribution gives the percentages of all instructions of that type for that input that have a dependency distance of zero, one, two, three, etc., up to 19. Additional percentages are given for dependency distances between 20 and 100, between 101 and 1000, and above 1000, for a total of 23 dependency distance frequencies per input per instruction type. So the total instruction mix and dependency information, D_{HLS}, (in bytes) is

$$D_{HLS} = (5 + 2 + (4 \times 2 + 1) \times 23) \times 4 = 856$$

After the workload is characterized, HLS generates 100 basic blocks using a normal random variable of the mean block size and standard deviation. Then, a uniform random variable of the instruction percentages is used to fill in the instructions of each basic block. Each basic block terminates in a branch, so the percentage of branches is excluded from the random variable. The overall instruction mix turns out to be about right since the basic block size variable is also based on the workload.

For each randomly generated instruction, a uniform random variable over the dependency distance frequencies is used to specify a dependency for each instruction input. An effort is made to make an instruction not dependent on a store within the current basic block, so that a dependency distance may be calculated multiple times. But if the dependency that is generated stretches beyond the limits of the basic block, no change is made.

The basic blocks are connected into a graph via the branches. Each branch has both a taken pointer and a not taken pointer to other blocks. The not-taken pointer points to the next sequentially-generated basic block. A uniform random variable on the average number 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 branches, a normal random variable on the mean backward block jump distance and standard deviation, set statically to 10 and 3 in the code, determine which basic block the taken pointer points to. For forward branches, a normal random variable on the mean forward block jump distance and standard deviation, set statically to 3 and 2 in the code, determine which basic block the taken pointer points to. Normal random variables based on the forward and backward branch predictability means and standard deviations, also statically set to 85% and 0% each, are used to get a value to compare against for each branch. Later, during simulation, the machine-dependent branch predictability determines dynamically if the branch is actually taken or not, and the corresponding branch pointer is followed.

Unfortunately, as described in the next section, there is no real advantage to using the above branching scheme in a simulation system in which the basic block sizes, instructions, and their dependencies, are all randomly generated from statistical distributions.

After the machine independent statistics are processed, the instruction graph is executed on a generalized superscalar execution model. Sim-OutOrder statistics such as L1 and L2 I-cache and D-cache hit rates and overall branch predictability model the machine dependent locality structures. For comparison with SimpleScalar, the load and store queues are modelled as a single queue. No load-hit-store forwarding is modelled. Delayed-hits are also not modelled, so in the default configuration the only parallel cache miss operation occurs over the two memory ports available to the load-store execution unit. As in SimpleScalar, stores execute immediately when they reach the tail of the queue and an execution unit is available.

Because of the generalized execution model, there is no issue-width concept in the HLS system. The issue of instructions to the issue queues associated with the execution units is instead limited by the queue size and dispatch window and, ultimately, by the fetch window. There is also no limit to the number of completions per cycle in HLS, so the completion rate is also front-end limited. These are conducive to obtaining quick convergence to an average result for well-behaved benchmarks, but they make it difficult to exactly correlate the system to SimpleScalar for technical loops, as we shall see.

5. Setup and Procedure

Following the procedure in [OSKI00], running on an IBM Power3 P260, SimpleScalar release 2.0 [BURG97] and the modified Sim-Fast found at [OSKI03] were downloaded and compiled to target big-endian pisa binaries. Sim-OutOrder using default parameters was then run on the big-endian SPECint95 binaries found at [SOHI03] using up to one billion instructions of one reference input dataset. As implemented in the code from [OSK03], the modified Sim-Fast was run on the input dataset to completion.

The HLS system was then downloaded from [OSKI03] and compiled using *g++* (*gcc* version 2.95.3) without change.

6. The HLS Graph Structure

We wish to show that, since instructions are generated randomly and placed into the basic blocks, no distinct advantage is obtained in HLS when using a graph structure as given in [OSKI00]. We vary the percentages of backward branches, the backward branch jump distance, the forward branch jump distance, and the graph connections themselves to demonstrate that the graph structure as used in HLS has no impact on IPC.

First, the HLS run results for a reference input data set is given. The data set may not be the same as in [OSKI00] if multiple reference inputs exist because the particular datasets used in [OSKI00] are not specified.

Table 2: Baseline SPECint95 Run for Graph Structure Studies

Benchmark	SimpleScalar IPC	HLS IPC	HLS Sigma	Error (%)
gcc	0.9096	0.962255	0.029882	5.787
perl	1.1610	1.23764	0.052625	6.600
m88ksim	1.4586	1.401735	0.041695	3.897
ijpeg	1.8449	1.884525	0.067900	2.146
vortex	0.9422	0.925260	0.025933	1.796
compress	1.1290	1.249695	0.079047	10.689
go	0.9644	1.04897	0.031119	8.768
li	1.5442	1.518960	0.060196	1.633

Now we take *gcc* and vary the percentage of backward branches used to form the graph. Note that there does not seem to be a trend in the error.

Table 3: Fraction of Branches that are Backward versus Error for gcc

Fraction Backward Branches	SimpleScalar IPC	HLS IPC	HLS Sigma	Error (%)
0.05	0.9096	0.950270	0.032830	4.470
0.10	0.9096	0.964385	0.027900	6.021
0.15	0.9096	0.963525	0.027856	5.927
0.20	0.9096	0.969035	0.035735	6.533
0.30	0.9096	0.956085	0.027676	5.109
0.40	0.9096	0.964090	0.040602	5.989
0.50	0.9096	0.966575	0.060843	6.262
0.60	0.9096	0.936545	0.049673	2.961
0.70	0.9096	0.948450	0.058764	4.270
0.80	0.9096	0.948605	0.050113	4.287
0.90	0.9096	0.984435	0.062481	8.226

We now vary the mean backward block jump distance from 1 to 16. Again, no trend.

Table 4: Backward Branch Mean Jump Distance versus Error for gcc

Mean Backward Block Jump Distance	SimpleScalar IPC	HLS IPC	HLS Sigma	Error (%)
1	0.9096	0.960260	0.026757	5.568
2	0.9096	0.951070	0.034218	4.558
3	0.9096	0.943925	0.024706	3.772
4	0.9096	0.968600	0.026717	6.485
5	0.9096	0.958100	0.025197	5.331
6	0.9096	0.947485	0.034427	4.164
7	0.9096	0.963295	0.029373	5.902
8	0.9096	0.970905	0.019472	6.738
9	0.9096	0.964240	0.041481	6.006
10	0.9096	0.953565	0.026938	4.832
11	0.9096	0.965455	0.036890	6.139
12	0.9096	0.964835	0.029053	6.071
13	0.9096	0.949805	0.030311	4.419
14	0.9096	0.960670	0.028884	5.613
15	0.9096	0.962885	0.029452	5.857
16	0.9096	0.957170	0.031255	5.228

We now vary the mean forward block jump distance. Again, there appears to be no trend.

Table 5: Forward Branch Mean Jump Distance versus Error for gcc

Mean Forward Block Jump Distance	SimpleScalar IPC	HLS IPC	HLS Sigma	Error (%)
1	0.9096	0.969510	0.029858	6.585
2	0.9096	0.948625	0.042222	4.289
3	0.9096	0.950420	0.022023	4.486
4	0.9096	0.958320	0.026656	5.355
5	0.9096	0.966225	0.030679	6.224
6	0.9096	0.950215	0.032600	4.464
7	0.9096	0.950880	0.026010	4.537
8	0.9096	0.960470	0.025520	5.591
9	0.9096	0.968590	0.030637	6.484
10	0.9096	0.966165	0.033810	6.217
11	0.9096	0.960370	0.032818	5.580

We now randomize the not-taken (fall-through) target for the basic blocks in each benchmark.

Table 6: HLS Error When Randomizing the Not-Taken Branch

Benchmark	SimpleScalar IPC	HLS IPC	HLS Sigma	Error (%)
gcc	0.9096	0.965660	0.032898	6.162
perl	1.1610	1.193370	0.111949	2.787
m88ksim	1.4586	1.421580	0.064695	2.537
ijpeg	1.8449	1.929850	0.176567	4.603
vortex	0.9422	0.928405	0.027221	1.463
compress	1.1290	1.248515	0.174408	10.584
go	0.9644	1.080000	0.058706	11.985

Table 6: HLS Error When Randomizing the Not-Taken Branch

Benchmark	SimpleScalar IPC	HLS IPC	HLS Sigma	Error (%)
li	1.5442	1.484475	0.116028	3.866

Within the error obtained using separate random seeds, which can be over 3%, there is no significant effect. Now we set the taken target to the next sequentially created block.

Table 7: HLS Error Setting the Taken Branch to Next-Sequential

Benchmark	SimpleScalar IPC	HLS IPC	HLS Sigma	Error (%)
gcc	0.9096	0.965210	0.019936	6.112
perl	1.1610	1.219300	0.030253	5.020
m88ksim	1.4586	1.426905	0.037285	2.171
ijpeg	1.8449	1.920560	0.039689	4.100
vortex	0.9422	0.926190	0.030187	1.698
compress	1.1290	1.217400	0.054286	7.828
go	0.9644	1.063855	0.030351	10.311
li	1.5442	1.514995	0.067419	1.890

Again no significant difference. Now do the same but the last block simply feeds the first created block. The total effect is to create a big loop of basic blocks. There is no substantive difference in error.

Table 8: HLS Error Using Big Loop of Basic Blocks

Benchmark	SimpleScalar IPC	HLS IPC	HLS Sigma	Error (%)
gcc	0.9096	0.972175	0.026239	6.878
perl	1.1610	1.220555	0.033056	5.128
m88ksim	1.4586	1.414640	0.038046	3.012
ijpeg	1.8449	1.888350	0.041286	2.354
vortex	0.9422	0.921780	0.025405	2.166
compress	1.1290	1.230770	0.049961	9.013
go	0.9644	1.068515	0.025836	10.794
li	1.5442	1.511875	0.041735	2.092

To summarize, when using individual instructions as the granule of simulation, i.e. randomly picking instructions and placing them in basic blocks, the HLS graph structure itself makes no difference to accuracy as measured by comparing IPC between SimpleScalar and HLS runs. This begs the question of what the simulation granule should be for statistical simulation. As we shall argue, very good accuracy can be obtained by using a simulation granularity at the basic block level. This is similar to the block size simulation granularity presented in [NUSS01], but, to avoid unnecessary aliasing, the starting point is the set of the most frequent dynamic basic blocks.

Note that the HLS graph is a representation of program built from a real program. The graph contains a set of basic blocks with the instructions that represent the original workload. This idea will assume more significance when we discuss benchmark synthesis below.

7. HLS Runs

In each of the following tables, the benchmarks are summarized at the bottom of the table.

Here we try to quantify the error of HLS running on SPECint95, SPECfp95, and the technical loops. We use SPEC95 to compare with the results in [OSKI00] using the code downloaded from [OSKI03]. The modified Sim-Fast program runs each benchmark for up to 50 B instructions to collect workload statistics. An input reference data set is used. This may not be the same one used in [OSKI00] if multiple reference sets are available since the dataset that was used was not specified.

Table 9: Baseline HLS Run

Benchmark	SimpleScalar IPC	HLS IPC	HLS Sigma	Error (%)
gcc	0.9096	0.962840	0.032180	5.852
perl	1.1610	1.219825	0.035786	5.065
m88ksim	1.4586	1.379890	0.037567	5.395
ijpeg	1.8449	1.876680	0.065075	1.721
vortex	0.9422	0.911755	0.026343	3.230
compress	1.1290	1.261490	0.063339	11.734
go	0.9644	1.070320	0.031205	10.981
li	1.5442	1.506735	0.068131	2.425
tomcatv	0.9314	1.166815	0.019603	25.274
su2cor	1.0188	1.184175	0.023817	16.231
hydro2d	1.0888	1.139675	0.096239	4.671
mgrid	1.6826	1.171475	0.039468	30.376
applu	1.3779	1.334175	0.045145	3.172
turb3d	1.6288	1.655495	0.070213	1.637
apsi	1.1737	1.361210	0.053095	15.974
wave5	1.1619	1.488095	0.051179	28.073
fpppp	0.7117	0.687255	0.016716	3.433
swim	1.1813	1.096045	0.034478	7.216
saxpy	1.2878	1.031680	0.082820	19.887
sdot	1.2412	0.798475	0.061000	35.668
sfill	2.4769	2.228660	0.101467	10.021
scopy	1.5940	1.279990	0.099503	19.698
ssum2	1.4090	0.901300	0.104733	36.031
sscale	1.3875	1.149555	0.109330	17.148
striad	1.4165	0.877010	0.069869	38.085
ssum1	1.6621	0.927170	0.060633	44.215
All (mean/stdev)	15.508192		12.760963	
SPECint (mean/stdev)	5.800375		3.489195	
SPECfp (mean/stdev)	13.605700		10.561481	
TECH (mean/stdev)	27.594125		11.523699	

Note that, while SPECint does well as in [OSKI00], SPECfp has twice the correlation error. The technical loop error is more than four times worse. Also note the large standard deviation for SPECfp and the technical loops. While more work could be done to try to calibrate the generalized HLS processor for all the benchmarks using test data sets, attempts to do so failed. As described later, additional workload information was needed to obtain a framework that worked for all three kinds of benchmarks.

A large improvement in correlation can be had by changing the modified Sim-Fast from HLS to collect statistics for only one billion instructions, the same number of instructions used to get cache and branch predictor statistics in Sim-OutOrder. Both SPECint and SPECfp are significantly improved, but the standard deviation for SPECfp remains large because of *mgrid* and *apsi*. The technical loops are not helped.

Table 10: HLS using Only 1 Billion Sim-Fast Instructions

Benchmark	SimpleScalar IPC	HLS IPC	HLS Sigma	Error (%)
gcc	0.9096	0.969790	0.036009	6.616
perl	1.1610	1.200830	0.038747	3.429
m88ksim	1.4586	1.426720	0.035693	2.184

Table 10: HLS using Only 1 Billion Sim-Fast Instructions

Benchmark	SimpleScalar IPC	HLS IPC	HLS Sigma	Error (%)
ijpeg	1.8449	1.874115	0.067641	1.582
vortex	0.9422	0.926870	0.023626	1.626
compres	1.1290	1.211340	0.082729	7.292
go	0.9644	1.073670	0.042132	11.329
li	1.5442	1.475600	0.069037	4.441
tomcatv	0.9314	0.952090	0.023910	2.220
su2cor	1.0188	0.990465	0.027333	2.780
hydro2d	1.0888	1.168840	0.076533	7.350
mgrid	1.6826	1.189765	0.040772	29.289
applu	1.3779	1.355025	0.034835	1.659
turb3d	1.6288	1.644165	0.064775	0.942
apsi	1.1737	1.385880	0.058306	18.076
wave5	1.1619	1.158195	0.070632	0.317
fpppp	0.7117	0.700820	0.018319	1.527
swim	1.1813	1.137020	0.062677	3.747
saxpy	1.2878	0.992110	0.124944	22.959
sdot	1.2412	0.759980	0.052006	38.769
sfill	2.4769	2.287070	0.099732	7.663
scopy	1.5940	1.281400	0.095744	19.610
ssum2	1.4090	0.879885	0.096139	37.551
sscale	1.3875	1.109400	0.074767	20.042
striad	1.4165	0.892850	0.078510	36.966
ssum1	1.6621	0.912170	0.078307	45.118
All (mean/stdev)	12.887846		13.737734	
SPECint (mean/stdev)	4.812375		3.190147	
SPECfp (mean/stdev)	6.790700		9.001347	
TECH (mean/stdev)	28.584750		11.992705	

Note that the technical loops are uniformly slower in HLS than in SimpleScalar. While studying the technical loop error, it was discovered that the modified Sim-Fast in HLS makes no distinction between memory instructions in the PISA language that carry out auto-increment or auto-decrement on the address register after memory access and those that do not. When not in auto-increment or auto-decrement mode, memory instructions do not change the contents of the general-purpose register used as the address for an access. The HLS Sim-Fast code always assumes memory operations are in this mode. This causes the code to assume register dependences between instructions that do not exist and make codes with significant numbers of load and store dependences appear to run slower. The Sim-Fast code was modified to check the instruction operand for the condition and mark dependences accordingly. The following run shows the results.

Table 11: HLS with Auto-Inc/Dec Instruction Bug Fix

Benchmark	SimpleScalar IPC	HLS IPC	HLS Sigma	Error (%)
gcc	0.9096	0.980690	0.029812	7.814
perl	1.1610	1.308385	0.045983	12.693
m88ksim	1.4586	1.472850	0.052602	0.975
ijpeg	1.8449	2.128685	0.078539	15.381
vortex	0.9422	0.939565	0.018257	0.277
compres	1.1290	1.288890	0.080300	14.161
go	0.9644	1.088810	0.034582	12.900

Table 11: HLS with Auto-Inc/Dec Instruction Bug Fix

Benchmark	SimpleScalar IPC	HLS IPC	HLS Sigma	Error (%)
li	1.5442	1.703840	0.064999	10.337
tomcatv	0.9314	0.963815	0.022415	3.479
su2cor	1.0188	1.014700	0.033177	0.401
hydro2d	1.0888	1.213430	0.062992	11.445
mgrid	1.6826	1.576605	0.085137	6.298
applu	1.3779	1.482015	0.045743	7.555
turb3d	1.6288	1.709135	0.063649	4.931
apsi	1.1737	1.608430	0.058035	37.038
wave5	1.1619	1.290210	0.066091	11.042
fpppp	0.7117	0.701545	0.020937	1.425
swim	1.1813	1.216150	0.053905	2.949
saxpy	1.2878	1.104650	0.112750	14.220
sdot	1.2412	0.843590	0.067161	32.033
sfill	2.4769	2.243655	0.086004	9.415
scopy	1.5940	1.406540	0.113086	11.759
ssum2	1.4090	0.913105	0.091436	35.193
sscale	1.3875	1.276735	0.122670	7.982
striad	1.4165	0.982945	0.081332	30.606
ssum1	1.6621	1.022080	0.133868	38.505
All (mean/stdev)	13.108231		11.455160	
SPECint (mean/stdev)	9.317250		5.466737	
SPECfp (mean/stdev)	8.656300		10.096836	
TECH (mean/stdev)	22.464125		11.936102	

Again, work was done to try to recalibrate HLS, but intrinsic workload differences made it impossible to reconcile the errors for all three groups. In SPECint, *perl*, *ijpeg* and *compress* increased in error. In SPECfp, the error for *apsi* almost doubled.

We also ran HLS using a dispatch window size of 32.

Table 12: HLS using a Dispatch Window of 32

Benchmark	SimpleScalar IPC	HLS IPC	HLS Sigma	Error (%)
gcc	0.9178	0.981825	0.028521	6.974
perl	1.1627	1.241670	0.052421	6.790
m88ksim	1.4624	1.443345	0.059080	1.301
ijpeg	2.0443	1.891380	0.048275	7.479
vortex	0.9501	0.935230	0.026634	1.564
compres	1.2126	1.179490	0.091150	2.729
go	0.9928	1.078000	0.054173	8.580
li	1.5804	1.489775	0.078504	5.733
tomcatv	0.9502	0.971220	0.026506	2.211
su2cor	1.0358	1.012415	0.037292	2.256
hydro2d	1.2736	1.253245	0.073781	1.597
mgrid	1.8161	1.177150	0.040049	35.181
applu	1.6507	1.353435	0.058746	18.007
turb3d	1.9772	1.661580	0.053740	15.961
apsi	1.3412	1.405510	0.056961	4.793
wave5	1.2482	1.188670	0.099693	4.768

Table 12: HLS using a Dispatch Window of 32

Benchmark	SimpleScalar IPC	HLS IPC	HLS Sigma	Error (%)
fpppp	0.7143	0.706010	0.016745	1.159
swim	1.3620	1.123050	0.058747	17.543
saxpy	1.6621	1.014810	0.092639	38.943
sdot	1.7558	0.819395	0.059563	53.331
sfill	2.4771	2.245675	0.099129	9.341
scopy	1.6901	1.267900	0.098147	24.979
ssum2	1.6510	0.897710	0.108843	45.625
sscale	1.7469	1.152055	0.097205	34.050
striad	1.8094	0.922470	0.071190	49.016
ssum1	1.8126	0.923840	0.091264	49.031
All (mean/stdev)	17.267000		17.343272	
SPECint (mean/stdev)	5.143750		2.671159	
SPECfp (mean/stdev)	10.347600		10.551376	
TECH (mean/stdev)	38.039500		13.890552	

SPECint does better, but the technical loops do much worse.

8. Improving Correlation Errors using S-HLS

While trying to correlate the technical loops, it was noticed that, because of its generalized execution model, HLS has no concept of an issue width. The issue of instructions to the issue queues associated with the execution units is simply limited by the issue queue sizes and dispatch window and, ultimately, by the fetch width. There is also no limit to the number of completions per cycle in HLS, so the completion rate is also front-end limited. These are conducive to obtaining quick convergence to an average result for well-behaved benchmarks, but they make it difficult to exactly correlate the system to SimpleScalar for technical loops. HLS also has no separate issue queues for longer latency integer or floating point operations.

In order to help with the task of correlation, the SimpleScalar-HLS (S-HLS) system was developed. The goal of S-HLS is to more accurately model the execution engine of SimpleScalar. We want the execution engines to match for two reasons. First, if the execution engine is modelled accurately, then the correlation errors due to the machine-independent workload characteristics can be more thoroughly studied. Second, if we model the execution engine accurately and model the workload characteristics sufficiently, we expect to be able to achieve very low correlation errors on a variety of benchmarks. Both of these goals are in contrast to HLS, in which it is assumed that, because of the generalized processor, machine parameters will be calibrated or tuned to achieve low correlation errors for the benchmarks under study. Even so, we wish to run more efficiently than SimpleScalar, so outside the pipeline we do not model everything exactly. The caches and branch predictor in particular still use the statistical parameters taken from the SimpleScalar runs. S-HLS is therefore an intermediate combination of the execution engine in [NUSS01] and the statistical locality structures and graphical front-end of [OSK00]. Also, based on the analysis of the graph structure above and in order to obtain the fastest convergence, we model the program as a single loop of basic blocks.

The overall framework of HLS was retained including the front-end execution graph structure, but the execution engine was rewritten to include an RUU (Register-Update Unit) as in SimpleScalar, with a dispatch window equal to the RUU size, an issue width, and a completion width. These are all parameters to S-HLS and are set to the corresponding SimpleScalar machine values.

We first run the benchmarks on S-HLS using only the workload characteristics modelled in HLS, as described above. The difference here is that the execution engine flow, delays and parameters are all chosen to match those in SimpleScalar, while in HLS, as mentioned, the generalized processor needs to be tuned for the benchmarks under study. Given the more accurate machine model, we wish to determine what additional workload characteristics need to be modelled to achieve small correlation errors. Table 13 shows the baseline run.

There are large errors for particular benchmarks, such as *ijpeg*, *compress* and *apsi*. But overall, there is a remarkable similarity of the mean error for each of the three kinds of benchmarks and the overall error. This is an indication

Table 13: S-HLS Baseline Run

Benchmark	SimpleScalar IPC	S-HLS IPC	S-HLS Sigma	Error (%)
gcc	0.9096	0.967090	0.016043	6.319
perl	1.1610	1.268023	0.028371	9.217
m88ksim	1.4586	1.479110	0.025797	1.405
ijpeg	1.8449	2.669050	0.052241	44.670
vortex	0.9422	0.911880	0.011047	3.217
compres	1.1290	1.589893	0.036913	40.822
go	0.9644	1.094280	0.018289	13.466
li	1.5442	1.823467	0.033757	18.083
tomcatv	0.9314	0.938056	0.014192	0.713
su2cor	1.0188	0.949989	0.010429	6.753
hydro2d	1.0888	1.306746	0.022473	20.016
mgrid	1.6826	1.711185	0.014135	1.697
applu	1.3779	1.697247	0.015573	23.175
turb3d	1.6288	2.088964	0.031786	28.250
apsi	1.1737	1.679349	0.021318	43.080
wave5	1.1619	1.400348	0.025185	20.521
fpppp	0.7117	0.709588	0.006782	0.294
swim	1.1813	1.348526	0.016732	14.155
saxpy	1.2878	1.518453	0.059191	17.909
sdot	1.2412	1.105377	0.032714	10.941
sfill	2.4769	2.993007	0.071628	20.835
scopy	1.5940	1.999458	0.092190	25.435
ssum2	1.4090	1.328866	0.054605	5.686
sscale	1.3875	1.745563	0.036771	25.805
striad	1.4165	1.318285	0.054329	6.932
ssum1	1.6621	1.392186	0.045314	16.238
All (mean/stdev)	16.370538		12.608257	
SPECint (mean/stdev)	17.149875		15.630231	
SPECfp (mean/stdev)	15.865400		13.210275	
TECH (mean/stdev)	16.222625		7.297599	

that the underlying operation of the machine is not skewed to favor a particular benchmark class.

We enhance the workload model to reduce correlation errors. From the analysis of the graph structure, we noted that the instruction level granularity in the workload analysis of HLS did not seem to contribute to accuracy. In [NUSS01], the basic block size is used as the granularity of simulation for a variety of studies. Instruction distributions are maintained for each basic block size found in the workload. However, multiple basic blocks in a workload may happen to have the same size but different functionality and dependence relationships. This *block size aliasing* can reduce the effectiveness of the techniques described in [NUSS01]. As an example, in *gcc*, there are 66 different sequences that are of length 9. One sequence contains 7 sequential loads and an integer operation. Another equally likely sequence contains 6 sequential integer operations followed by a load and an integer operation. The dependence and stream information for these two blocks are very different.

Instead, S-HLS uses the exact sequence of instructions in the basic block as the granularity of simulation. The frequencies of each basic block so defined is maintained and used as a distribution function for building the sequence of basic blocks in the S-HLS graph. By using the basic block as the granule of simulation, additional information can easily be extracted and maintained. Table 14 below lists the number of basic blocks found for each benchmark that account for the top 99% of all basic blocks in the first one billion instructions, ordered by the blocks with the highest frequencies. It also gives the average length of the basic blocks (not weighted by frequency of the basic block, but just over the count of basic blocks, for the purposes of determining how much information must be stored). It also gives the number of basic block sizes that alias to other blocks.

Table 14: Basic Blocks and Size Aliasing in 99% of Dynamic Basic Blocks

benchmark	number of basic block	average static basic block length	number of blks in 50% or more of the dynamic blks	block size aliases in the top 50% of dynamic blks	total number of static block size aliases	average static loads and stores per basic blk
gcc	696	10	7	3	662	5.00
perl	71	5	11	4	56	2.77
m88ksim	98	8	5	1	78	3.36
ijpeg	51	24	5	1	27	5.78
vortex	269	12	5	1	234	6.81
compress	10	7	3	0	1	0.90
go	550	13	12	4	507	4.49
li	72	6	5	1	58	3.49
tomcatv	143	8	8	3	118	3.45
su2cor	99	7	7	2	81	2.98
hydro2d	209	9	16	7	178	3.42
mgrid	32	39	2	0	5	12.00
applu	89	51	8	0	33	14.17
turb3d	77	15	6	0	45	4.62
apsi	314	19	20	3	249	7.36
wave5	68	9	8	3	44	2.59
fpppp	159	27	22	8	112	12.80
swim	40	14	6	0	20	4.53
saxpy	10	4	1	0	3	1.60
sdot	10	4	1	0	3	1.30
sfill	10	4	1	0	3	1.40
scopy	10	4	1	0	3	1.50
ssum2	10	4	1	0	4	1.20
sscale	10	4	1	0	3	1.50
striad	10	4	1	0	3	1.60
ssum1	10	4	1	0	3	1.60
average	120.2	12.4	6.3	1.6	97.4	4.32

The number of basic blocks is relatively small. For SPEC benchmarks with small average block sizes, the proportion of size aliases among the blocks that account for 50% or more of dynamic blocks is quite large. The average loads and stores column is not the dynamic average but the static average found using all basic blocks. It is used later to help calculate information storage requirements. As future work, the system could be augmented to use basic block size as the granule of simulation as in [NUSS01] in order to quantify the impact of block size aliasing.

In the following run, the basic blocks are chosen from the cumulative frequency distribution maintained for each basic block. Dependences for each instruction in each basic block are still taken from the global statistics found for the entire benchmark. For memory operations in a basic block, the L1 and L2 hit rates are still taken from the global statistics for each workload.

Table 15: S-HLS using Basic Block Instruction Sequences

Benchmark	SimpleScalar IPC	S-HLS IPC	S-HLS Sigma	Error (%)
gcc	0.9096	0.930754	0.024677	2.324
perl	1.1610	1.238857	0.038527	6.705
m88ksim	1.4586	1.395092	0.039903	4.353
ijpeg	1.8449	2.262868	0.093465	22.654
vortex	0.9422	0.879734	0.012541	6.628
compres	1.1290	1.604509	0.042010	42.116

Table 15: S-HLS using Basic Block Instruction Sequences

Benchmark	SimpleScalar IPC	S-HLS IPC	S-HLS Sigma	Error (%)
go	0.9644	1.050701	0.022531	8.947
li	1.5442	1.798550	0.054339	16.470
tomcatv	0.9314	0.885911	0.017648	4.882
su2cor	1.0188	0.943626	0.019350	7.377
hydro2d	1.0888	1.267340	0.026856	16.396
mgrid	1.6826	1.781148	0.020237	5.855
applu	1.3779	1.562656	0.076503	13.407
turb3d	1.6288	1.898085	0.045452	16.531
apsi	1.1737	1.531739	0.030696	30.504
wave5	1.1619	1.278123	0.039632	10.001
fpppp	0.7117	0.671739	0.014222	5.613
swim	1.1813	1.264830	0.037038	7.070
saxpy	1.2878	1.354089	0.021920	5.146
sdot	1.2412	0.979834	0.013060	21.056
sfill	2.4769	2.480448	0.004016	0.141
scopy	1.5940	1.948808	0.029080	22.257
ssum2	1.4090	1.400508	0.023833	0.601
sscale	1.3875	1.596286	0.010458	15.046
striad	1.4165	1.313990	0.019275	7.235
ssum1	1.6621	1.446342	0.017716	12.980
All (mean/stdev)	12.011346		9.526510	
SPECint (mean/stdev)	13.774625		12.417656	
SPECfp (mean/stdev)	11.763600		7.500449	
TECH (mean/stdev)	10.557750		8.070553	

There are several things to note. First, the correlation errors were reduced uniformly for the three benchmark classes. However, some benchmarks such as compress and *apsi*, still show high correlation errors.

Second, a lot of information is maintained for each benchmark. Using Table 14, the average basic block length times the number of basic blocks gives the amount of information that needs to be maintained. There is also a runtime cost to S-HLS to read in the information and use it.

Third, the information varies in size depending on the workload. For the technical loops and compress, very little information is maintained. For the technical loops only one basic block accounts for 99% of the basic blocks used. For other benchmarks like *apsi* and *gcc*, more information is kept. Because of the variety of workloads, it makes sense to maintain more information for those that require it and less for those that do not. While HLS maintains the same small amount of information for every benchmark, there is a cost in terms of accuracy and the need to tune the processor for each class of benchmarks. In a later section, we study the tradeoff between reducing the amount of basic block information and reducing the correlation error.

In the next run, we include the use of dependence information for each basic block. Over the lifetime of the workload, a single basic block may have a variety of dependency relationships with other instructions. The granularity of simulation includes both the basic block instruction sequence and its dependence sequence. In practice, however, the dependencies that vary are those that extend to instructions external to the basic block. In this study, in order to reduce the amount of information maintained, we merge the dependences into the smallest dependency relationship found in any basic block with the same instruction sequence.

Table 16: S-HLS using Basic Block Dependency Information

Benchmark	SimpleScalar IPC	S-HLS IPC	S-HLS Sigma	Error (%)
gcc	0.9096	0.879541	0.020545	3.303
perl	1.1610	1.248184	0.025940	7.508

Table 16: S-HLS using Basic Block Dependency Information

Benchmark	SimpleScalar IPC	S-HLS IPC	S-HLS Sigma	Error (%)
m88ksim	1.4586	1.393629	0.051280	4.453
ijpeg	1.8449	2.117462	0.075550	14.772
vortex	0.9422	0.878597	0.016402	6.749
compres	1.1290	1.382506	0.023606	22.453
go	0.9644	1.037899	0.022703	7.620
li	1.5442	1.646120	0.044311	6.600
tomcatv	0.9314	0.855831	0.022379	8.112
su2cor	1.0188	0.933190	0.027497	8.402
hydro2d	1.0888	1.225168	0.033493	12.523
mgrid	1.6826	1.769635	0.009705	5.171
applu	1.3779	1.520872	0.024841	10.375
turb3d	1.6288	1.885516	0.049102	15.760
apsi	1.1737	1.528396	0.034668	30.219
wave5	1.1619	1.118354	0.057583	3.746
fpppp	0.7117	0.673322	0.013711	5.391
swim	1.1813	1.258897	0.054098	6.567
saxpy	1.2878	1.461255	0.010959	13.468
sdot	1.2412	1.085607	0.012296	12.534
sfill	2.4769	2.480262	0.004075	0.134
scopy	1.5940	1.943272	0.015235	21.910
ssum2	1.4090	1.514421	0.015164	7.480
sscale	1.3875	1.612845	0.008474	16.240
striad	1.4165	1.388360	0.012554	1.985
ssum1	1.6621	1.533435	0.009739	7.740
All (mean/stdev)	10.046731		6.821220	
SPECint (mean/stdev)	9.182250		5.938455	
SPECfp (mean/stdev)	10.626600		7.390154	
TECH (mean/stdev)	10.186375		6.819019	

The errors are reduced for the SPEC benchmarks, but overall error for the technical workloads do not improve significantly. 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. Writebacks form a large percentage of the L2 traffic for the benchmarks. 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 calculation. This is for two reasons. First, the writebacks generally occur in parallel with the servicing of the miss so they do not contribute to the latency of the operation. Second, writebacks are not modelled in HLS. 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 technical loops reveals that the miss rates for loads and stores are quite different. In SAXPY, for example, both loads miss to the L1, but all stores hit.

Because of these considerations, in S-HLS, each basic block maintains L1 and L2 miss rates for both loads and stores. Here is a run with the addition of this information:

Table 17: S-HLS using Basic Block Load and Store Cache Miss Rates

Benchmark	SimpleScalar IPC	S-HLS IPC	S-HLS Sigma	Error (%)
gcc	0.9096	0.883598	0.028198	2.857
perl	1.1610	1.178220	0.029753	1.482
m88ksim	1.4586	1.347716	0.040249	7.601

Table 17: S-HLS using Basic Block Load and Store Cache Miss Rates

Benchmark	SimpleScalar IPC	S-HLS IPC	S-HLS Sigma	Error (%)
ijpeg	1.8449	1.698654	0.134561	7.926
vortex	0.9422	0.862149	0.019887	8.495
compres	1.1290	1.245927	0.063531	10.355
go	0.9644	0.985707	0.034733	2.208
li	1.5442	1.450064	0.057884	6.095
tomcatv	0.9314	0.873802	0.020739	6.183
su2cor	1.0188	0.912161	0.015182	10.466
hydro2d	1.0888	1.119110	0.095286	2.782
mgrid	1.6826	1.729715	0.058554	2.800
applu	1.3779	1.474173	0.082393	6.985
turb3d	1.6288	1.772929	0.089612	8.847
apsi	1.1737	1.386719	0.121742	18.148
wave5	1.1619	1.123232	0.054697	3.326
fpppp	0.7117	0.672129	0.008628	5.559
swim	1.1813	1.085897	0.041301	8.075
saxpy	1.2878	1.230181	0.089998	4.473
sdot	1.2412	1.191536	0.118701	4.000
sfill	2.4769	2.480206	0.003007	0.130
scopy	1.5940	1.823195	0.398821	14.377
ssum2	1.4090	1.412356	0.155133	0.236
sscale	1.3875	1.377914	0.105713	0.689
striad	1.4165	1.267905	0.149000	10.489
ssum1	1.6621	1.282134	0.113065	22.859
All (mean/stdev)	6.824731		5.349703	
SPECint (mean/stdev)	5.877375		3.082663	
SPECfp (mean/stdev)	7.317100		4.376375	
TECH (mean/stdev)	7.156625		7.632725	

All benchmarks improved, but a few of the technical loops still have errors greater than 10%. The problem is that the technical loops need information about 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. Normally the critical paths are randomized to less critical paths, and performance is overestimated.

One solution is to keep overlap statistics. This will solve this particular problem, but does not provide for the modelling of additional memory operation features. In S-HLS, we chose to try to identify the streams that the memory operations define. We track the load and store L1 hit, L1 miss or L2 hit information for about 100 cycles near the end of the one billion instruction run. We use this stream information in the sequence order of the instructions in the basic block to determine the miss characteristics of the memory operations during simulation. Note that this is a simplistic way to operate. As future work, more sophisticated stream information could be gathered and made more compact. Also, the stream access is not optimal, i.e. we start collecting miss values at a random cycle and stop collecting them about 100 cycles later. The following results are produced:

Table 18: S-HLS using Detailed Stream Information

Benchmark	SimpleScalar IPC	S-HLS IPC	S-HLS Sigma	Error (%)
gcc	0.9096	0.871076	0.043900	4.234
perl	1.1610	1.176667	0.022471	1.348
m88ksim	1.4586	1.327717	0.033566	8.972
ijpeg	1.8449	1.725202	0.180483	6.487

Table 18: S-HLS using Detailed Stream Information

Benchmark	SimpleScalar IPC	S-HLS IPC	S-HLS Sigma	Error (%)
vortex	0.9422	0.860189	0.029202	8.703
compres	1.1290	1.249317	0.070983	10.655
go	0.9644	0.986819	0.016969	2.323
li	1.5442	1.479090	0.075880	4.215
tomcatv	0.9314	0.863170	0.017195	7.324
su2cor	1.0188	0.914685	0.017860	10.218
hydro2d	1.0888	1.059177	0.103037	2.719
mgrid	1.6826	1.777686	0.027317	5.650
applu	1.3779	1.435848	0.129544	4.204
turb3d	1.6288	1.727198	0.146318	6.040
apsi	1.1737	1.323856	0.097410	12.792
wave5	1.1619	1.120683	0.050583	3.546
fpppp	0.7117	0.664898	0.011435	6.575
swim	1.1813	1.065523	0.049969	9.800
saxpy	1.2878	1.375968	0.009551	6.845
sdot	1.2412	1.302454	0.026077	4.934
sfill	2.4769	2.480979	0.002906	0.161
scopy	1.5940	1.606451	0.044572	0.780
ssum2	1.4090	1.397683	0.041592	0.802
sscale	1.3875	1.365524	0.018724	1.582
striad	1.4165	1.493669	0.026799	5.446
ssum1	1.6621	1.632211	0.048951	1.797
All (mean/stdev)	5.313538		3.371519	
SPECint (mean/stdev)	5.867125		3.149809	
SPECfp (mean/stdev)	6.886800		3.044980	
TECH (mean/stdev)	2.793375		2.383359	

All benchmarks are improved, but the technical loops improved significantly. The combined techniques seem to give low errors and consistent results across all three suites.

The system as described above is flexible. Address or data information could be maintained if specific branch prediction or value prediction studies were to be undertaken. As future work, the branch predictor statistics could be maintained on a basic block granularity. This would probably improve correlation for *compress* [NUSS01]. There are many other workload characteristics that could be modelled [JOHN99]. Those would more or less help decrease correlation errors.

We also ran S-HLS with a dispatch window of 32.

Table 19: S-HLS using a Dispatch Window of 32

Benchmark	SimpleScalar IPC	S-HLS IPC	S-HLS Sigma	Error (%)
gcc	0.9178	0.896867	0.022793	2.279
perl	1.1627	1.217511	0.024213	4.713
m88ksim	1.4624	1.384089	0.029517	5.353
ijpeg	2.0443	2.038928	0.164339	0.260
vortex	0.9501	0.879853	0.010600	7.392
compres	1.2126	1.373960	0.094769	13.305
go	0.9928	1.029788	0.025797	3.724
li	1.5804	1.515755	0.067362	4.089

Table 19: S-HLS using a Dispatch Window of 32

Benchmark	SimpleScalar IPC	S-HLS IPC	S-HLS Sigma	Error (%)
tomcatv	0.9502	0.881076	0.011191	7.273
su2cor	1.0358	0.923782	0.018508	10.813
hydro2d	1.2736	1.319106	0.049834	3.572
mgrid	1.8161	2.219689	0.035524	22.221
applu	1.6507	1.811340	0.169709	9.730
turb3d	1.9772	2.001863	0.156067	1.246
apsi	1.3412	1.638326	0.112274	22.152
wave5	1.2482	1.212617	0.056180	2.849
fpppp	0.7143	0.674981	0.009063	5.503
swim	1.3620	1.302611	0.053179	4.359
saxpy	1.6621	1.872126	0.059424	12.635
sdot	1.7558	1.851868	0.045019	5.470
sfill	2.4771	2.481336	0.002432	0.170
scopy	1.6901	1.603003	0.042466	5.152
ssum2	1.6510	1.660596	0.045759	0.580
sscale	1.7469	1.788622	0.039245	2.387
striad	1.8094	1.921933	0.055989	6.218
ssum1	1.8126	2.016871	0.058391	11.268
All (mean/stdev)	6.719731		5.727280	
SPECint (mean/stdev)	5.139375		3.658480	
SPECfp (mean/stdev)	8.971800		7.181607	
TECH (mean/stdev)	5.485000		4.288354	

Overall error is still low, but a few benchmarks, including *mgrid* and *apsi*, do worse.

9. Implementation Costs

To achieve the performance in Table 18, a significant amount of information must be generated and maintained about the dynamic basic blocks that make up 99% of the blocks encountered in the first one billion instructions of the benchmark. We calculate the cost in bytes as function of the number of basic blocks (NBB), the average length of the basic blocks (LBB), the number of loads and stores in the basic block (NLS), and the number of exact stream data used (NSD). The information is summarized in Table 20. NSD is NLS x 100 = 4.3 x 100 = 430 in our runs. Averaged over all of the benchmarks (see Table 14), NBB is 120.2, LBB is 12.4, and NLS is 4.3. The error reduction is the average reduction in correlation error as each technique augments the previous technique using Tables 15 through 18. If the branch predictability is maintained, an additional cost of NBB x 1 x 1 x 4 bytes would need to be added.

Table 20: Error Reduction Costs

Information	Cost Formula	Average Cost per Benchmark (in Bytes)	Reduction in Error using the Technique	Cost in Bytes per %Error Reduction	Comment
Cumulative Frequencies	NBB x 4 bytes	480	26.6%	46.1	1 float/BB
Instruction Sequences	NBB x LBB x 1/2 byte	746			< 16 op types
Dependency Statistics	NBB x LBB x 2 x 1 byte	2980	16.31%	92.7	any dep 0/1 < 255
Cache Miss Statistics	NBB x 4 x 4 bytes	1924	32.14%	59.9	ld/st l1/l2 miss rates
Stream Data	NBB x NLS x 1/4 byte x NSD	55562	47.16%	1178.16	2 bits per datum

Clearly, including detailed stream data is inefficient on average compared to using the other techniques. Note that particular benchmarks, such as the technical loops, have drastically smaller NSD values because the average loads and stores per basic block is small (see Table 14), and the amount of data necessary for complete characterization of the stream is small (see Table 1). In Section 12, we will explore additional techniques to reduce information storage.

In future work, a study of the effect of the relative error versus absolute error may indicate that errors need not be as low as those achieved here to carry out useful studies, and therefore not all techniques need to be applied. This may be particularly true in early design studies [EECK03].

10. Higher Level Modelling: Basic Block Maps

The basic block graph structure as used in HLS was shown to give equivalent simulation results as a single loop of basic blocks. But the graph structure can be extended to model multiple phases of a program more accurately using *basic block maps*. A basic block map is an ordered list of the usage frequency of the set of basic blocks over the cycles of the program. For example, a basic block map might describe a program in the following way:

basic block numbers {2..23, 42, 56..96} with weights {0.50, 0.25, 0.25} used for the first 15% of cycles
basic block numbers {1..51, 65, 75..100} with weights {0.15, 0.70, 0.15} used for the next 23% of cycles
basic block numbers {2, 3, 89, 91} with weights {0.20, 0.20, 0.60} used for the last 62% of cycles

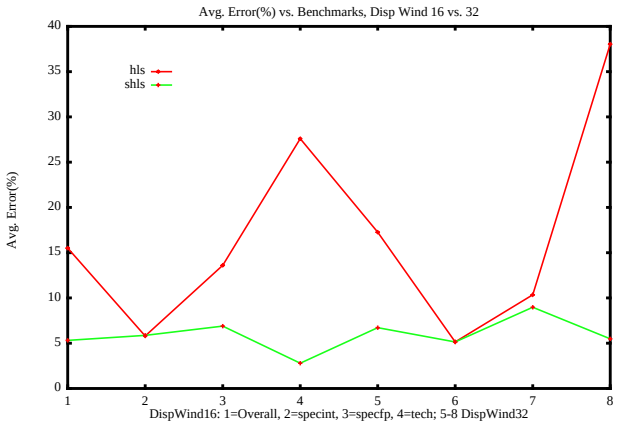
This framework models programs more accurately as phases in program execution are encountered. Future research will evaluate different mapping strategies: mapping blocks into groups inside the graph structure, mapping blocks by changing the graph structure itself, or a combination of both.

As an example, a simple code created from the concatenation of *sdot* and *ssum1* from Table 1 has correlation errors of 39.4% and 19.3% in HLS and S-HLS, respectively. The error in S-HLS is due to the way basic blocks are generated. 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. Using a basic block map inside the global graph loop with the first half of the blocks mapped to *sdot* and the second half mapped to *ssum1*, an error of 1.6% is obtained. Obviously, additional workload information must be generated and maintained to support basic block maps.

In HLS, the large error is due to an effect similar to the basic block size aliasing discussed in Section 8. HLS has no means of separating out the two kinds of blocks and executing the phases of the program, so large errors result.

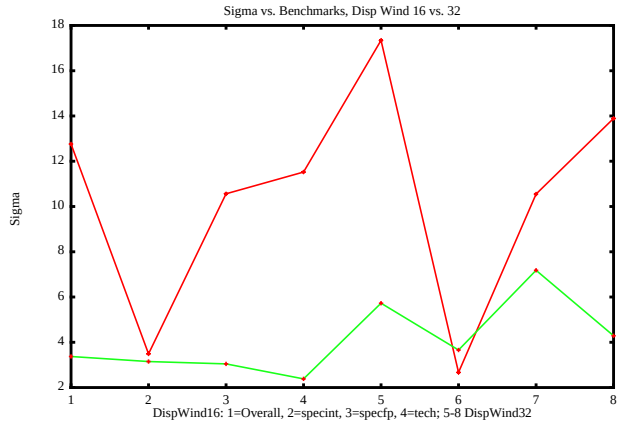
11. Comparison of HLS to S-HLS

The following charts give a visual comparison of HLS to S-HLS using the SPEC95 benchmarks and the technical loops. In this figure, the correlation error means for all benchmarks, SPECint95, SPECfp95, and the technical loops are compared given a dispatch window of 16 or a dispatch window of 32. S-HLS is run with all techniques enabled. The chart shows that S-HLS correlates as well or better than HLS.

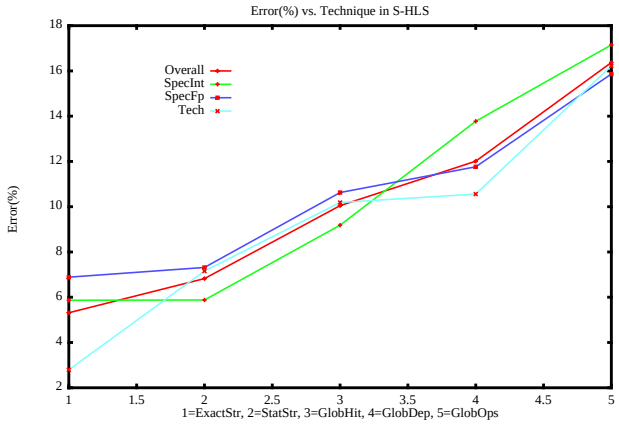


Note that S-HLS seems more consistent overall. The next chart gives the standard deviation for the same runs. S-

HLS has a more moderate deviation over all benchmarks than HLS. The deviation worsens somewhat for the larger dispatch window.



The next chart plots the correlation error increase over the benchmark suites as the various strategies for reducing error are removed.



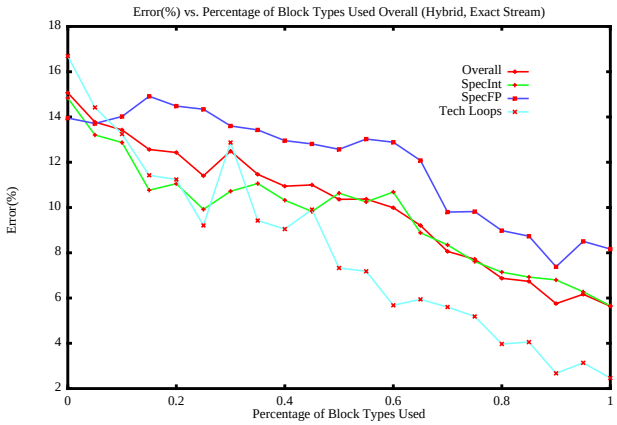
12. Basic Block Information Reduction Techniques

In previous sections, our first strategy for reducing error was to create and maintain a list of all basic blocks that account for the top 99% of the dynamic basic blocks encountered in a run, each with its own instruction sequence. We then added more detailed information to each basic block to reduce error, such as dependence as stream information.

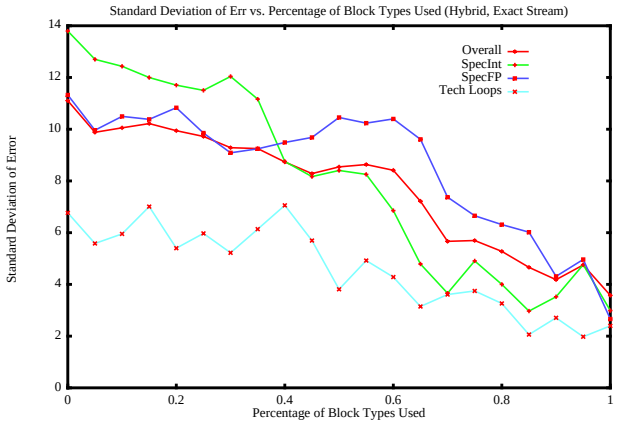
In this section we are interested in techniques to reduce the number of basic blocks that are stored. If we somehow merge or combine the information in basic blocks that have the same size, we are left with statistics similar to those in [NUSS01]. One technique we used above was to combine basic blocks that have the same size and instruction sequences but different dependency information. An alternative is to merge blocks with very similar instruction sequences and dependence information, but slightly different sizes. The techniques to do that would include pattern matching between the instruction sequences in basic blocks and ranking the match results, similar to techniques currently used for mapping the human genome. Another approach would be to merge basic block information, but only on a graduated scale as the frequency of the blocks decreases, i.e. the most frequent blocks are more exact and unique, while the less frequent are merged together either by block size, dependencies or stream. Those studies are left for future research.

Instead of a graduated scale, we can simply merge information for fewer than the top 99% of the dynamic basic blocks encountered in a run. The basic block type information is used for a percentage of the top dynamic blocks

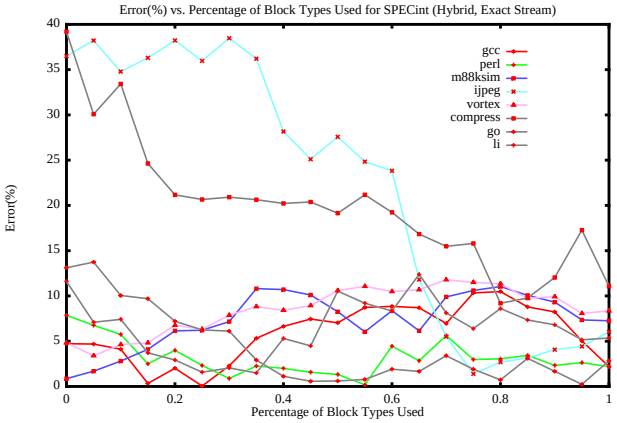
encountered and the baseline global information is used for the rest. The following figure shows the mean correlation error for the three benchmark suites and overall as the percentage of basic blocks that use dynamic block information is lowered.



On average, errors decrease from 15.07% to 5.63% when all dynamic block information is used. The technical loops, in which one basic block accounts for over 99% of all basic blocks encountered dynamically, are improved the most. For SPEC, errors are less than 10% when using 65% of local basic block information. The following figure shows the standard deviation of the error as the local block information use increases.



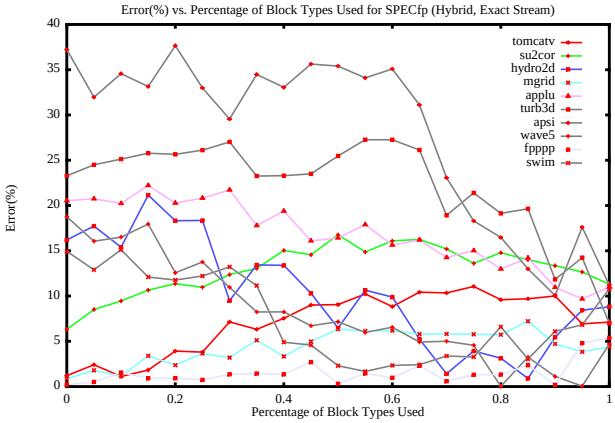
For particular benchmark suites, the situation becomes more complex. The following figure shows the results for



the SPECint benchmarks. *Perl* and *go* have very low error, below 5%, when using 35% or more of the dynamic block

information. *Gcc*, *m88ksim*, *vortex* and *li* always have errors below 10%. However, *ijpeg* needs 70% of the dynamic block information to achieve below 10% error. For *compress*, error ends up relatively high due to the use of global branch predictability information for all blocks. This could be improved by using local branch predictability information for each basic block [NUSS01].

For the SPECfp, the situation is more complicated. *Tomcatv*, *mgrid* and *fpppp* do well using no dynamic block



information. *Swim*, *wave5* and *hydro2d* do well using 40% of dynamic block information. *Su2cor*, *applu*, *turb3d*, and *apsi* need 90% or more to get consistently low errors.

13. Statistical Simulation Using Benchmark Suites

The previous sections demonstrate that the correlation of benchmarks using statistical simulation can be done at many levels. The workload information used can be as detailed as necessary to obtain the required level of error correlation. Programs can be modelled at a high level using basic block maps. The basic blocks may contain detailed information or no information.

Conversely, the point is made that low-information simulations may work for particular benchmarks, or for benchmark suites, but may not work for other benchmarks or suites.

One exciting area open to investigation is the use of information about workload characteristics found using statistical simulation to predict trends in future workloads. By analyzing at some level of detail the information in the basic blocks or basic block maps of benchmark suites over time, and extrapolating the trends to the future, perhaps predictions about those characteristics can be made.

14. The Case for Synthetic Benchmarks based on Statistical Simulation Information

So far we have talked about statistical simulation as a way to quickly simulate workloads on an execution engine during early system design phases. We abstracted the workload characteristics to some level of accuracy, and then we created a graph of the resulting basic blocks. We also talked about creating and using basic block maps to abstract the phase behavior of a benchmark. We then argued that the graph of basic blocks is really an abstraction of the original benchmark or program that converges to the correct IPC quickly.

In this section, we argue that the graph structure, really an abstracted program, can be used to generate synthetic benchmarks that exhibit the some or all of the same characteristics of the original program when executed on a processor.

There are many reasons that one might want to generate synthetic benchmarks from real code [WONG88]. Here are a few:

1) Code Abstraction: Since the synthetic benchmark is an abstraction of a user program or benchmark, the functional details of the code are hidden from the user of the synthetic benchmark, while the runtime characteristics of interest are similar. This would allow a company to synthesize benchmarks of their most important code and then disseminate the benchmarks. This permits others to assess the code's behavior on their computer system, or to change their current system designs to accommodate the benchmark. Benchmarks based on mission-critical industry-standard code would proliferate to the academic and design communities.

2) Portability: Some benchmarks, like TPCC, are very difficult to get up and running quickly on a particular sys-

tem. In early processor design phases, it is difficult to tell how the design will fare on the benchmark before the machine is built. If these complex benchmarks can be abstracted to simple benchmarks that converge quickly to a result, their workload characteristics can be tested earlier in the design phase.

3) Combining Workloads: Synthetic benchmarks can be created that are built from the characteristics of multiple benchmarks or benchmark suites. This would facilitate, for example, the creation of a benchmark exhibiting average SPECint behavior.

4) Simulating Future Workloads on Current Designs: As mentioned above, one exciting are open to investigation is the use of information about workload characteristics found using statistical simulation to predict trends in future workloads. By analyzing at some level of detail the information in the basic blocks or basic block maps of benchmark suites over time, and extrapolating the trends to the future, perhaps predictions about those characteristics can be made. Then, benchmarks with those characteristics can be synthesized and studied on current processor designs.

15. Overview of Synthetic Code Generation in S-HLS and the Major Associated Problems

A code generator was built into S-HLS that uses the data structures and information that already exists for statistical simulation. At a high level, the code generator takes the basic blocks of instructions and outputs a single module of c-code which contains calls to assembly-language instructions (in the PISA language). Each instruction, including branches, in each basic block maps to a single assembly language instruction call in the c-code. The assembly language instructions are each demarcated by labels in the code.

The code generator is simplified by the fact that the basic blocks are configured into a single loop as described previously, since the original HLS graph structure made no difference to the error correlation. Therefore the default model is to generate c-code that executes all instructions in one giant loop for a fixed number of iterations. This has implications for the branch predictor locality, since both the taken and not taken paths of any branch are configured to be to the next sequential basic block.

There are several possible ways to model branch predictability accurately at the basic block granularity. In one scenario, a basic block is broken into a collection of basic blocks based on the predictability of its branch. Two sets of basic blocks are created, the taken set and the not-taken set, and the number of items in each set is determined by the total number of basic blocks multiplied by the probability that a branch is taken or not-taken. Then in the code generated for a basic block which jumps to the block, a simple unsigned counter accessing a jump table is used to access one of the basic blocks from the set. Over many iterations, even though the branch predictions will most likely not match those of the original code, the instruction access profile will match that of the original code. One disadvantage of this scheme is that studies targeting branch predictor design to improve performance of the original code can not be undertaken using the synthetic benchmark. Another disadvantage is that additional code (the jump table) is added to the synthetic benchmark which did not exist in the original code, skewing the instruction mix and IPC results. Future research can search for low-overhead predictability generating functions which, when implemented in the synthetic code, cause the branch predictability to match that of the original code. For the present, we do not try to solve the branch predictability problem and restrict our attention to codes which exhibit high branch predictability.

Another problem that must be solved is the problem of generating code that exhibits the same icache miss rate as the original code. One solution is to duplicate enough basic blocks such that the miss rate obtained matches the miss rate of the original code. Of course, the duplication must be done such that instruction mix and IPC results match the original code. A simple way to achieve that is to only add multiples of the entire synthetic code. For the present, we do not try to solve the icache miss rate problem and restrict our attention to codes with low icache miss rates.

Another problem that we do address in S-HLS is the dcache miss rate problem. The memory operations in the synthetic code must access data in the same manner as the original code so as to generate hits and misses in the same way. As described above, S-HLS tracks the streaming behavior of the memory operations. This information is used to determine the stride of the address incrementer that feeds each load or store instruction. Knowledge of the L1 and L2 cache blocksize is needed to properly assign the strides. For a stream of L1 hits, the increment is 0. For a stream that always misses in the L1 but hits 50% in the L2, if the L1 blocksize is 32 bytes and the L2 blocksize is 64 bytes, then the increment to the memory operation is 32. Unless the streams are simple, it is difficult to assign a stride which correctly matches the miss rates in the original code. Future research can search for stream generating functions which more closely match miss rates. For the present, we restrict our attention to codes which exhibit simple streams.

The technical loops are ideal candidates for a first study of benchmark synthesis using statistical simulation. As demonstrated in previous sections, modelling the technical loops is not trivial, yet the amount of code in the loop kernels is minimal, amounting to no more than a dozen instructions when compiled. In addition, the loops exhibit high

branch predictability, so that the branch predictability problem above does not have to be solved immediately. They also have low icache miss rates, so the icache miss rate problem also does not have to be solved immediately. They also lend themselves to studies of the dcache miss rate problem because they exhibit simple stream behavior.

16. Benchmark Synthesis using S-HLS

This section describes code generation in S-HLS. There are two major phases: graph analysis and code generation. The goal is to take the statistically-generated instructions and dependence graph and create an object code that can be translated into c-code with assembly-language calls that represent the workload. Each instruction is mapped one-to-one and onto one assembly language call. Each instruction in a basic block has a unique output variable name. There are several reasons for using c-code instead of generating an assembly language program. First, the code is more portable to other machines. Second, c-code headers and variable declarations are easy to generate. Third, the code is less prone to errors. One disadvantage is that a compiler may remove instructions that do not produce results that are used later unless special precautions are taken, as described below. The instructions may not produce results that are used later because the instructions were statistically generated, and no real function is being performed.

Starting with graph analysis, for each basic block in the graph, the input data structures are labelled according to their functionality. If an instruction input was created which had no dependency, it is labelled as an immediate. Branches with immediates are paired with previous compatible instructions. For convergence purposes, the graph will be traversed multiple times, and since we know we might access streams of memory, loop limits are created which guarantee memory will not be stepped on.

All instruction input dependencies are then created. The starting dependence is the dependent instruction chosen for statistical simulation. The issue is *compatibility*: if the dependency is not compatible with the input type of the dependent instruction, then another instruction must be chosen. The algorithm is to move forward and backward from the starting dependency through the list of instructions in sequence order until the dependency is compatible. An alternative would be to have an algorithm to change the type of either the dependency or the dependent instruction to match the dependency. For this study we chose not to change any instruction types except in the case of memory location access counters and the loop counter, as described below. An instruction is allowed to be dependent on its own output. If more than 15 instructions are checked and a dependency cannot be made compatible, the program ends with an error message. The one exception is a store that is operating on data which was not generated in the program but external to it. An additional variable of the correct data type is created for the store.

Table 21 shows the compatibility of instructions in the PISA assembly language. The *dependent-inputs* column gives the PISA instruction inputs that are being tested for compatibility. For loads and stores, the memory location access register must be an integer type. When found, it is labelled as a memory access counter for special processing during the code generation phase. When all instructions have compatible dependencies, a search is made for an additional integer instruction which is then labelled as the loop counter. The branch in the last basic block in the list uses the loop counter to determine when the program is complete. The number of loops is chosen large enough to assure IPC convergence. In general, this means the number of loops must be larger than the longest memory access stream pattern of any memory operation. In practice, the number of loops does not have to be large to satisfy simple stream access patterns.

Table 21: Dependence Compatibility Chart

dependent instruction	dependent inputs	compatible with:	comment
Int	0/1	Int, Ld-Int	
Flt	0/1	Flt, Ld-Flt	
Ld-Int/Flt	0	Int	dep0 is addr resolution input
St-Int	0	Int, Ld-Int	dep0 must store int
St-Flt	0	Flt, Ld-Flt	dep0 must store float
St-Int/Flt	1	Int	dep1 is addr resolution input
Br-Int	0/1	Int, Ld-Int	
Br-Flt	0/1	Flt, Ld-Flt	

The next phase of synthesis is code generation. First, the c-code main header is generated. Next, sequencing

through all of the instructions, we generate the necessary variable declarations, including special *register-increment* variables, the loop variable, and pointers to the correct memory type for the memory access instructions. Next, we generate *malloc* declarations for the memory access instructions with size based on the number of iterations the program will execute through the instructions. As an alternative, static memory can be assigned instead of using malloc, but then the code size increases substantially at compile time.

Next, we generate initializations for each register-increment variable to a value dictated by the stream type accessed by its memory operations. The register-increment variable is assigned the value that will be added to the memory location counter after each access, i.e. the stride of the memory access. The following table, Table 22, gives the initialization value of the stride in 4 byte increments given the stream's particular L1 and L2 hit rates. The table was generated based on an L1 line size of 32 bytes, and an L2 line size of 64 bytes.

Table 22: L1 and L2 Hit Rates versus Stride (in 4 Byte increments)

L1 Hit Rate	L2 Hit Rate	Stride
0.0000	0.5000	8
0.1172	0.4956	9
0.1250	0.5000	7
0.2422	0.4948	10
0.2500	0.5000	6
0.3672	0.4938	11
0.3750	0.5000	5
0.4922	0.4923	12
0.5000	0.5000	4
0.6172	0.4898	13
0.6250	0.5000	3
0.7422	0.4848	14
0.7500	0.5000	2
0.8672	0.4706	15
0.8750	0.5000	1
0.9922	0.0000	16
1.0000	0.0000	0

So, for a stream that always hits in the L1, a stride of 0 is chosen. For a stream that hits half the time in the L1 and half the time in the L2, a stride of 4 x 4 = 16 is used. Note that not all combinations are possible, so for a particular stream the stride that gives the L1 hit rate closest to the L1 hit rate of the statistically generated stream is chosen. As future work, multiple instructions in the code could be used to create more complex stream behavior to match additional L1 and L2 hit rate characteristics.

Next, the loop counter variable initialization is generated, and is set equal to the number of times the instructions will be executed.

Next, the instructions are generated as c-code calls to PISA assembly language instructions. Each call is given an associated unique label. Memory access counters are generated using *addu*, adding in the increment-register variable to its variable value. The loop counter is generated as an *addi* with *-1* as the decrement value. Floating point operations can have long or short latency characteristics. Long latency operations are generated using *mul.s* and short latency operations are generated using *add.s*. Loads and stores use *lw, sw, l.s* or *s.s* depending on the type. Currently double types are not handled, but are easily added for future work. Branches use the *beq* type, and can have either integer or float operands. While strictly speaking assembly language is not portable, mapping the simple PISA assembly language calls to any assembly language is a straight-forward process.

Next, code is generated to print out some output variables depending on a switch value. At runtime, the switch variable will never be set. This trick keeps assembly language instructions that produce no results that are used later from being optimized out of the compilation. For now, all instruction variables are put into this print block. This has implications for register spills for codes that are more than a few basic blocks long. As future research, either the compiler can be flagged to not remove the assembly language instructions or variables will be consolidated in the generated code and only a minimum number will be put in the print block.

Next, code to free the malloced memory is generated, and finally a c-code footer is generated.

17. Benchmark Synthesis Results for the Technical Loops

The technical loops were processed through S-HLS using all of the techniques to reduce correlation error described in previous sections. Code generation was enabled for those runs and c-code was produced using the synthesis process described above. The c-code was then cross-compiled for the PISA language using *gcc* version 2.95.3 with flag *-O* and a binary was generated. The binary was then run through SimpleScalar. It should be noted that, since the code represents a program and is not the original program, the overflow and underflow errors in SimpleScalar were changed to warnings to prevent early program exits. In the following table, Table 23, the SimpleScalar IPC of the original code is shown, followed by the IPC of the synthetic code, followed by the error.

Table 23: Correlation Error for Synthetic Benchmarks

benchmark	SimpleScalar IPC	Synthetic Benchmark IPC	Error
saxpy	1.2878	1.2220	5.4%
sdot	1.2412	1.2342	0.8%
sfill	2.4769	2.3853	3.6%
scopy	1.5940	1.5610	1.9%
ssum2	1.4090	1.3992	0.7%
sscale	1.3875	1.2943	7.9%
striad	1.4165	1.3370	5.6%
ssum1	1.6621	1.6482	0.6%

The mean error is 3.31%, but several are above 5%. The major sources of error include the error in the workload characteristics from which the code was generated and the error in the code generation process itself. As an example of the latter, in some cases instructions are used as memory operation address registers that were simple adders in the original code. The IPC can be subtly affected by these small differences in functionality.

One area of future research will investigate how to take the synthetic benchmark and make it more accurate. If the synthetic benchmark is run through SimpleScalar, and then that output is run through SimpleScalar again, and so on,

As future work, benchmark synthesis will be attempted on more general-purpose workloads. Challenges to accuracy include the branch predictability problem and the icache miss rate problem as previously described. The system provides a framework from which these problems can be investigated.

18. Conclusions

In this report, we show that accurate statistical simulation of technical loops requires more information than previously modelled. We show that simulation of code blocks with a granularity on the order of the basic block improves accuracy for technical loops and does not decrease accuracy for other workloads. The cost of the additional information is quantified.

Higher-level basic block maps are proposed to model program phases and achieve higher accuracy. The basic block maps, implemented as a graph structure, represent an actual workload and achieve good correlation accuracy. We show how to synthesize benchmarks from the graph. To do this, memory accesses are modelled as streams in the synthetic benchmark. We give correlation results versus a cycle-accurate simulator for the technical loops. Ideas for synthesizing more accurate statistical benchmarks are discussed.

The system provides a flexible framework from which statistical simulation and benchmark synthesis can be investigated.

19. References

[AGAR00] V. Agarwal, M. S. Hrishikesh, S.W. Keckler and D. Burger, “Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures,” IEEE Symposium on Computer Architecture, 2000, pp. 248-259.

[BURG97] D. C. Burger and T. M. Austin, “The SimpleScalar Toolset,” Computer Architecture News, 1997.

[CARL98] R. Carl and J. E. Smith, “Modelling Superscalar Processors Via Statistical Simulation,” Workshop on Performance Analysis and Its Impact on Design, June 1998.

[DIEF99] K. Diefendorff, “Power4 Focuses on Memory Bandwidth,” Microprocessor Report, October 6, 1999.

[EECK03] L. Eeckhout, Accurate Statistical Workload Modelling, Ph.D. Thesis, Universiteit Gent, 2003.

[HUK01] J. Huk, S. W. Keckler and D. Burger, “Exploring the Design Space of Future CMPs,” IEEE Conference on Parallel Architectures and Compilation Techniques, Oct. 2001, pp. 199-210.

[JOHN90] M. Johnson, Superscalar Microprocessor Design, PTR Prentice-Hall, 1990.

[JOHN99] L. K. John, P. Vasudevan and J. Sabarinathan “Workload Characterization: Motivation, Goals, and Methodology,” Proceeding of the IEEE Workshop in Workload Characterization, October 1999.

[JOSH02] 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.

[LEEC98] C. Lee and M. Potkonjak, “A Quantitative Approach to Development and Validation of Synthetic Benchmarks for Behavioral Synthesis,” IEEE International Conference on Computer Aided Design, 1998, pp. 347-350.

[NUSS01] S. Nussbaum and J. E. Smith, “Modelling Superscalar Processors Via Statistical Simulation,” Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, September 2001, pp. 15-24.

[NUSS02] S. Nussbaum and J.E. Smith, “Statistical Simulation of symmetric Multiprocessor Systems,” Proceedings of the 35th Annual Simulation Symposium, April 2002, pp. 89-97.

[OSKI00] 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.

[OSKI03] http://www.cs.washington.edu/homes/oskin/tools.html

[SANK03] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, “Exploiting ILP, TLP, DLP with Polymorphous TRIPS Architecture,” Proceedings of the 30th Annual International symposium on Computer Architecture, June 2003.

[SMIT95] J. E. Smith and G. S. Sohi, “The Microarchitecture of Superscalar Processors,” Proceedings of the IEEE, vol. 83, no. 12, December 1995, pp. 1609-1624.

[SOHI03] http://www.cs.wisc.edu/~mscalar/simplescalar.html

[STRE03] http://www.cs.virginia.edu/stream/ref.html

[TEND02] J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le and B. Sinharoy, “POWER4 System Microarchitecture,” IBM Journal of Research and Development, January 2002, pp. 5-25.

[VERP00] P. Verplaetse, J. Van Campenhout and D. Stroobandt, “On Synthetic Benchmark Gneeration Methods,” IEEE International Symposium on Circuits and Systems, May 28, 2001, pp. 213-216.

[WONG88] W. S. Wong and R. J. T. Morris, Sbenchmark Synthesis Using the LRU Cache Hit Function,” IEEE Transactions on Computers, Vol. 37, No. 6, JUne 1998, pp. 637-645.