Automatic Testcase Synthesis and Performance Model Validation for High-Performance PowerPC Processors

Robert H. Bell, Jr. [†‡] Rajiv R. Bhatia[†‡] Lizy K. John[‡] Jeff Stuecheli[†‡]
John Griswell[†] Paul Tu[†] Louis Capps[†] Anton Blanchard[†] Ravel Thai[‡]
[†] *IBM Systems and Technology Division*
Austin, Texas
robbell@us.ibm.com
[‡] *Electrical and Computer Engineering*
The University of Texas at Austin
ljohn@ece.utexas.edu

Abstract

The latest high-performance IBM PowerPC microprocessor, the POWER5 chip, poses challenges for performance model validation. The current state-of-the-art is to use simple hand-coded bandwidth and latency testcases, but these are not comprehensive for processors as complex as the POWER5 chip. Applications and benchmark suites such as SPEC CPU are difficult to set up or take too long to execute on functional models or even on detailed performance models.

We present an automatic testcase synthesis methodology to address these concerns. By basing testcase synthesis on the workload characteristics of an application, source code is created that largely represents the performance of the application, but which executes in a fraction of the runtime. We synthesize representative PowerPC versions of the SPEC2000, STREAM, TPC-C and Java benchmarks, compile and execute them, and obtain an average IPC within 2.4% of the average IPC of the original benchmarks and with many similar average workload characteristics. The synthetic testcases often execute two orders of magnitude faster than the original applications, typically in less than 300K instructions, making performance model validation for today's complex processors feasible.

1. Introduction

Modern high-performance microprocessors are quite complex. For example, the POWER4™ and POWER5™ chips are dual-core PowerPC® microprocessors used in IBM server systems [28][25]. The POWER4 chip is built from 1.5 million lines of VHDL and 174 million transistors [16], and the POWER5 chip contains 276 million transistors [25]. The designs drive detail and complexity into the performance models used for performance projections.

For complex chips, it is important that the performance models be validated [7] against cycle-accurate functional models during the design process in order to minimize incorrect design decisions due to inaccurate performance models. As complexity increases, the gap in accuracy can grow quickly, so validation is needed more frequently. Subtle instruction interactions in the POWER4 and POWER5 chips necessitate very accurate performance models.

Prior validation efforts have focused on bandwidth and latency tests, resource limit tests, or micro-tests [8] [27][5][19][18][17][13]. These are usually hand-written microbenchmarks that are too small to approximate the performance of many workloads [4]. Once validated using these tests, performance models still exhibit large errors on realistic workloads like SPEC [3][4]. Randomly generated tests are inefficient at representing real workloads [4].

Applications themselves cannot be used for performance model validation because of their impossibly long simulation runtimes [7]. In [24], only one billion functional model simulated cycles *per month* are obtained. In [16], farms of machines provide many simulated cycles in parallel, but individual tests on processor models may execute orders of magnitude slower than hardware emulator speeds of 2500 cycles per second.

Trace sampling techniques such as SimPoint [23] and SMARTS [30] can reduce runtimes in simulators, but the executions still amount to tens of millions of instructions. Statistical simulation [9][20][10] can further reduce the necessary trace lengths, 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 [22], but there is no attempt to reduce trace lengths. Hsieh and Pedram synthesize instructions for power estimation [11], but there is no attempt to maintain the machine-independent workload characteristics necessary to represent the original applications [1][10].

Bell and John [2][5][4] synthesize C-code programs from reduced synthetic traces generated from the workload characteristics of executing applications, as in statistical simulation. The low-level workload characteristics of the original application are retained by instantiating individual operations as volatile *asm* calls. The synthetic testcases execute orders of magnitude faster than the original workloads while retaining good performance accuracy.

In this work, the synthesis effort is broadened to support high-performance PowerPC processors such as the POWER5 chip. In addition, two performance model validation approaches and validation results are presented using testcases for a PowerPC processor.

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 synthesis results for the POWER5 processor and Section 5 presents validation results for a follow-on PowerPC processor.

2. Representing Complexity

Representative testcase synthesis [3][4] is achieved using the workload characterization and statistical flow graph of statistical simulation [10][1]. A walk of the statistical flow graph produces a synthetic trace which, when combined with memory access and branching models, is instantiated as a C-code envelope around a sequence of asm calls - a simple but flexible testcase.

When the synthetic is executed, the proportions of the instruction sequences are similar to the proportions of the same sequences in the original executing application, but the number of times the sequence is executed is significantly reduced, so that the total number of instructions executed and, therefore, overall runtime are much reduced. By repeating the execution of the sequences a small number of times, convergence of memory access and branching model behavior is assured, usually in less than 300K instructions [3][4].

The relatively small number of instructions and the flexibility of the source code make the synthetics useful for accurate performance model validation [3][4]. In the case of the POWER5 chip, executions of a cycle-accurate model built directly from the functional VHDL hardware description language model [29][16] can be compared against the detailed M1 performance model [14][13] used for performance projections. In this work, we synthesize specifically for POWER5 chip execution and then use the testcases to validate an improved PowerPC processor.

2.1 Performance Model

The IBM PowerPC performance modeling environment is trace-driven to reduce modeling and simulation overhead [19][15][13]. The M1 performance model implements a detailed, cycle-accurate core. Coupled with the M1 is a detailed model of the L2, L3 and memory [13].

Elements of a high-performance PowerPC model must capture the functional details of the processor. The POWER5 chip features out-of-order instruction execution, two fixed point and two floating point units, 120 general purpose rename registers, 120 floating point rename registers, complex pipeline management, 32-entry load and store reorder queues, on-board L2 controller and 1.9-MB cache, L3 cache controller, 36-MB off-chip victim L3, memory controller, SMP fabric bus interface, dynamic power management, and support for simultaneous multithreading [25]. Table 6 gives additional configuration parameters.

2.2 Model Validation

The POWER5 performance model was validated to within 1% for particular codes [13]. However, it is a difficult and time consuming proposition to validate a model for a large variety of programs. Automatic testcase synthesis addresses this concern by automating the synthesis process and by synthesizing high level codes that can target various platforms. In this work we target two platforms for validating the performance model: cycle-accurate RTL model simulation and simulation on a hardware emulator.

2.2.1 Validation using RTL Simulation. Figure 1 shows the RTL validation methodology. The synthetic testcase is compiled and converted to execute on a VHDL model using the standard IBM functional verification simulation methodology [29][16]. The converted code is then executed to completion on the VHDL model. The compiled testcase is also unrolled into a trace using a PowerPC instruction interpreter [19][15][6]. Only completed instructions are maintained in the trace. The trace is then executed to completion on the M1 performance model.

Both VHDL and performance model executions generate information at the level of the instruction,

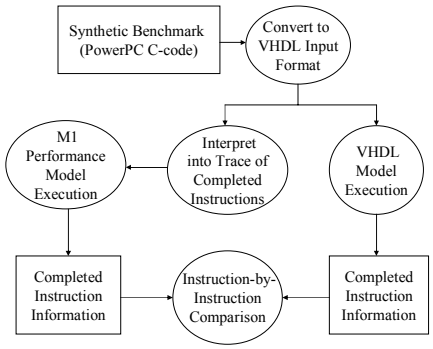

Figure 1. RTL Validation Methodology

including the cycle the instruction completes, address and opcode. The cycles at which an instruction completes in both the VHDL and performance models are not identical in an absolute sense because of how cycles are maintained and counted in the two simulators. However, the performance model is of sufficient detail such that the completion cycle of an instruction minus the completion cycle of the previous instruction, that is, the cycle difference between any two completing instructions, should be equivalent. The analysis relies on the fact that the methodology generates instructions for both models from a compilation of a single piece of code. Each and every instruction is executed on both the VHDL and M1 models and completes in the same order in both. Instructions may complete in different completion buffers in the same cycle, but should complete in the same cycle.

The instantaneous error, E, for each instruction is defined in terms of the difference in cycles between the completion time of the current instruction and the completion time of the previous instruction in both VHDL and M1 models. The instantaneous error for the i^{th} instruction is defined as:

$$E(i) = D_R(i) - D_P(i)$$

where

$$D_R(i) = C_R(i) - C_R(i-1)$$

and likewise

$$D_P(i) = C_P(i) - C_P(i-1)$$

In these equations, $C_R(i)$ is the completion cycle of the i^{th} instruction when executing the VHDL (RTL) model, and $C_P(i)$ is the completion cycle of the i^{th} instruction when executing the M1 (Performance) model. The intuition behind the E calculation is that an instruction is using the same machine resources and completing in the same order, putatively in the same cycle, in both models, so differences between sequential instructions should be identical. A difference indicates that resource allocations or execution delays are not modeled similarly for instructions in the vicinity of the instruction that has an instantaneous error.

Note that instruction dependences and resource usages using the same workload in both models will limit the instantaneous error and push it toward zero; there can be no large accumulation of error between any two related instructions. However, the completion time of an instruction that is modeled properly may be underestimated if an older instruction in program order fails to complete on time and the younger instruction is not dependent on the older but on a prior instruction and has been ready to complete for some time.

The instantaneous errors can be categorized to narrow down the search for microarchitectural bugs in which a hardware feature was not implemented properly, and modeling, abstraction, and specification errors [5] in the performance model. Section 5 gives an example analysis.

2.2.3 Validation using a Hardware Emulator. The compiled synthetic testcases can also be input to an RTL model executing on the AWAN hardware accelerator [16]. AWAN is composed of programmable gate arrays and can be configured to emulate a VHDL model. The AWAN array supports very large chips. The entire POWER4 chip, for example, can be emulated at a rate of more than 2500 cycles per second [16]. The cycle counts for a run are obtained from AWAN registers and compared to the M1 performance model cycle counts. Detailed workload execution information can be obtained from the AWAN machine. In Section 5, validation results using AWAN are presented.

3. Synthesis for PowerPC

With reference to Figure 2, the four major phases of synthesis: *workload characterization; graph analysis; register assignment* and *code generation*. The following paragraphs present only the changes to synthesis to support the PowerPC instruction set and high-performance processors. Additional synthesis detail for each phase, as well as exact synthesis parameters and algorithms for *Pisa* and *Alpha* code targets, can be found in Bell and John [2][3][4].

3.1 Workload Characterization

The workload characterization is carried out at the granularity of the basic block [2][3][4]. The floating-point instruction abstraction is augmented to support extremely long PowerPC *fma* instructions.

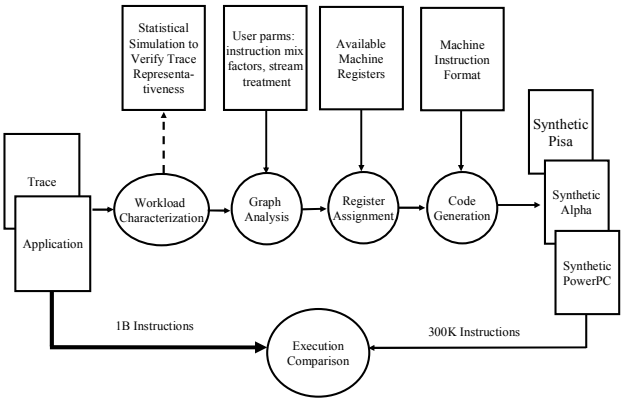

Figure 2: Synthesis and Simulation Overview

3.2 Graph Analysis

The statistical flow graph is analyzed and specific memory access and branching models are constructed.

3.2.1 Instruction Miss Rate and I-cache Model. The number of basic blocks are tuned to match the original I-cache miss rate (IMR) [3]. Specific basic blocks are chosen from a walk of the statistical flow graph [3]. The IMR sometimes changes after the branches are configured in the branching model (Section 3.2.5). To compensate, the *branch jump granularity* is adjusted to change the number of basic blocks or instructions that a configured branch jumps.

3.2.2 Instruction Dependences and Compatibility. For each instruction, its dependences are assumed to be the same as those chosen during statistical simulation. The dependences are then made compatible with the instruction by moving forward and backward from the starting dependence through the list of instructions in sequence order [2][3][4]. Table 1 shows the compatibility of instructions for the PowerPC instruction set. The *Inputs* column gives the assembly instruction inputs that are being tested for compatibility.

In our PowerPC experiments, some benchmarks had high average numbers of moves away from the starting dependence. To compensate, after 25 moves, a compatible instruction is inserted into the basic block near the starting dependence. The total number of inserts for each benchmark is shown in the *dependence inserts* column of Table 4. The highest numbers are associated with *mgrid* and *applu*, which have the largest average basic blocks sizes, at 125 and 115 respectively. Using the inserts, the average number of moves per instruction input, shown in column *dependence moves*, is generally small.

Table 1. PowerPC Dependence Compatibility Chart

Dependent Instruction	Inputs	Dependence Compatibility	Comment
Integer	0/1	Integer, Load-Integer	
Float	0/1/2	Float, Load-Float	3 Inputs for *fma*
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	0/1	Integer, Load-Integer	Condition Registers

In some cases, the *dependence factor* in Table 5 is used to multiply the synthetic dependences to more closely match the overall averages in the original application. The dependence adjustment for *mgrid* is necessary due to its large average basic block size and, therefore, small number of synthetic basic blocks.

3.2.3 Loop Counters and Program Termination. The number of executed loops, *loop iterations,* is shown in Table 4. An *mtspr* instruction initializes the internal count register to the loop iterations, and the final branch checks for zero in the count register and decrements it.

3.2.4 Memory Access Model. The stream access classes and associated strides [2][3] for the POWER5 cache configurations are shown in Table 2. The first two rows are useful only for stores in store-through POWER machines. Stores in the L2 gather such that a simple walk through memory results in a 50% L2 hit rate. If the L1 hit rate is below 3.17%, the L2 hit rate is matched. The (non-zero) L1 Hit Rate is based on the line size of 128 bytes in the POWER5 chip:

$$L1_{HitRate} = 1 - (Stride / 128) \cdot 4$$

where the stride is given in increments of 4 bytes.

There can be a large error in stream behavior for two reasons. An actual L1 hit rate may fall between the hit rates in two rows [2][3], but for the PowerPC configuration this maximizes to only about 3% error. A larger error is associated with the lack of distinguishing L2 hit rate quanta. Since the L1 and L2 line sizes are the same in our PowerPC machine, it is difficult to get positive L2 hit rates with simple stride models.

Consequently, we implement walks through particular cache congruence classes. We call them *bounded* streams to differentiate them from streams that continually increment through memory. The implementation makes use of the default 4-way set associative L1 and 10-way set associative L2 in the machines under study. The difference in associativity means that walks through a class will hit in the L2 but miss in the L1 to the extent that the entire class is walked. If the L2 hit rate is greater than the L1 hit rate

multiplied by the *bounded factor* in Table 5, then the stream in the basic block is changed from a simple stride stream (*stream pools* in Table 4) to a congruence class walk (*bounded stream pools* in Table 4). To achieve the particular L1 and L2 hit rates in a row of Table 3, the *instruction reset* column gives the total number of 8K accesses that are necessary before repeating the same access sequence in the congruence class.

Note that, for studies of cache size design changes, congruence class walks essentially clamp the hit rates to a particular level, since the rates will not change unless the associativity changes. The ultimate effect of the use of this factor is to adjust the ratio of the L1 and L2 hit rates to more closely match that of the original application.

Because of accumulated errors in stream selection, manipulation of the streams [3][4] was extended. In Table 5, the *stream factor* multiplies the moving average of the L1 hit rate taken from the table during each lookup, and if the result is greater than the original hit rate by *(N·10%)*, the selected stream is chosen from the preceding $(N+1)^{st}$ row. This has the effect of reducing overall hit rates for the first load or store fed by an LSCNTR. Similarly for the bounded streams, the *bounded stream factor* in Table 5 multiplies the L1 hit rate.

The *load-store-offset factor* changes the address offset of loads and stores to a value from one to 8K based on a uniform random variable. The factor value usually has a proportional effect on cache miss rates and IPC because of the random access but fewer load-stores address collisions. The *load-hit-store factor* changes the number of stores that have the same word address offset as loads. The factor value has an inversely proportional effect on IPC. We also implemented a simple way to increase both L1 and L2 misses by configuring a fraction of non-bounded streams to stride by a fraction of a 4KB page. *Mcf* configures three streams to walk a page, and *art* and *java* configure one stream to walk 0.8 and 0.6 of a page, respectively. Future work will investigate generating these factors directly from the workload characterization.

Table 2. L1 and L2 Hit Rates versus Stride

L1 Hit Rate	L2 Hit Rate	Stride
0.0000	1.00	0 (store-through)
0.0000	0.50	1 (store-through)
0.0000	0.00	32
0.0313	0.00	31
0.0625	0.00	30
0.0942	0.00	29
0.1250	0.00	28
0.1563	0.00	27
0.1875	0.00	26
0.2188	0.00	25
0.2500	0.00	24
0.2813	0.00	23
0.3125	0.00	22
0.3438	0.00	21
0.3750	0.00	20
0.4063	0.00	19
0.4380	0.00	18
0.4688	0.00	17
0.5000	0.00	16
0.5313	0.00	15
0.5625	0.00	14
0.5938	0.00	13
0.6250	0.00	12
0.6563	0.00	11
0.6875	0.00	10
0.7188	0.00	9
0.7500	0.00	8
0.7813	0.00	7
0.8125	0.00	6
0.8438	0.00	5
0.8750	0.00	4
0.9063	0.00	3
0.9375	0.00	2
0.9688	0.00	1

3.2.5 Branch Predictability Model. Unlike the *Pisa* and *Alpha* syntheses [2][3][4], the *branch jump granularity* was not needed for *mgrid* and *applu* because their PowerPC versions have very high branch predictabilities, but it was used to tune the branch predictability for several SPECint benchmarks such as *eon* and *twolf*, which have relatively low branch predictabilities. In those cases, the branch jumps past one instruction of the next basic block.

Likewise, the capability to skew the length of the basic block by choosing sized successors [3] was not needed for *mgrid* and *applu* because of their high branch predictabilities, but it was used to tune the block sizes of various benchmarks. In Table 5, as the

Table 3. L1 and L2 Hit Rates vs. Instruction Reset (Congruence Classes)

L1 Hit Rate	L2 Hit Rate	Instruction Reset
1.0000	1.0000	4
0.6000	1.0000	5
0.3333	1.0000	6
0.1429	1.0000	7
0.0000	1.0000	8
0.0000	0.8182	11
0.0000	0.6667	12
0.0000	0.5385	13
0.0000	0.4286	14
0.0000	0.3333	15
0.0000	0.2500	16
0.0000	0.1765	17
0.0000	0.1111	18
0.0000	0.0526	19
0.0000	0.0000	20

basic block size factor is reduced from unity, the block size is skewed toward the *basic block length* value.

3.3 Register Assignment

For the codes under study, the number of registers available for streams averages about 8 and for code use about 12 (*code registers* and *stream pools* + *bounded stream pools,* in Table 4). In the *Pisa* and *Alpha* studies, pools are greedily consolidated by iteratively combining the two least frequent pools until the limit is reached [2][3][4]. For the PowerPC codes, the top most frequent pools are never watered down with less frequent pools; the last pool under the limit consolidates all less frequent pools. In all cases, the consolidated pools use the pool stride or reset value that minimizes the hit rate. The *stream pools* and *bounded stream pools* are consolidated separately.

3.4 Code Generation

The code generator of Figure 2 emits a single module of C-code that contains calls to assembly-language instructions in the PowerPC language. The data access counters [2][3][4] are emitted as *addi* instructions that add their associated stride to the current register value. The loop counter is emitted as an *add* of minus one to its register. Short latency floating-point operations are generated using *fadd*., long latency floating-point operations using *fmul*, and extremely-long latency operations using *fmadd*. Loads use *lwz* or *lfs,* depending on the type, and similarly for stores. Branches use the *bc* with integer operands.

Tables 4 and 5 give the synthesis information for the PowerPC SPEC2000 and STREAM codes as described in this section and in [2][3]. The *runtime ratio* is the user runtime of the original benchmark for one hundred million instructions (1M for STREAM) divided by the user runtime of the synthetic testcase on various

Table 4. Synthetic Testcase Properties (PowerPC)

Name	Number of Basic Blks	Number of Instructions	Loop Iterations	Stream Pools	Bounded Stream Pools	Code Registers	Dependence Moves	Dependence Inserts	Runtime Ratio
gcc	750	2524	80	5	3	12	6.093	60	**437.58**
gzip	840	3683	119	4	4	12	0.465	1	**481.1**
crafty	360	3699	56	5	3	12	0.797	4	**718.11**
eon	330	3879	41	3	5	12	3.113	40	**1181.51**
gap	510	3940	65	4	4	12	0.33	0	**954**
bzip2	300	1859	144	5	5	10	0.418	0	**562.62**
vpr	400	2855	121	7	3	10	0.648	13	**1051.48**
mcf	800	3561	71	7	3	10	0.649	0	**495.19**
parser	795	4013	54	8	2	10	0.833	0	**1113.85**
perlbmk	600	3834	55	9	1	10	1.95	0	**998.11**
vortex	500	2417	90	2	8	10	0.889	0	**994.29**
twolf	540	3952	71	3	7	10	0.596	1	**1190.29**
mgrid	30	4008	65	8	2	10	1.632	255	**1050.13**
mesa	400	3362	81	5	3	12	1.32	23	**1123**
art	200	4213	46	6	2	12	1.4	228	**902.18**
lucas	80	2367	141	3	5	12	1.915	0	**872.08**
ammp	200	1700	160	4	4	12	6.608	0	**749.18**
applu	30	3851	63	6	2	12	1.204	272	**378.77**
apsi	200	3208	70	8	0	12	4.585	0	**345.54**
equake	50	2459	71	7	1	12	9.499	0	**700.57**
galgel	120	3868	53	6	2	12	11.225	0	**583.31**
swim	70	3468	71	4	4	12	1.769	85	**1079.23**
sixtrack	150	2624	144	5	2	12	1.12	0	**494.55**
wupwise	200	2756	69	5	3	12	11.095	0	**1258.9**
facerec	200	2530	113	5	3	12	3.982	0	**616.75**
fma3d	150	3596	49	6	2	12	5.594	0	**445.68**
saxpy	1	8	33334	2	0	12	0	0	**28.27**
sdot	1	6	50001	2	0	12	0.125	0	**71.85**
sfill	1	3	100001	1	0	12	6.25	0	**22.47**
scopy	1	6	50001	2	0	12	0	0	**61.75**
ssum2	1	4	100001	1	0	12	0.2	0	**18.67**
sscale	1	7	50001	2	0	12	0	0	**23.23**
striad	1	9	33334	3	0	12	0	0	**27.16**
ssum1	1	9	33334	3	0	12	0	0	**26.97**
tpc-c	4500	23102	12	2	8	10	0.571	0	**447.77**
java	4750	23391	14	3	7	10	0.416	0	**447.59**

POWER3 and POWER4 workstations. Each pass through the synthesis process takes less than five minutes on an IBM p270 (400 MHz). The results show a two or three order of magnitude speedup using the synthetics.

In practice, there are multiple synthetic benchmarks that more or less satisfy the metrics obtained from the workload characterization and overall application performance. Synthesis is carried out a number of times [2][3][4] until the metric deltas versus the original application are relatively small. Among the sets of parameters that are obtained, the one that most closely reproduces the original performance is preferred. Usually multiple synthesis passes plus think time are necessary to tune the synthesis parameters for each testcase.

4. Synthesis Results

In this section, we present the results of simulation experiments for the synthetic POWER5 testcases obtained in the last section. In Section 5, we use these testcases to validate the performance model of an improved processor.

4.1 Experimental Setup and Benchmarks

We use a profiling system derived from the system used in [3], which evolved from HLS [20][21]. The POWER5 M1 performance model described in Section 2 was augmented with code to carry out the workload characterization as in [10][1][3]. The 100M instruction SPEC2000 [26] PowerPC traces used in [14][13] and described in [6] were executed on the augmented M1. We also add an internal DB2 instruction trace of TPC-C [6][12] and a 100M instruction trace for SPECjbb (*java*) [26]. In addition, single-precision versions of the STREAM and STREAM2 benchmarks [17] with a one million-loop limit were compiled on a PowerPC machine. We use the default POWER5 configuration in Table 6 [25].

A code generator for the PowerPC target was built into the synthesis system, and C-code was produced using the synthesis methods of Section 3. The synthetic

Table 5. Synthetic Testcase Memory Access and Branching Factors (PowerPC)

Name	Dependence Factor	Bounded Factor	Stream Factor	Bounded Stream Factor	Load-Hit-Store Factor	Load-Store Offset Factor	BP Factor	Basic Block Size Factor	Basic Block Length
gcc	1	1	1	1	0.58	0.24	1.01	0.95	4
gzip	1	1	1.2	1.2	1	1	0.65	1	-
crafty	1	1	0.9	0.9	0.15	0.97	1	0.8	10
eon	1	1	0.9	0.9	0.1	1	0.8	0.9	10
gap	1	1	1	1	0.28	0.98	1.03	1	-
bzip2	1	1	1	1	0.92	0.965	0.5	0.96	6
vpr	1	1.05	0.75	0.7	1	1	0.75	1	-
mcf	1	1	1	1	1	1	0.9	1	-
parser	1	1	1	1	1	1	1.02	0.9	5
perlbmk	1	1.05	1	1	0.85	0.998	1.05	0.95	5
vortex	1.5	0.01	1.1	0.1	0.01	1	1	0.9	5
twolf	1	1	1.05	1.1	1	1	0.8	1	-
mgrid	3.0	0.75	0.8	0.8	0.1	0.92	1	1	-
mesa	0.9	1	0.96	0.95	1	0.96	0.95	1	-
art	1	1.5	1.5	1.5	1	0.01	1	1	-
lucas	1	0.1	1	1	1	0.83	1	0.9	20
ammp	1	0.9	1.35	1	1	0.89	1	1	-
applu	1	1	1	1	1	0.7	1	1	-
apsi	1	1.05	1	1	0.53	0.93	1	1	-
equake	1	0.98	1.1	1.1	0.33	0.88	1	1	-
galgel	1	1.1	1	1.1	0.22	0.74	1	1	-
swim	1	1	1.05	1.2	0.98	0.99	1	1	-
sixtrack	1.5	1.1	0.9	0.9	0.08	0.78	1.03	1	-
wupwise	1	1	1	1	0.29	0.98	1.03	0.95	10
facerec	1	1	1	1	1	0.85	1	1	-
fma3d	1	1	1	1.02	0.25	1	0.93	0.98	20
saxpy	1	1	1	1	1	1	1	1	-
sdot	1	1	1	1	1	1	1	1	-
sfill	1	1	1	1	1	1	1	1	-
scopy	1	1	1	1	1	1	1	1	-
ssum2	1	1	1	1	1	1	1	1	-
sscale	1	1	1	1	1	1	1	1	-
striad	1	1	1	1	1	1	1	1	-
ssum1	1	1	1	1	1	1	1	1	-
tpc-c	2.0	1	1	1	0.3	1	1	0.93	5
java	1	1	1	1	1	1	1.05	0.95	5

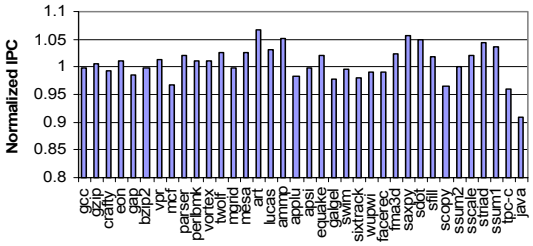

Figure 3: IPC for Synthetics Normalized to Actual Application IPCs

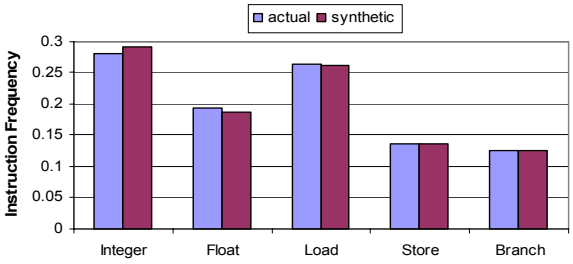

Figure 4: Average Instruction Frequencies

Table 6. Default Simulation Configuration, PowerPC ISA

Instruction Size (bytes)	4
L1/L2 Line Size (bytes)	128/128
Machine Width	8
Dispatch Window;LSQ;IFQ	120 GPRs, 120 FPRs; 32 LD, 32ST;64
Memory System	32KB 4-way L1 D, 64KB 2-way L1 I, 1.9M 10-way L2, 36MB 12-way L3
Functional Units	2 Fixed Point Units, 2 Floating Point Units
Branch Predictor	Combined 16K Tables, 12 cycle misspredict penalty

testcases were compiled on a PowerPC machine using *gcc* with optimization level *–O2* and executed to completion in the M1.

4.2 POWER5 Synthesis Results

The following figures show results either for the synthetics normalized to the original application results or for both the original applications, *actual*, and the synthetic testcases, *synthetic*. Figure 3 shows the normalized IPC for the testcases. The average IPC prediction error [10] for the synthetic testcases is 2.4%, with a maximum error of 9.0% for *java*. The other commercial workload, *tpc-c*, gives 4.0% error. 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 1.8% with a maximum of 3.4% for integers. Figure 5 shows that the basic block size varies per benchmark with an average error of 5.2% and a maximum of 18.0% for *apsi*. The largest absolute errors by far are for *mgrid* and *applu*. 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 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 eight unique block types. The top 90% of basic block frequencies in the synthetic *mgrid* differ by 27.5% on average from the basic block frequencies of the original workload. This is in contrast to testcases with large numbers of basic blocks such as *gcc*, which differ by only 3.5% for the top 90% of blocks.

The POWER5 I-cache miss rates (normalized to the maximum miss rate) are all accounted for in Figure 6, but they are not very interesting because most are less than 1%, and much less than the *tpc-c* and *java* miss rates. The low miss rates are due to the effectiveness of instruction prefetching in the POWER5 chip [25][28]. The results also support the common wisdom that the SPEC do not sufficiently challenge modern I-cache design points. The synthetic benchmarks do well on the commercial workloads but still average 7% error. These errors could probably be reduced by carrying out more synthesis passes. In general, the synthetics have larger miss rates than the applications because they are executed for fewer instructions [3]. 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 1.1%, shown normalized in Figure 7. The largest errors are *bzip2* at 5.4% and *equake* at 4.7%. The L1 data cache miss rates are shown normalized in Figure 8. The average error is 4.3% with a maximum error of 31% for eon. But *eon* has a very small miss rate, as do the

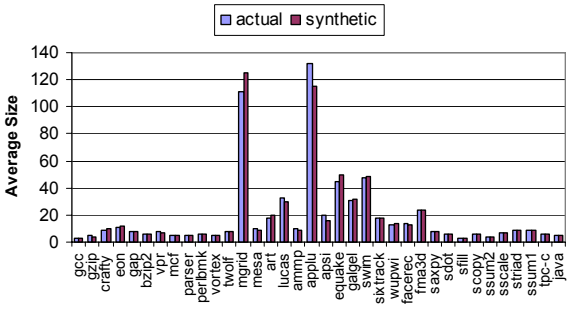

Figure 5: Basic Block Sizes

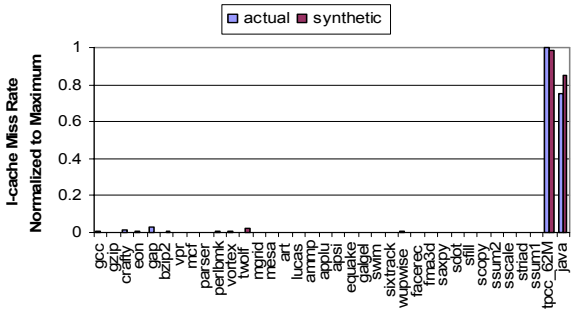

Figure 6: I-cache Miss Rate

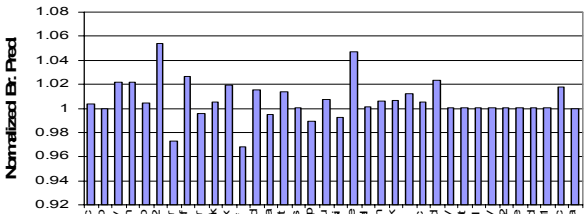

Figure 7: Branch Predictability

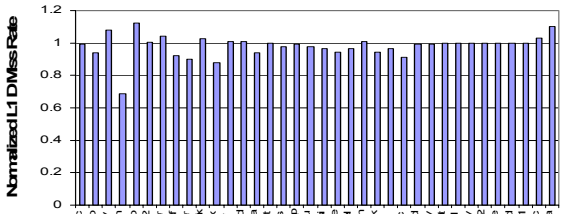

Figure 8: L1 D-cache Miss Rate

other benchmarks with errors much greater than 4%, so again the execution impact of their errors is also small. Looking at the raw miss rates, the trends using the synthetic testcases clearly correspond with those of the original workloads. This can not be seen in the normalized results.

For the unified L2 miss rates, the average error is 8.3% for those benchmarks with miss rates higher than 3%. For the others, errors can be large. Large errors due to the simple streaming memory access model are often mitigated by the small magnitude of the L1 and L2 miss rates [3]. But problematic situations occur when a large L2 miss rate error is offset by an L1 miss rate that is smaller than that of the original application. Still, for larger L2 miss rates, the trends in miss rates using the synthetics clearly correspond to those of the original applications. The L3 miss rates are also generally very small and often not represented well by the synthetics. As mentioned, research into more accurate memory access models is needed to represent all levels of the memory hierarchy.

Figure 9 shows the average dependence distances for the input operands over all benchmarks. It shows a 7.5% error on average for the non-branch dependence distances. For the branches, the M1 profiler classifies certain PowerPC trap and interrupt operations as unconditional jumps to the next basic block and their dependences are not modeled. Also, the profiler records particular condition registers as the dependences for conditional branches, while synthesis uses one specific condition register to implement the branching model. The performance impact is small compared to that of the branch predictability itself.

The integer dependence errors are caused by the conversion of many integer instructions to LSCNTRs,

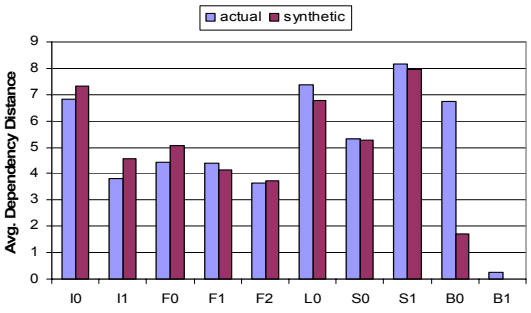

Figure 9: Dependency distances

the memory access stride counters. A stride counter overrides the original function of the integer instruction and causes dependence relationships to change. Another source of error is the movement of dependences during the search for compatible dependences in the synthesis process. The movement is usually less than one position (Table 4), but several benchmarks show significant movement. The dependence insertion technique reduces errors for the dependent instruction, but the inserted instruction may itself contain dependence errors.

4.3 Design Change Case Study: Rename Registers and Data Prefetching

We now study a set of design changes using the same synthetic codes; that is, we take the same testcases described in the last section, change the machine parameters in the M1 model and re-execute them. In this study, we increase by more than *2x* the number of rename registers available to the mapper for GPRs, FPRs, condition registers, link/count registers, FPSCRs, and XERs [28]. This effectively increases the dispatch window of the machine. We also enable data prefetching and execute in "real" mode, i.e. with translation disabled. With these enhancements the average increase in IPC for the applications for the full 100M instructions is 56.7%.

We execute the synthetic testcases using the same machine configuration. The average change in IPC for the synthetics is 53.5%. The raw IPC comparison again clearly shows that the trends in the IPC changes for all synthetics follow those of the original applications. The average absolute IPC prediction error is 13.9%, much less than half of the IPC change percentage, an important consideration when deciding if the testcase properly evaluates the design change [4].

The average relative error [10] for the change is 13.3%. The synthetic IPC change is generally lower than that of the application. The effect is explained by the use of bounded streams in PowerPC synthesis, which clamps particular synthetic stream miss rates to the base levels, as explained Section 3.

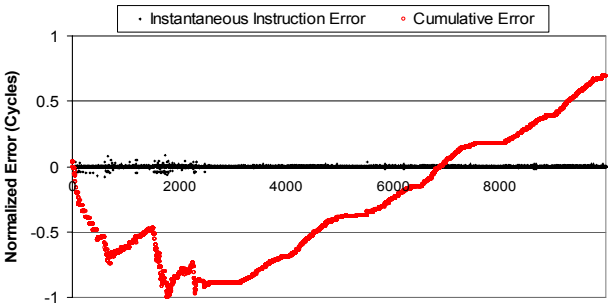

Figure 10: Instantaneous Error for Each Instruction and Cumulative Error in Cycles for 10K Instructions (*gcc*)

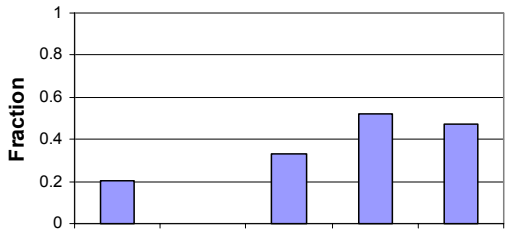

Figure 11: Fraction of Instructions with Errors (gcc)

5.0 Performance Model Validation Results

The synthetic testcases are used to validate the performance model of a POWER5 follow-on processor.

5.1 RTL Simulation

Several of the same traces used in Section 4 were executed on the VHDL and M1 models of the new processor using the process described in Section 2.2. Figure 10 illustrates the kind of detailed analysis that can be undertaken using the RTL validation methodology. Information for each instruction in the execution of 10K instructions of *gcc* is plotted, normalized to the maximum cumulative error.

The instantaneous errors are plotted as individual points either above or below the *x*-axis. If above the axis, the error is positive, meaning that the execution of the instruction in the VHDL model took longer than execution in the M1 performance model. Ideally, the performance model would execute at the same rate or slower than the RTL model, so that designers do not project overestimates of performance for their designs. The cumulative sum of the instantaneous errors is also plotted in Figure 10. It is clear that the M1 is providing overly-optimistic projections for *gcc* after only 10K instructions. The slope of the cumulative error later in the testcase indicates the direction of the performance model projections where positive is worse than flat or negative. The error is more erratic at the beginning of the execution because the early instructions are associated with the header and initialization C-code, not the body of the testcase, and they are not repeated.

The cumulative error plot also indicates repeating sequences, or phases, of behavior at various scales of analysis. The phases are related to specific code areas, and theor identification can lead to rapid performance model fixes. Viewed at one particular scale, for example, a phase can be said to start after 2800 instructions and end at about 4000. Another phase starts there and ends at 5200, and then the phases repeat. Both phases together are about as long as the body of the synthetic testcase.

To pinpoint the differences between VHDL and M1 model execution for *gcc*, we analyze the errors by instruction class as in Figure 11. All classes show a large percentage of errors (*gcc* has no floating point instructions), but Figure 12 shows that average load and store errors, whether calculated over all instructions or just instructions with errors, have the largest impact on performance.

Figure 13 breaks down the fractions of instructions with errors into buckets of errors that are multiples of 25 cycles. The vast majority of ALU and Branch operations with errors have errors that are less than 25 cycles, while 12.0% of loads and 6.8% of stores with errors have errors that are higher than 100 cycles, deep into the memory hierarchy.

Regardless of the accuracy of the memory access models used to create the synthetic streams, the results using them show that loads and stores are the least likely instructions to be modeled correctly in the performance model. This is valuable information to feed back to the performance modeling team.

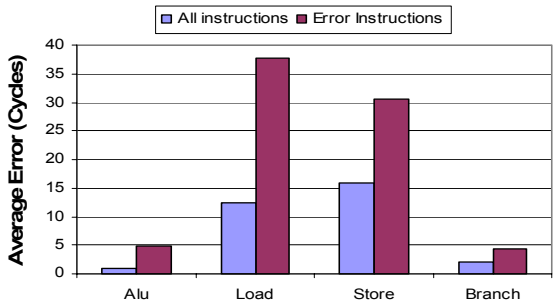

Figure 12: Average Error per Class over All Instructions and Error Instructions (*gcc*)

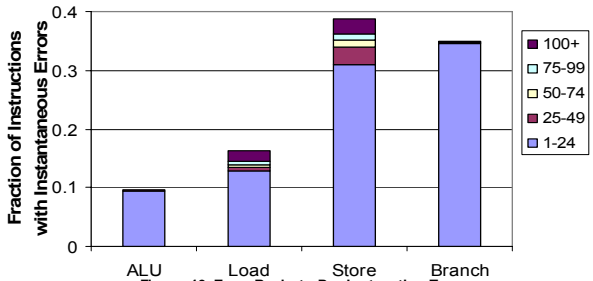

Figure 13: Error Buckets Per Instruction Type (25 Cycle Intervals)

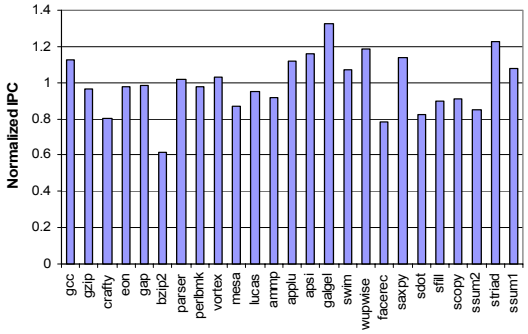

Figure 14: Normalized IPC for M1 versus AWAN using Synthetic Testcases

5.2 Hardware Emulator Validation

The same synthetic testcases are input to an AWAN hardware emulator executing the PowerPC VHDL models and to the M1 models used in the last section. In this case the runtime to collect the data is many times less than VHDL simulation [16]. Figure 14 shows the M1 IPC normalized to the AWAN results for some of the testcases. Most of the errors are within 20%, but there are several outliers, including *bzip2* and *galgel*. The hardware emulator provides more rapid VHDL simulation to speed model validation investigations.

6.0 Conclusions

The complexity of modern processors drives complexity into the performance models used to project performance, and this complexity exacerbates the problem of validating the performance models. We synthesize representative PowerPC versions of the SPEC2000, STREAM, TPC-C and Java benchmarks, compile and execute them, and obtain an average IPC within 2.4% of the average IPC of the original benchmarks. The synthetic testcases often execute two orders of magnitude faster than the original applications, typically in less than 300K instructions, making performance model validation for today's complex processors feasible. We also present the application of the synthetic testcases to performance model validation using RTL model simulation and execution on a hardware emulator.

7.0 Acknowledgements

This research is partially supported by the National Science Foundation under grant number 0429806, the IBM Center for Advanced Studies (CAS), an IBM SUR grant, the IBM Systems and Technology Division, and Advanced Micro Devices.

8.0 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, in conjunction with ISCA-31, 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] R. H. Bell, Jr. and L. K. John, "Improved Automatic Testcase Synthesis for Performance Model Validation," International Conference on Supercomputing, June 20, 2005.

[4] R. H. Bell, Jr. and L. K. John, "Efficient Power Analysis using Synthetic Testcases," IEEE International Symposium on Workload Characterization, October 7, 2005.

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

[6] 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.

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

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

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

[10] 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.

[11] 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.

[12] W. W. Hsu, A. J. Smith and H. C. Young, "Characteristics of Production Database Workloads and the TPC Benchmarks," IBM Systems Journal, Vol. 40, No. 3, 2001, pp. 781-802.

[13] I. Hur and C. Lin, “Adaptive History-Based Memory Schedulers,” Proceedings of the International Symposium on Microarchitecture, 2004.

[14] H. Jacobson, P. Bose, Z. Hu, A. Buyuktosunoglu, V. Zyuban, R. Eickemeyer, L. Eisen, J. Griswell, D. Logan, B. Sinharoy and J. Tendler, “Stretching the Limits of Clock-Gating Efficiency in Server-Class Processors,” Proceedings of the International Symposium on High-Performance Computer Architecture, February 2005.

[15] S. R. Kunkel, R. J. Eickemeyer, M. H. Lipasti, T. J. Mullins, B. O’Krafka, H. Rosenberg, S. P. VanderWiel, P. L. Vitale and L. D. Whitley, “A Performance Methodology for Commercial Servers,” IBM J. Res. & Dev. Vol. 44, No. 6, November 2000, pp. 851-872.

[16] J. M. Ludden, *et al.*, “Functional Verification of the POWER4 Microprocessor and the POWER4 Multiprocessor Systems,” IBM J. Res. Dev., Vol. 46, No. 1, January 2002.

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

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

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

[20] 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.

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

[22] 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.

[23] 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.

[24] R. Singhal, et al., “Performance Analysis and Validation of the Intel Pentium4 Processor on 90nm Technology,” Intel Tech. J., Vol. 8, No. 1, 2004.

[25] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer and J. B. Joyner, “POWER5 System Microarchitecture,” IBM J. Res. & Dev. Vol. 49 No. 4/5, July/September 2005, pp. 505-521.

[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. W. Victor, *et al.*, “Functional Verification of the POWER5 Microprocessor and POWER5 Multiprocessor Systems,” IBM J. Res. & Dev., Vol. 49, No. 4/5, July/September 2005, pp. 541-553.

[30] 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.