Improved Automatic Testcase Synthesis for Performance Model Validation

Robert H. Bell, Jr. †‡
† IBM Systems and Technology Division
Austin, Texas
robbell@us.ibm.com

Lizy K. John ‡
‡ Department of Electrical and Computer Engineering
The University of Texas at Austin
ljohn@ece.utexas.edu

ABSTRACT

Performance simulation tools must be validated during the design process as functional models and early hardware are developed, so that designers can be sure of the performance of their designs as they implement changes. The current state-of-the-art is to use simple hand-coded bandwidth and latency testcases to assess early performance and to calibrate performance models. Applications and benchmark suites such as SPEC CPU are difficult to set up or take too long to execute on functional models. Short trace snippets from applications can be executed on performance and functional simulators, but not without difficulty on hardware, and there is no guarantee that hand-coded tests and short snippets cover the performance of the original applications.

We present a new automatic testcase synthesis methodology to address these concerns. By basing testcase synthesis on the workload characteristics of an application, we create source code that largely represents the performance of the application, but which executes in a fraction of the runtime. We synthesize representative versions of the SPEC2000 benchmarks, compile and execute them, and obtain an average IPC within 2.4% of the average IPC of the original benchmarks with similar average workload characteristics. In addition, the changes in IPC due to design changes are found to be proportional to the changes in IPC for the original applications. The synthetic testcases execute more than three orders of magnitude faster than the original applications, typically in less than 300K instructions, making performance model validation feasible.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques

General Terms

Performance, Design, Verification

Keywords

Automatic Benchmark Synthesis, Synthetic Benchmarks, Benchmarking, Performance Modeling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'05, June 20–22, Boston, MA, U.S.A.
Copyright © 2005 ACM 1-59593-167-8/06/2005…$5.00.

1. INTRODUCTION

The process of verifying that a performance model is accurate with respect to a functional model or hardware is known as performance model validation [5]. Validation is necessary at various points during the design process to minimize incorrect design decisions due to inaccurate performance models. While the relative error of design changes based on inaccurate performance models is often similar to the relative error using accurate models [5][12], the correspondence is not without limit. As functional models are improved in the design phases, or when first pass hardware is obtained, accurate performance models can pinpoint with more certainty the effects of particular design changes.

Prior validation efforts have focused on bandwidth and latency tests, resource limit tests, or micro-tests [6][27][3][18][17][16]. These tests are usually hand-written microbenchmarks that validate the basic processor pipeline latencies, including cycle counts of individual instructions, cache hit and miss latencies, pipeline issue latencies for back-to-back dependent operations, and pipeline bypassing. Black and Shen describe automatic testcases created with up to 100 random instructions [3], not enough to approximate many workload characteristics. Desikan *et al.* use microbenchmarks to validate the Alpha 21267 to 2% error [11]. However, the validated simulator still gives errors from 20% to 40% on the SPEC2000 applications.

Applications themselves cannot be used for performance model validation because of their impossibly long simulation runtimes [5]. Trace sampling techniques such as SimPoint [22] and SMARTS [33] can reduce runtimes in simulators, but the executions still amount to tens of millions of instructions. Statistical simulation can further reduce the necessary trace lengths [8][19][12], but executing traces on functional models or hardware is difficult. Sakamoto *et al.* present a method to create a binary image of a trace along with a memory dump and execute those on a specific machine and a logic simulator [21], but there is no attempt to reduce trace lengths. Hsieh and Pedram synthesize instructions for power estimation [14], but there is no attempt to maintain the machine-independent workload characteristics necessary to represent the original applications [1]. The design community recognizes the need for an automatic way to generate relevant and flexible synthetic benchmarks [23], but no such method has been forthcoming.

The major contribution of this paper is a step toward such a method. We synthesize testcases that are more representative of applications than microbenchmarks and yet converge to results quickly. We use the workload characterization of statistical simulation to capture the dynamic features of the program, adding memory access and branching models. A testcase is generated as C-code, with low-level instructions instantiated as

asm statements. When compiled and executed, the synthetic code reproduces the dynamic workload characteristics of an application, and it can be easily executed on a variety of performance and functional simulators, emulators, and hardware, with significantly reduced runtimes.

The rest of this paper is organized as follows. Section 2 presents the conceptual framework of the testcase synthesis method and some of its benefits. Section 3 describes the synthesis approach in detail. Section 4 presents experimental results. Section 5 presents related work, and the last sections present conclusions and references.

2. REPRESENTATIVE SYNTHETICS

We achieve a degree of representativeness by synthesizing a testcase from a workload characterization of the low-level dynamic runtime characteristics of an application, the same characterization used in statistical simulation [8][19][12][1]. We make the following key observation: the statistical flow graph [19][12][1] is a reduced representation of the control flow instructions of the application – a compact representative program. In addition, the synthetic trace from workload characterization converges to an accurate result in a fraction of the time of the original workload [19][12][1]. We combine the representative trace with novel algorithms for locality structure synthesis to automatically generate a simple but flexible testcase. The testcase is a C-code envelope around a sequence of *asm* calls that reproduce the behavior of low-level basic blocks. The testcase is easily retargeted for use on machines with similar ISAs. For example, transforming a C program with *Alpha asm* calls to *PowerPC asm* calls is straightforward, since both instruction sets follow the RISC philosophy.

Ideally, our testcases would be benchmark replacements, but the memory and branching models used to create them introduce errors (Section 3), making them a solution in the "middle" between microbenchmarks and applications. Many of the application characteristics are maintained (Section 4), but there is much room for future work into more accurate models.

The synthetic traces in statistical simulation have been shown to exhibit representative behavior when executed on program phases [12][1]. Likewise, a testcase to represent an entire program can be created by concatenating together testcases synthesized from each phase. In this work, we demonstrate testcase synthesis on a single phase.

Our C-code envelope increases portability to execution-driven simulators, emulators and hardware. At synthesis-time, user parameters can modify workload characteristics to study predicted trends of future workloads. At runtime, parameters can switch between sections of code, changing the mix of program phases or modeling consolidated programs.

Because we synthesize from low-level workload statistics, we avoid questions of high-level programming style, language, or library routines that plagued the representativeness of the early hand-coded synthetic benchmarks such as Whetstone [10] and Dhrystone [30].

Synthesis using statistics rather than actual source effectively hides the functional meaning of the code and data, and motivates increased code sharing between industry and academia. Many vendors hesitate to share their proprietary applications and data for research. This is particularly true in database, embedded and systems software areas. Figure 1 shows the proposed path to code sharing.

3. SYNTHESIS APPROACH

With reference to Figure 2, the following sections detail the four major phases of synthesis: *workload characterization; graph analysis; register assignment* and *code generation*.

3.1 Workload Characterization

A profile of the dynamic execution of an application produces a set of workload characteristics. This is the same analysis that gives good statistical simulation correlation [12][1]. The most significant characteristics are the original basic block instruction sequences and the instruction dependencies [1], but we also characterize the branch predictabilities and the L1 and L2 I-cache and D-cache miss rates at the granularity of the basic block. Instructions are abstracted into five classes plus sub-types: integer, floating-point (short or long execution times), load (integer or float), store (integer or float), and branch (on integer or float).

There is no separate input dataset for the synthetic testcases. The input dataset manifests itself in the final workload characteristics obtained from the execution profile. While separate testcases must be synthesized for each possible dataset, the automatic synthesis approach makes that feasible.

3.2 Graph Analysis

In the following sections, the analysis of the workload characterization for code synthesis is described.

3.2.1 Instruction Miss Rate and I-cache Model

The number of basic blocks to be instantiated in the synthetic testcase is estimated based on a default I-cache size and configuration (16K entries, 32B blocks, 1-way associativity). We then tune the number of synthetic basic blocks to match the original I-cache miss-rate (*IMR)*. Specific basic blocks are chosen from a walk of the statistical flow graph, as in [12][1]. Usually a small number of synthesis iterations are necessary to match the IMR. The numbers of basic

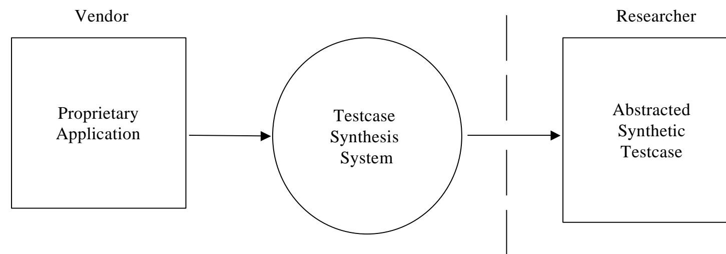

Figure 1. Proprietary Code Sharing using Testcase Synthesis

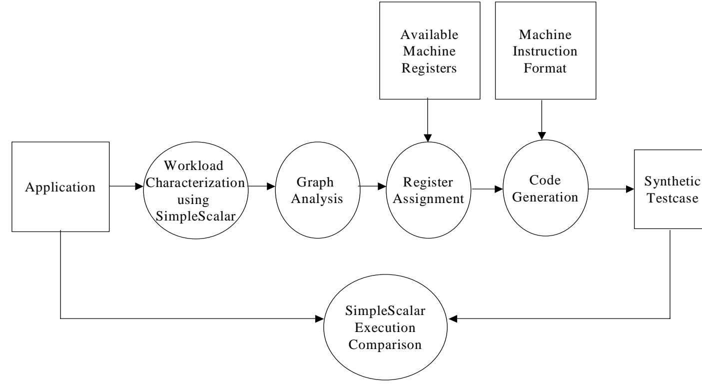

Figure 2. Testcase Synthesis and Simulation Overview

blocks and instructions synthesized for the *Alpha* versions of the SPEC2000 and STREAM benchmarks are shown in Table 3.

3.2.2 Instruction Dependencies and Compatibility

All instruction input dependencies are assigned. The starting dependence is exactly the dependent instruction chosen as an input during statistical simulation. The issue then becomes operand *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. The average number of moves per instruction input is shown in Table 3 for the SPEC2000 and STREAM in column *dependency moves*, and is generally small. In the case of a store or branch that is operating on external data for which no other instruction in the program is compatible, an additional variable of the correct data type is created.

Table 1 shows the compatibility of instructions for the *Alpha* instruction set. The *Inputs* column gives the assembly instruction inputs that are being tested for compatibility. For loads and stores, the memory access register must be an integer type. When found, it is attributed as a memory access counter (*LSCNTR*) for special processing during the code generation phase.

3.2.3 Loop Counters and Program Termination

When all instructions have compatible dependencies, a search is made for an additional integer instruction that is attributed as the loop counter (*BRCNTR*). The branch in the last basic block in the program checks the BRCNTR result to determine when the program is complete. The number of executed loops, *loop iterations* in Table 3, is chosen to be large enough to assure IPC convergence. Conceptually, this means that the number of loops must be larger than the longest memory access stream pattern of any memory operation among the basic blocks. In practice, the number of loops does not have to be very large to characterize simple stream access patterns. Experiments have shown that the product of the loop iterations and the number of instructions must be around 300K to achieve low branch predictabilities and good stream convergence. The loop iterations are therefore approximately *300K/(number of instructions).* This can be tuned with a user parameter. An additional integer instruction is converted to a *cmple* to compare the BRCNTR to zero for the final branch test.

Table 1. Dependence Compatibility Chart

Dependent Instruction	Inputs	Dependence Compatibility	Comment
Integer	0/1	Integer, Load-Integer	
Float	0/1	Float, Load-Float	
Load-Integer/Float	0	Integer	Memory access counter input
Store-Integer	0	Integer, Load-Integer	Data input
Store-Float	0	Float, Load-Float	Data input
Store-Integer/Float	1	Integer	Memory access counter input
Branch-Integer	0/1	Integer, Load-Integer	
Branch-Float	0/1	Float, Load-Float	

3.2.4 Memory Access Model

The LSCNTR instructions are assigned a stride based on the D-cache hit rate found for their corresponding load and store accesses during workload characterization. The memory accesses for data are modeled using the 16 simple stream access classes shown in Table 2. The stride for a memory access is determined first by matching the L1 hit rate of the load or store fed by the LSCNTR, after which the L2 hit rate for the stream is predetermined. If the L1 hit rate is below 12.5%, the L2 hit rate is matched. The table was generated based on an L1 line size of 32 bytes and an L2 line size of 64 bytes, and the corresponding stride is shown in 4 byte increments.

By treating all memory accesses as streams and working from a base cache configuration, the memory access model is kept simple. This reduces the impact on the testcase instruction sequences and dependencies, which have been shown to be critical for correlation with the original workload [1]. On the

Table 2. L1 and L2 Hit Rates versus Stride

L1 Hit Rate	L2 Hit Rate	Stride
0.0000	0.000	16
0.0000	0.0625	15
0.0000	0.1250	14
0.0000	0.1875	13
0.0000	0.2500	12
0.0000	0.3125	11
0.0000	0.3750	10
0.0000	0.4375	9
0.0000	0.5000	8
0.1250	0.5000	7
0.2500	0.5000	6
0.3750	0.5000	5
0.5000	0.5000	4
0.6250	0.5000	3
0.7500	0.5000	2
0.8750	0.5000	1
1.0000	N/A	0

other hand, there can be a large error in stream behavior when an actual stream hit rate falls between the hit rates in two rows, and the simple model is responsible for correlation error when the cache hierarchy changes (see Section 4.3). More complicated models might walk cache congruence classes or pages (to model TLB misses), or move, add, or convert instructions to implement specific access functions. There are many models in the literature that can be investigated as future work, for example [24][29][9][15].

In some cases, we found that additional manipulation of the streams was necessary for correlation of the testcases because of the cumulative errors in stream selection. In Table 3, the *stream factor* multiplies the L1 hit rate taken from the table during each lookup, and if the result is greater than the original hit rate, the selected stream is chosen from the preceding row. This has the effect of reducing overall hit rates for the first load or store fed by an LSCNTR.

Because the dependency analysis may cause several memory access operations to use the same LSCNTR, the overall access rate at the granularity of the basic block may be significantly in error. During synthesis, the overall miss rate for the basic block is estimated as the number of LSCNTRs feeding the block divided by the total number of loads and stores in the block. The *miss rate estimate factor* in Table 3 modifies the calculated miss rate estimate and causes the selected table row for the LSCNTR to change accordingly. Smaller factors increase the basic block miss rate while larger factors decrease it. Usually a small number of synthesis iterations are needed to find a combination of factors to model the overall access rates of the application.

3.2.5 Branch Predictability Model

To model branch predictability, we calculate the branches that will have taken-targets based on the global branch predictability, *BR*, of the original application (assumed greater than 50%). An integer instruction (attributed as the *BPCNTR*) that is not used as a memory access counter or a loop counter is converted into an *invert* instruction operating on a particular register every time it is encountered. If the register is set, the branch jumps past the next basic block in the default loop. The *invert* mechanism causes a branch to have a predictability of 50% for 2-bit saturating counter predictors. The target *BR* is:

$$BR = (F \cdot N + (1 - F) \cdot N \cdot (0.5)) / N$$

where *(1 - F)* is the fraction of branches in the synthetic benchmark that are configured to use the *invert* mechanism, and *N* is the total number of synthesized branches. Solving for *(1 -*

Table 3. Synthetic Testcase Properties

Name	Number of Basic Blks	Number of Instructions	Stream Pools	Code Regis	BP Factor	Stream Factor	Miss Rate Est. Factor	Loop Iterations	Dependency Moves	Actual Runtime (s)	Synthetic Runtime (s)	Runtime Ratio
gcc	850	4585	9	8	1.15	1.07	1.00	51	0.943	6602.85	3.49	**1891.93**
gzip	408	4218	7	10	1.10	1.01	1.00	71	0.188	16695.06	3.88	**4302.85**
crafty	635	4896	9	8	1.15	1.00	1.00	54	0.363	6277.21	3.75	**1673.92**
eon	580	4394	9	8	1.15	1.00	1.00	50	1.209	67064.77	3.15	**21290.40**
gap	268	4193	9	8	1.15	1.00	1.00	62	0.477	5283.60	3.37	**1567.83**
bzip2	311	2515	9	8	1.15	1.00	1.00	109	0.147	10853.61	3.70	**2933.41**
vpr	550	4135	9	8	1.00	1.04	1.00	74	0.977	6470.38	4.32	**1497.77**
mcf	727	4189	9	8	1.05	1.00	1.00	61	0.374	18450.95	3.41	**5410.84**
parser	741	3949	9	8	1.10	1.05	1.00	71	0.567	6459.54	3.95	**1635.33**
perlbmk	606	4263	9	8	1.00	1.00	1.00	61	0.519	22269.29	3.36	**6627.76**
vortex	947	5006	9	8	1.10	1.00	1.00	47	0.466	5919.24	3.39	**1746.09**
twolf	739	4315	9	8	1.04	1.08	1.00	72	0.498	18976.74	4.26	**4454.63**
mgrid	30	3930	7	10	1.20	1.30	0.25	69	9.413	62918.27	3.81	**16513.98**
mesa	619	4292	9	8	1.05	1.00	1.00	64	0.867	56597.98	3.60	**15721.66**
art	450	3762	7	10	0.90	1.50	0.47	73	1.111	89628.10	5.74	**15614.65**
lucas	210	3359	7	10	1.00	1.00	1.00	164	0.691	14697.19	6.28	**2340.32**
ammp	715	4092	13	4	1.00	1.50	0.50	88	0.152	21799.12	9.34	**2333.95**
applu	19	3363	7	10	1.50	1.00	1.10	76	20.293	14149.28	4.00	**3537.32**
apsi	488	4379	7	10	1.00	1.00	0.30	64	7.195	61669.89	3.62	**18693.34**
equake	758	4328	7	8	1.00	1.00	1.00	65	1.468	17989.55	3.59	**5011.02**
galgel	273	3967	7	10	1.00	1.50	0.55	66	0.491	24391.40	4.80	**5081.54**
swim	131	3866	9	8	1.50	1.10	1.00	91	0.579	18347.04	4.97	**3691.56**
sixtrack	621	4173	9	8	1.00	1.02	1.00	84	1.810	21028.38	4.55	**4621.62**
wupwise	176	3656	6	6	1.05	1.03	1.00	224	0.926	18306.39	8.56	**2138.60**
facerec	176	3126	9	8	1.10	0.95	1.05	89	4.622	17156.95	3.79	**4526.90**
fma3d	869	4377	6	8	1.03	1.00	1.00	147	0.149	32235.77	6.59	**4891.62**
saxpy	1	10	2	12	1.00	1.00	1.00	30000	0.000	150.78	3.95	**38.17**
sdot	1	10	2	12	1.00	1.00	1.00	30000	0.000	417.74	3.75	**111.40**
sfill	1	6	1	12	1.00	1.00	1.00	70000	0.333	202.00	3.65	**55.34**
scopy	1	8	2	12	1.00	1.00	1.00	30000	0.000	38.24	3.07	**12.46**
ssum2	1	6	1	12	1.00	1.00	1.00	30000	0.143	53.27	2.26	**23.57**
sscale	1	8	2	12	1.00	1.00	1.00	30000	0.000	38.37	3.10	**12.38**
striad	1	12	3	12	1.00	1.00	1.00	30000	0.000	57.16	4.51	**12.67**
ssum1	1	10	3	12	1.00	1.00	1.00	30000	0.000	91.49	4.24	**21.58**

F), the fraction of branches that must be configured is *2*(1 – BR)*. A uniform random variable over this fraction is used to pick which branches are configured.

The fraction *BR* is sometimes not sufficient to model the branch predictability because of variabilities in the mix of dynamic basic blocks used and the code size. To compensate, the *BP Factor* in Table 3 multiplies *BR* to increase or decrease the number of configured branches. Usually a small number of synthesis iterations are needed to tune this factor.

In an additional implementation, a branch jumps past a user-defined number of basic blocks instead of just one, but this did not result in improved branch predictability. In another implementation, a branch jumps past a user-defined number of instructions in the next basic block. This also did not improve predictability except for *mgrid* and *applu*, which have large average basic block sizes such that jumping past an entire basic block significantly changes the instruction mix. In those cases, the branch jumps past ten instructions of the next basic block.

During synthesis experiments, it was noticed that benchmarks with large average basic block sizes and therefore small numbers of basic blocks in the final synthetic code are prone to have a skewed basic block mix that favors shorter basic blocks. For *mgrid* and *applu*, during basic block selection, if a uniform random variable is greater than an additional factor, set to 0.5 and 0.9, respectively, then the successors of the previous block that are on average longer than 50 instructions are checked first to be included.

When configuring branches, the BRCNTR, *cmple* and BPCNTR instructions must not be skipped over by a taken branch, or loop iterations may not converge or the branch predictability may be thrown off. Code regions containing these attributed instructions are carefully avoided.

3.3 Register Assignment

All architected register usages in the synthetic testcase are assigned exactly during the register assignment phase. Most ISAs, including the *Alpha ISA*, specify dedicated registers that should not be modified without saving and restoring. In practice, not all registers need to be used to achieve a good synthesis result. In our experiments, only 20 general-purpose registers divided between memory access stream counters and code use are necessary. For the benchmarks under study, the number of registers available for streams averages about 8 and for code use about 9 (*stream pools* and *code registers* in Table 3). Three additional registers are reserved for the BRCNTR, *cmple*, and BPCNTR functions.

Memory access streams are pooled according to their stream access characteristics and a register is reserved for each class (*stream pools* in Table 3). All LSCNTRs in the same pool increment the same register, so new stream data are accessed similarly whether there are a lot of LSCNTRs in the pool and few loop iterations or few in the pool but many iterations. For applications with large numbers of stream pools, synthesis consolidates the least frequent pools together (using the most frequent LSCNTR stride among them) until the total number of registers is under the limit.

In practice, a roughly even split between code registers and pool registers improves benchmark *quality*. High quality is defined as a high correspondence between the instructions in the compiled benchmark and the original synthetic C-code instructions. With too few or too many registers available for code use, the compiler may insert stack operations into the binary. The machine characteristics may not suffer from a few stack operations, but for this study we chose to synthesize code without them.

The available code registers are assigned to instruction outputs in a round-robin fashion.

3.4 Code Generation

The code generator of Figure 2 takes the representative instructions, the instruction attributes from graph analysis, and the register assignments and outputs a single module of C-code that contains calls to assembly-language instructions in the *Alpha* language. Each instruction in the representative trace maps one-to-one to a single volatile *asm* call in the C-code. The steps are detailed in the following paragraphs.

First, the C-code *main* header is generated. Then variable declarations are generated to link output registers to memory access variables for the stream pools, the loop counter variable (BRCNTR), the branching variable (BPCNTR), and the *cmple* variable. Pointers to the correct memory type for each stream pool are also declared. *Malloc* calls for the stream data in memory are generated with size based on the number of iterations per program loop. Initializations are generated for each stream pool output register to the head of the data.

The BRCNTR register is initialized to the number of times the instructions will be executed. The instructions are then generated as calls to assembly language instructions. Each call is given an associated unique label. LSCNTRs are generated using *addi* instructions to add the stride to its current register value. The BRCNTR is generated as an *add* of minus one to its register. Long latency floating-point operations are generated using *muls* and short latency operations are generated using *adds*. Loads use *lwz* or *lds,* depending on the type, and similarly for stores. Branches use the *beq* type, and can have either integer or float operands. The basic blocks are analyzed and code is generated to print out unconnected output registers depending on a switch value. The switch is never set, but the print statements guarantee that no code is eliminated during compilation. Code to free the *malloced* memory is generated, and finally a C-code footer is generated.

Table 3 gives the synthesis information for the SPEC2000 and STREAM codes as described in this section. The *runtime ratio* is the user runtime of the original benchmark for one billion instructions (1M for STREAM) divided by the user runtime of the synthetic testcase on various Power3 (400MHz) and Power4 (1.2 GHz) workstations. Variations in runtime reflect network traffic during the runs. Each pass through the synthesis process takes about three minutes on an IBM p270

Table 4. Default Simulation Configuration, Alpha ISA

Instruction Size (bytes)	4
L1/L2 Line Size (bytes)	32/64
Machine Width	4
Dispatch Window/LSQ/IFQ	16/8/4
Memory System	16K 4-way L1 D, 16K 1-way L1 I, 256K 4-way unified L2
L1/L2/Memory Latency+transfer (cycles)	1/6/34
Functional Units	4 I-ALU, 1 I-MUL/DIV, 4 FP-ALU, 1 FP-MUL/DIV
Branch Predictor	Bimodal 2K table, 3 cycle misspredict penalty

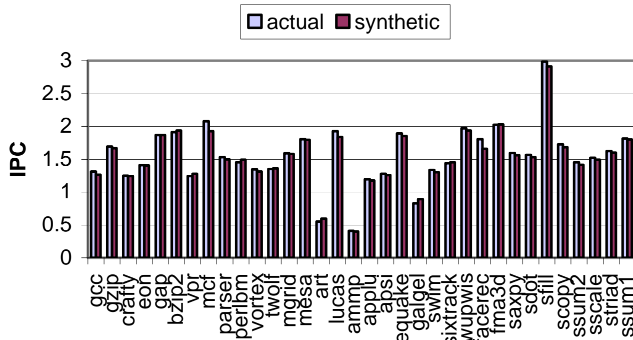

Figure 3. Actual vs. Synthetic IPC

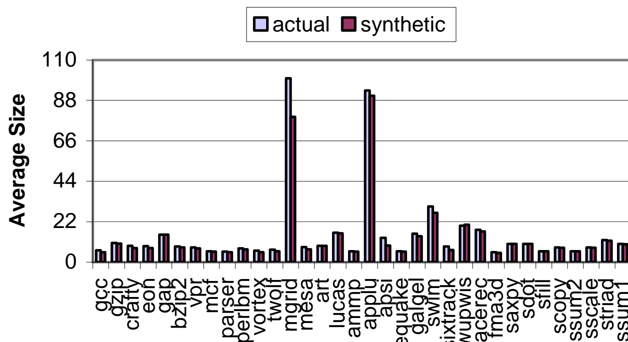

Figure 5. Actual vs. Synthetic Basic Block Sizes

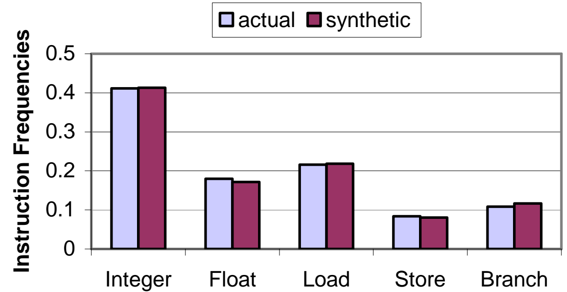

Figure 4. Actual vs. Synthetic Instruction Frequencies

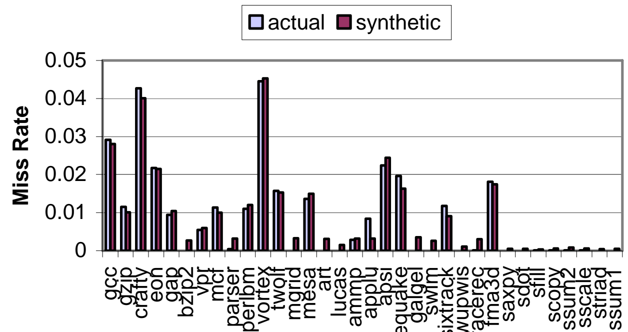

Figure 6. Actual vs. Synthetic I-cache Miss Rate

(400 MHz). An average of about ten passes plus think time were necessary to tune the synthesis parameters for each testcase. All the data in Table 3 took less than five workdays to generate.

4. SYNTHESIS RESULTS

In this section we present the results of the testcase synthesis experiments.

4.1 Experimental Setup and Benchmarks

We use a system [1] modified from HLS [19][20]. SimpleScalar Release 3.0 [7] was downloaded and *sim-cache* was modified to carry out the workload characterization as in [12][1]. The SPEC2000 [26] *Alpha* binaries were executed in *sim-outorder* on the first reference dataset for the first billion instructions (corresponding to a single program phase [1]). In addition, single-precision versions of the STREAM and STREAM2 benchmarks [16] with a one million-loop limit were compiled on an *Alpha* machine. We use the default SimpleScalar configuration in Table 4, also used in [19]. SimpleScalar does not model an L3, so the memory latency used here is an estimate of L3 latency.

A code generator was built into HLS, and C-code was produced using the synthesis methods of Section 3. The synthetic testcases were compiled on an *Alpha* machine using *gcc* with optimization level *–O2* and executed to completion in SimpleScalar.

4.2 SPEC2000 Synthesis Results

The following figures show results for both the original applications, *actual*, and the synthetic testcases, *synthetic*. Figure 3 shows the IPC for the benchmarks. The average error for the synthetic benchmarks is 2.4%, with a maximum error of 8.0% for *facerec*. We discuss the reasons for the errors in the context of the figures below.

Figure 4 compares the average instruction percentages over all benchmarks for each class of instructions. The average prediction error for the synthetic testcases is 3.4% with a maximum of 7.3% for branches. Figure 5 shows that the basic block size varies per benchmark with an average error of 7.2% and a maximum of 21% for *mgrid*. The errors are caused by variations in the fractions of specific basic block types in the synthetic benchmark with respect to the original workload, which is a direct consequence of selecting a limited number of basic blocks during synthesis. For example, *mgrid* is synthesized with a total of 30 basic blocks made up of only six different unique block types. *Applu* is synthesized with 19 basic blocks but 18 unique block types. The basic block frequencies in the synthetic *mgrid* differ by 33.6% on average (only 8.4% for the top 40% of basic blocks) versus the basic block frequencies of the original workload. This is in contrast to testcases with large numbers of basic blocks such as *gzip*, which differ by only 2.2%.

The I-cache miss rates are shown in Figure 6. They show an error of 8.6% for benchmarks with IMRs above 1%, with a maximum of 22.9% for *sixtrack*. The number of synthetic instructions, however, is within 2.8% of the expected number given the I-cache configuration. The errors are again due to the process of choosing a small number of basic blocks with specific block sizes to synthesize the workload. For miss rates close to zero, a number of instructions less than 4096 is used, up to the number needed to give an appropriate instruction mix for the testcase. For the STREAM loops, only one basic block is needed to meet both the *IMR* and the instruction mix requirements. For the synthetic testcases, there appears to be a small but non-zero IMR, versus an essentially zero miss rate for

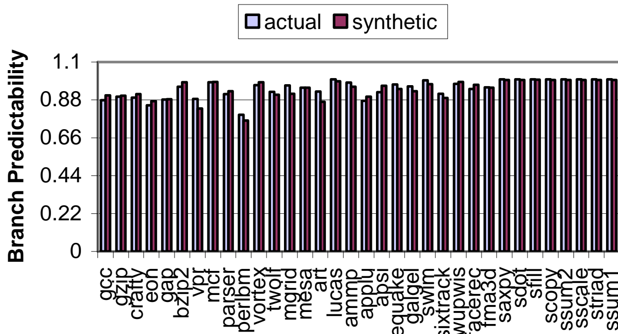

Figure 7. Branch Predictability

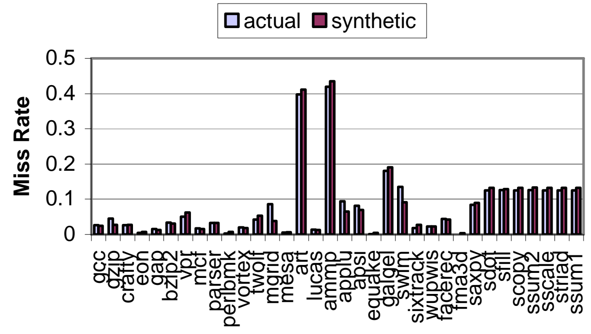

Figure 8. Actual vs. Synthetic DL1 Miss Rate

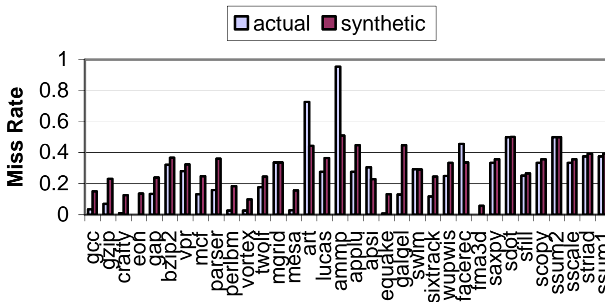

Figure 9. Actual vs. Synthetic UL2 Miss Rate

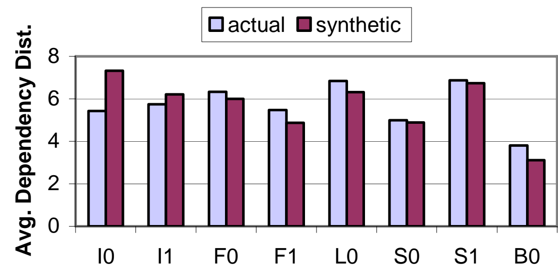

Figure 10. Actual vs. Synthetic Dependency Distances

some of the applications. This is because the synthetic testcases are only executed for about 300K instructions, far fewer than necessary to achieve a very small I-cache miss rate. However, since the miss rates are small, their impact on IPC when coupled with the miss penalty is also small.

The average branch predictability error is 1.9%, shown in Figure 7. The largest error is *art* at 6.4%, and *mgrid* has the third largest error at 4.9%. The L1 data cache miss rates are shown in Figure 8. For miss rates greater than 1%, the error is 12.3%. For these miss rates, the trends using the synthetic testcases clearly correspond with those of the original workloads. Again, there is more variation for smaller miss rates, but again the execution impact is also small.

In Figure 9, the unified L2 miss rates are shown. The large errors due to the simple streaming memory access model are often mitigated by small L1 miss rates. A good example is *gcc*, which has only a 2.6% L1 miss rate, and even the small L2 miss rate will not impact IPC significantly. Even though *art* and *ammp* have large L1 miss rates, the smaller L2 miss rates are offset by relatively larger I-cache miss rates and smaller branch predictabilities. The main cause of these errors is the fact that the current memory access model focuses on matching the L1 hit rate, and the L2 hit rate is simply predetermined as a consequence. The large error for *ammp* is partially explained by the fact that our small data-footprint synthetic testcases have data-TLB miss rates near zero, while the actual *ammp* benchmark has a data-TLB miss rate closer to 13%. As a consequence, the synthetic version does not correlate well when the dispatch window is increased and tends to be optimistic.

Figure 10 shows the average dependency distances, with 11.1% error on average. The largest components of error are the integer dependencies, caused by the conversion of many integer instructions to LSCNTRs, the memory access stride counters. A stride counter overrides the original function of the integer instruction and causes dependency relationships to change. Another source of error is the movement of dependencies during the search for compatible dependencies in the synthesis process. The movement is usually less than one position (Table 3), but *mgrid* and *applu*, the benchmarks with the largest average block sizes at 100.07 and 93.42, respectively, show significant movement. The branching model also contributes errors.

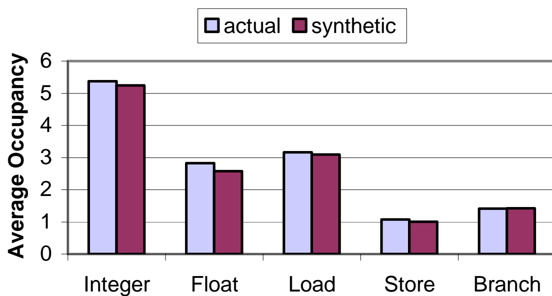

Figure 11. Actual vs. Synthetic Dispatch Window Occupancies

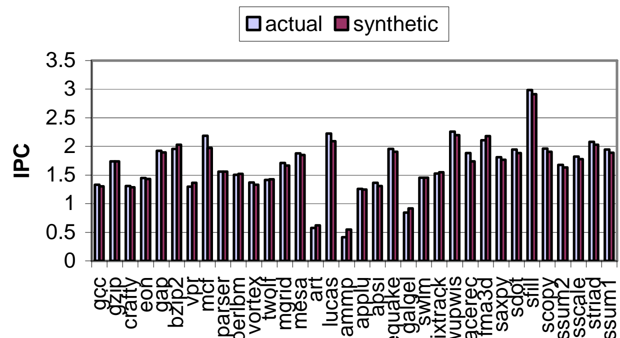

Figure 12. IPC Dispatch Window 32

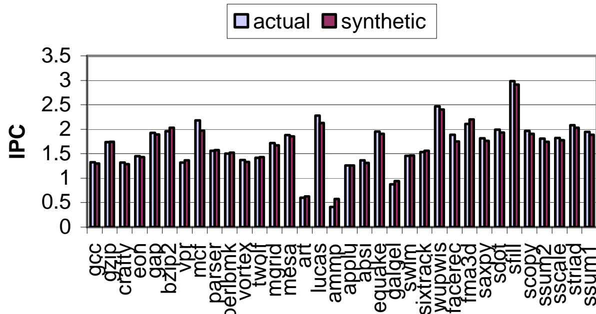

Figure 13. IPC Dispatch Window 64

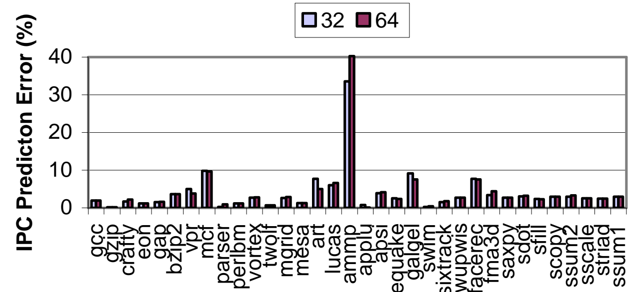

Figure 14. IPC Prediction Error for Dispatch Windows of 32 and 64

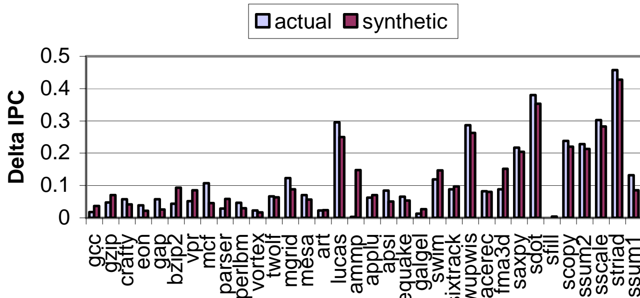

Figure 15. Delta IPC as Dispatch Window Increases from 16 to 32

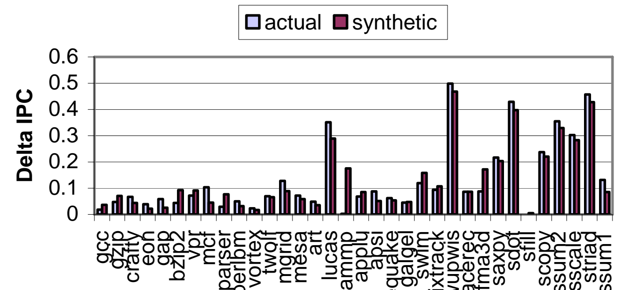

Figure 16. Delta IPC as Dispatch Window Increases from 16 to 64

Despite the dependency distance errors, Figure 11 shows that the average dispatch window occupancies are similar to those of the original benchmarks with an average error of 4.1%.

4.3 Assessing Design Changes

We now study design changes using the same synthetic testcases; that is, we take the testcases described in the last section, change the machine parameters in SimpleScalar and re-execute them. We test the accuracy of the synthetics by studying design changes in which the change in IPC is significantly greater than the average error between the synthetic and actual workload. For example, the dispatch window studies change IPC by more than 17% in each case.

Figures 12 and 13 show the absolute IPCs using a dispatch window of 32 and 64 with average errors of 3.0% and 3.1%, respectively. These numbers do not include *ammp*; as explained in the last section, *ammp* tends to be optimistic when the dispatch window changes because our small data footprint testcases do not model data-TLB misses. Figure 14 graphs the IPC prediction errors [12] for the dispatch windows. Most errors, except for *ammp,* are below 5%.

Figures 15 and 16 show the absolute change in IPC, *delta IPC*, as the same benchmarks and testcases are executed first with the default configuration (dispatch window of 16) and then with the dispatch window sizes changed to 32 and 64 respectively. The average relative errors [12] are 1.3% and 1.5%, respectively. The graphs show that, when an application change is large with respect to the changes in the other applications, the synthetic testcase change is also large relative to the change in the other synthetic testcases. These IPC changes would be large enough to trigger additional studies using a detailed cycle-accurate simulator, including an analysis of *ammp*. Chip designers are looking for cases in a large design space in which a design change may improve or worsen a design. In the case of the dispatch window studies, the results

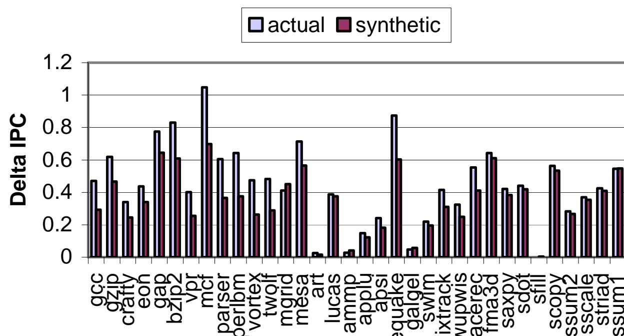

Figure 17. Delta IPC as L1 Data Latency Increases from 1 to 8

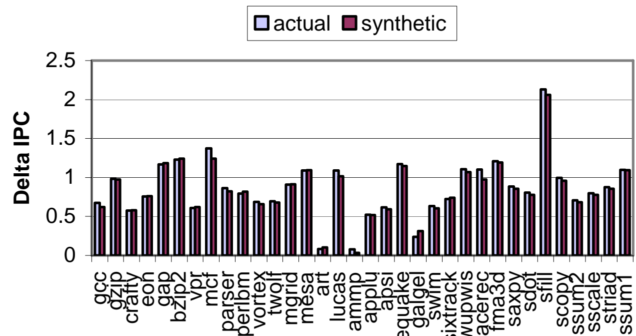

Figure 18. Delta IPC as Issue Width Increases from 1 to 4

Table 5. Average Synthetic IPC Errors and Relative IPC Errors vs. Actual Applications

Model	Commit Width 8		Commit Width 1		L1 D-cache (256:64:8)		L1 I-cache (1024:64:2)	
	IPC	Rel. IPC	IPC	Rel. IPC	IPC	Rel. IPC	IPC	Rel. IPC
SPEC95 (Perfect Branching)	3.7%	1.1%	2.8%	4.2%	4.3%	2.1%	9.8%	8.2%
SPEC95 (Branching Model)	2.6%	1.4%	3.2%	3.9%	3.2%	2.4%	8.7%	7.5%
SPEC2000 (Branching Model)	2.4%	0.2%	2.8%	2.7%	3.1%	1.0%	3.0%	1.3%

would trigger further cycle-accurate studies of *lucas, ammp, swim,wupwise, fma3d* and the STREAM benchmarks. Alternatively, the designers might be curious why the change did not help the SPECint testcases.

Figure 17 shows the delta IPC as the L1 D-cache latency is increased from 1 to 8. The average absolute IPC error is 9.5% and the relative error is 9.7%. The errors are high, but the larger changes in the actual benchmarks over 1B instructions are reflected in the synthetic testcases that run in seconds. However, it is apparent that the memory access model is less accurate for SPECint than for the other testcases. In fact, the average relative error for SPECint is 19.9% versus 4.2% for the others.

Figure 18 shows better results for the delta IPC as the issue width increases from 1 to 4. The average absolute error is 1.9%, and the relative error is 2.4%. Similar results for commit width changes, doubling the L1 D-cache (to 256 sets, 64B cache line, 8-way set associative), and doubling the L1 I-cache configuration (to 1024 sets, 64B cache line, 2-way set associativity) are shown in Tables 5 and 6. We also reproduce results from technical report [2] for the SPEC95 testcases synthesized the same way but targeting the *Pisa* ISA.

In an additional study, a code generation target for the PowerPC ISA was implemented, and an instruction trace for TPC-C (described in [4]) was fed through the synthesis code. The resulting testcase was then compiled and executed on a detailed performance simulator for the IBM Power4 processor [28] and compared to results for the original 62M instructions. A 6.4% IPC error was obtained, demonstrating the flexibility and retargetability of the synthesis approach.

5. RELATED SYNTHESIS WORK

Several ad-hoc techniques to synthesize workloads have been developed [32][25][31]. In [32], a linear combination of microbenchmarks is found that, when combined in a process called *replication* and executed, duplicates the LRU hit function of the target benchmark. There is no clear way to incorporate other execution characteristics like instruction mix into the technique.

In [14], assembly programs are generated that have the same power consumption signature as applications. However, all workload characteristics are modeled as microarchitecture-dependent characteristics, so the work is not useful for studies involving design trade-offs [13]. In particular, the instruction sequences and dependency relationships of the synthetic programs are not representative of the original workload, unlike in the present work. The cache access and branch predictor models in [14] are useful as high-level ideas or starting points, but the specific implementations in that work allow and rely on modifications to the workload features shown to be required for representative performance.

Sakamoto et al. [21] present a method to create a binary image of a trace and memory dump and execute those on a specific machine and a logic simulator, but the required binary image and fixup code are complicated and not easily portable to other systems and simulators. No attempt is made to create an abstract trace from statistics in order to reduce runtimes.

6. CONCLUSIONS AND FUTURE WORK

We propose a method for synthesizing representative testcases from the workload characteristics of an executing application. The target application's executable is analyzed in detail and representative sequences of instructions are instantiated as in-line assembly-language instructions inside synthetic C-code.

Unlike prior synthesis efforts, we focus on the low-level workload characteristics of the compiled and executing binary to create workloads that are representative of the effects of the application in the machine. Multiple synthetic testcases are necessary if the application is executed on multiple machines, significantly different ISAs, or multiple datasets, but the automatic process minimizes the cost of creating new testcases and enables consolidation of multiple representative phases into a single small testcase. Other benefits include portability, future workload generation, and code abstraction. Future work includes more accurate memory access and branching models.

We use the method to synthesize representative testcases for the SPEC2000 and STREAM *Alpha* benchmarks and find that testcases can be synthesized to an average IPC within 2.4% of the average IPC of the target applications with similar average instruction mix, cache access characteristics, dispatch window occupancies, and dependency characteristics, while runtimes are often three orders of magnitude shorter, making functional simulation and hardware simulation for performance validation feasible. In addition, the changes in IPC for a synthetic testcase due to design changes are found to be proportional to the corresponding IPC changes for the original application.

7. ACKNOWLEDGMENTS

The authors would like to thank Lieven Eeckhout, Koen De Bosschere, and the anonymous reviewers for their detailed comments. This research is supported by the National Science Foundation under grant number 0429806, the IBM Center for Advanced Studies (CAS), and an IBM SUR grant.

Table 6. Average Synthetic IPC Errors and Relative IPC Errors vs. Actual Applications

Model	Dispatch Window					DL1 Latency 8		Issue Width 1	
	16	32		64					
	IPC	IPC	Rel. IPC	IPC	Rel. IPC	IPC	Rel. IPC	IPC	Rel. IPC
SPEC95 (Perfect Branching)	3.9%	3.2%	2.1%	3.1%	2.7%	8.9%	6.9%	2.6%	4.1%
SPEC95 (Branching Model)	2.4%	3.1%	2.2%	3.3%	2.4%	11.1%	10.4%	2.1%	2.2%
SPEC2000 (Branching Model)	2.4%	3.1%	1.3%	3.0%	1.5%	9.5%	9.7%	1.9%	2.3%

8. REFERENCES

[1] R. H. Bell, Jr., L. Eeckhout, L. K. John and K. De Bosschere, “Deconstructing and Improving Statistical Simulation in HLS,” Workshop on Debunking, Duplicating, and Deconstructing, June 20, 2004.

[2] R. H. Bell, Jr. and L. K. John, “Experiments in Automatic Benchmark Synthesis,” Technical Report TR-040817-01, Laboratory for Computer Architecture, University of Texas at Austin, August 17, 2004.

[3] B. Black and J. P. Shen, “Calibration of Microprocessor Performance Models,” IEEE Computer, May 1998, pp. 59-65.

[4] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla and S. R. Kunkel, “A Multithreaded PowerPC Processor for Commercial Servers,” IBM J. of Res. Dev., Vol. 44 No. 6, November 2000.

[5] P. Bose and T. M. Conte, “Performance Analysis and Its Impact on Design,” IEEE Computer, May 1998, pp. 41-49.

[6] P. Bose, “Architectural Timing Verification and Test for Super-Scalar Processors,” IEEE International Symposium on Fault-Tolerant Computing, June 1994, pp. 256-265.

[7] D. C. Burger and T. M. Austin, "The SimpleScalar Toolset," Computer Architecture News, 1997.

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

[9] T. Conte and W. Hwu, “Benchmark Characterization for Experimental System Evaluation,” Hawaii International Conference on System Science, 1990, pp. 6-18.

[10] H. J. Curnow and B.A. Wichman, "A Synthetic Benchmark," Computer Journal, Vol. 19, No. 1, February 1976, pp. 43-49.

[11] R. Desikan, D. Burger and S. Keckler, “Measuring Experimental Error in Microprocessor Simulation,” IEEE International Symposium on Computer Architecture, 2001.

[12] L. Eeckhout, R. H. Bell, Jr., B. Stougie, L. K. John and K. De Bosschere, “Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies,” IEEE International Symposium on Computer Architecture, June 2004.

[13] L. Eeckhout, Accurate Statistical Workload Modeling, Ph.D. Thesis, Universiteit Gent, 2003.

[14] C. T. Hsieh and M. Pedram, "Microprocessor power estimation using profile-driven program synthesis," IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 17, No. 11, Nov. 1998, pp. 1080-1089.

[15] T. Lafage and A. Seznec, “Choosing Representative Slices of Program Execution for Microarchitecture Simulations,” IEEE Workshop on Workload Characterization, 2000.

[16] J. D. McCalpin, “Memory bandwidth and machine balance in current high performance computers,” IEEE Technical Committee on Computer Architecture newsletter, Dec. 1995.

[17] L. McVoy, “lmbench: Portable Tools for Performance Analysis,” USENIX Technical Conference, Jan. 22-26, 1996, pp. 279-294.

[18] M. Moudgill, J. D. Wellman and J. H. Moreno, “Environment for PowerPC Microarchitecture Exploration,” IEEE Micro, May-June 1999, pp. 15-25.

[19] M. Oskin, F. T. Chong and M. Farrens, "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Design," IEEE International Symposium on Computer Architecture, June 2000, pp. 71-82.

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

[21] M. Sakamoto, L. Brisson, A. Katsuno, A. Inoue and Y. Kimura, “Reverse Tracer: A Software Tool for Generating Realistic Performance Test Programs,” IEEE Symposium on High-Performance Computing,” 2002.

[22] T. Sherwood, E. Perleman, H. Hamerly and B. Calder, “Automatically characterizing large scale program behavior,” IEEE Conference on Architected Support for Programming Languages and Operating Systems, October 2002.

[23] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja and V. S. Pai, “Challenges in Computer Architecture Evaluation,” IEEE Computer, August 2003, pp. 30-36.

[24] E. S. Sorenson and J. K. Flanagan, “Evaluating Synthetic Trace Models Using Locality Surfaces,” IEEE Workshop on Workload Characterization,” Nov. 2002, pp. 23-33.

[25] K. Sreenivasan and A.J. Kleinman, “On the Construction of a Representative Synthetic Workload,” Communications of the ACM, March 1974, pp.127-133.

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

[27] S. Surya, P. Bose and J. A. Abraham, “Architectural Performance Verification: PowerPC Processors,” Proceedings of the IEEE International Conference on Computer Design, 1999, pp. 344-347.

[28] J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le and B. Sinharoy, "POWER4 System Microarchitecture," IBM J. of Res. and Dev., January 2002, pp. 5-25.

[29] D. Thiebaut, “On the Fractal Dimension of Computer Programs and its Application to the Prediction of the Cache Miss Ratio,” IEEE Transactions on Computers, Vol. 38, No. 7, July 1989, pp. 1012-1026.

[30] R. P. Weiker, "Dhrystone: A Synthetic Systems Programming Benchmark," Communications of the ACM, October 1984, pp. 1013-1030.

[31] J. N. Williams, “The Construction and Use of a General Purpose Synthetic Program for an Interactive Benchmark for on Demand Paged Systems,” Communications of the ACM, 1976, pp.459-465.

[32] W. S. Wong and R. J. T. Morris, "Benchmark Synthesis Using the LRU Cache Hit Function," IEEE Transactions on Computers, Vol. 37, No. 6, June 1988, pp. 637-645.

[33] R. E. Wunderlich, T. F. Wenisch, B. Falsafi and J. C. Hoe, “SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling,” IEEE International Symposium on Computer Architecture, June 2002.