The Case for Automatic Synthesis of Miniature Benchmarks

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

There are two parts of the design process that can benefit from reduced, miniature benchmarks that behave like longer-running applications during simulation: 1) the early design phase before an implementation exists, and 2) the later design phases when cycle-accurate functional models exist. In the early design phase, many hundreds or thousands of potential design tradeoffs must be evaluated rapidly at a high-level. In the later design phases, design changes are costly to undo, so potential changes need to be accurately evaluated using a performance model that has been validated against a functional model. Applications typically run too long to be executed completely for either early design tradeoffs or late performance model validation.

In this paper, we explore the potential for automatically synthesizing reduced miniature benchmarks from the execution characteristics of actual applications. We discuss the advantages and challenges of automatic synthesis and present evidence that miniature benchmarks can reproduce the machine behavior of much longer running applications. We present one approach to benchmark synthesis and show that IPC and many average characteristics of the executing synthetic benchmarks are similar to those of the applications that the synthetic benchmarks are derived from. We also show that an early design task like identifying performance trends due to design changes can be carried out while still reducing runtimes significantly. The synthetic benchmarks converge to results rapidly, enabling performance model validation.

1. Introduction

Over two decades ago, researchers used synthetic benchmarks like Whetstone[10] and Dhrystone[30] to approximate the performance of applications on their designs. However, the early synthetic benchmarks fell out of favor because they were difficult to maintain and upgrade in the face of ever-evolving languages, libraries and programming styles. In the early phases of the pre-silicon design process, researchers turned to simulation of real applications, and some applications (like SPEC [26]) have became benchmarks of computer performance. However, long runtimes for the latest benchmarks make full program simulation for early design studies impractical [22] [33].

In the later phases of the pre-silicon design process, the validation of a performance model against a functional model or hardware is necessary at various times in order to minimize incorrect design decisions due to inaccurate performance models [5]. As functional models are improved, accurate performance models can pinpoint with increasing certainty the effects of particular design changes. This translates into higher confidence in late pre-silicon or second-pass silicon design performance.

Prior validation efforts have focused on handwritten microbenchmarks or short tests of random instructions [6] [27] [4] [18] [17] [16] [34]. Black and Shen describe tests of up to 100 randomly generated instructions [4], not enough to approximate many characteristics of applications. Desikan *et al.* use microbenchmarks to validate an Alpha 21267 simulator to 2% error [11], but the validated simulator still gives errors from 20% to 40% when executing the SPEC2000 benchmarks.

Ideally, SPEC and other applications would be used for performance model validation, but, again, this is limited by their long runtimes on functional simulators [5]. In [36], only one billion simulated cycles *per month* are obtained. In [34], farms of machines provide many cycles in parallel, but individual tests on a 175 million-transistor chip model execute orders of magnitude slower than the hardware emulator speeds of 2500 cycles per second.

Sampling techniques such as SimPoint [22], SMARTS [33] and Intrinsic Checkpointing [35] can reduce application runtimes, making early design studies feasible, but it is still necessary to execute tens of millions of instructions. Statistical simulation creates representative synthetic traces with less than one million instructions [8] [19] [12], but traces are not useful for functional model validation.

Sakamoto *et al.* combine a modified trace snippet with a memory image for execution on a specific machine and a logic simulator [21], but the method is machine-specific

and there is no attempt to reduce the total number of simulated instructions. 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]. Wong and Morris [32] investigate synthesis for the LRU hit function to reduce simulation time, but no method of simultaneously incorporating other workload characteristics is developed. The research community recognizes the need for a general synthesis method [23], but none has been forthcoming.

In this paper, we discuss the problem of synthesizing reduced, miniature benchmarks for early design studies and performance validation. We describe an example synthesis system that uses the workload characterization and graph analysis of statistical simulation in combination with specific memory access and branching models as in [2][3]. A miniature benchmark 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 yet it can be easily executed on a variety of performance and functional simulators, emulators, and hardware, and with significantly reduced runtimes.

The rest of this paper is organized as follows. Section 2 presents properties necessary for the miniature benchmarks and some of their benefits. Sections 3 and 4 gives an overview of the synthesis approach and some experimental results. Section 5 gives some discussion, and the last sections present conclusions and references.

2. Representative Miniature Benchmarks

Automatic benchmark synthesis is most useful if the synthesized benchmark has the following two properties:

1) The benchmark reproduces the machine execution characteristics of the application upon which it is based.

2) The benchmark converges to a result much faster than the original application.

If the first property holds, the benchmark is said to be *representative* of the original application, at least over some range of instructions or workload characteristics. The workload characteristics can be categorized into two classes [13]: *microarchitecture-independent* metrics such as instruction mix, dependency distances, basic blocks, and temporal and spatial locality; and *microarchitecture-dependent* metrics such as cache miss rates and branch predictability. If the second property does not hold to some degree, there is no good reason to use the synthetic benchmark over the original application.

Prior work usually focuses on one of the properties at the expense of the other, or on both properties but over a narrow range. The hand-coded tests and automatic random tests in Black and Shen [4] converge quickly (property 2), but they provide limited or inefficient coverage of all the instruction interactions in a real application (property 1). The *reverse-tracer* system [21] achieves accurate absolute performance for a short trace (property 1), but no runtime speedup is obtained. In [14], both properties are achieved, but workload characteristics that are important to performance, like the instruction sequences and the dependency distances [1], are not maintained. Intrinsic Checkpointing [35] achieves both properties but is only incrementally faster than SimPoint [22].

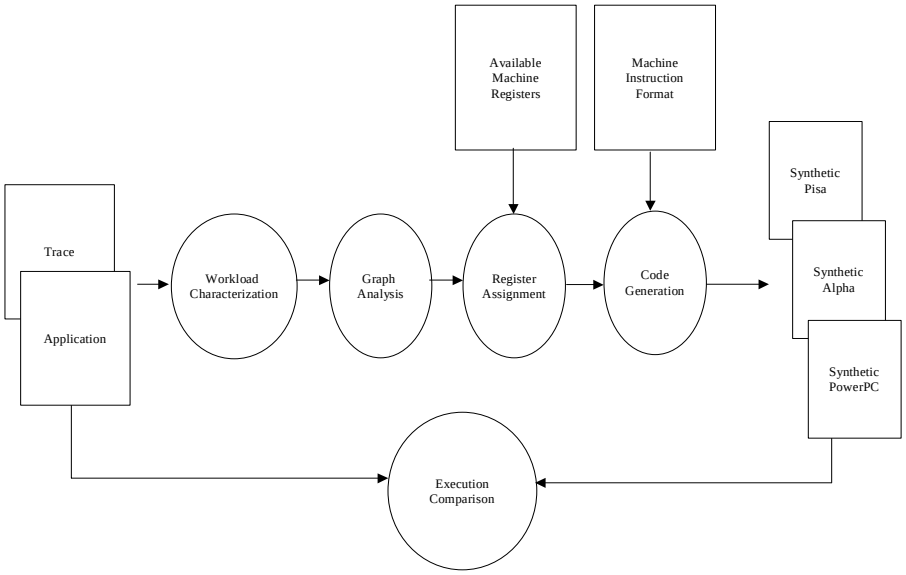

Figure 1: Miniature Benchmark Synthesis and Simulation Overview

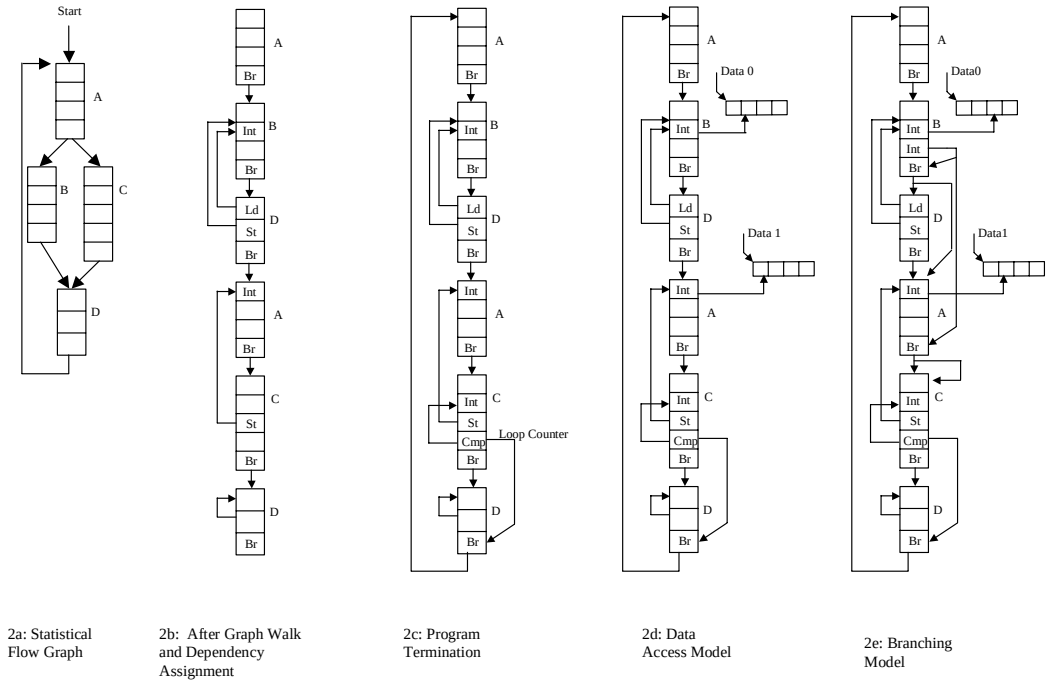

Figure 2: Benchmark Synthesis Illustrated

In practice, achieving both properties for the representative phases of an entire workload is difficult, but in most cases it is not necessary. For validation purposes, a reduced synthetic benchmark need represent only specific application features of interest, not all features. For early design studies, many prominent workload features must be represented, but absolute accuracy [12] need not be high as long as performance trends from design changes are visible, i.e. *relative* accuracy [12] is high.

Any miniature benchmark synthesis system that satisfies the two properties will introduce errors because the application size, in terms of the numbers of instructions executed, has been scaled down in order to obtain a runtime speedup. Ideally, the benchmark will be generated in a high-level language like C. Assuming that the errors can be kept at acceptable levels, such high-level benchmarks have several advantages over sampled traces. These include: portability to a variety of machines, emulators, and execution-driven simulators; and flexibility with respect to easier modification of the code to study changes in workload characteristics or future workloads.

In the next section, we present a synthesis system that takes a step toward automatically generating reduced, miniature benchmarks that can satisfy both properties simultaneously for many workload characteristics. We discuss the modeling abstractions and the errors that are introduced by the synthesis process.

3. Example Synthesis System

Figure 1 depicts the synthesis process at a high level. There are four major phases: *workload characterization; graph analysis; register assignment* and *code generation*. In this paper we give an overview of the synthesis process and philosophy. Additional detail, as well as exact synthesis parameters and algorithms for *Pisa* and *Alpha* code targets, can be found in [2] and [3].

Figure 2 gives a step-by-step illustration of the synthesis process, which we describe below. At a high level, we start with the statistical flow graph from statistical simulation [12][1], which is a reduced representation of the control flow instructions of the application. The graph is walked, giving a representative synthetic trace. We then apply algorithms to instantiate low-level instructions, specify branch behaviors and memory accesses, and generate code, yielding a simple but flexible program.

3.1. Workload Characterization

The dynamic workload characteristics of the target program are profiled using a functional simulator, cache simulator and branch predictor simulator as in [12][1]. The characterization system currently takes input from fast functional simulation using SimpleScalar [7] or trace-driven simulation in an IBM proprietary performance simulator. We characterize the basic block instruction

sequences, the instruction dependencies, 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 basic classes: integer, floating-point, load, store, and branch. Long and short execution times for integer and floating-point instructions are distinguished. We also track the IPC of the original workload to compare to the synthetic result.

The statistical flow graph [12][1] is assembled from the workload characterization. An example is given in Figure 2a. Basic blocks A, B, C and D each has various probabilities of branching to one or more basic blocks.

3.2. Graph Analysis

We use the workload characterization to build the pieces of the synthetic benchmark. The statistical flow graph is walked using the branching probabilities for each basic block, and a linear chain of basic blocks is assembled, as in Figure 2b. This chain will eventually be emitted directly as the central operations of the synthetic benchmark. The number of instantiated basic blocks is equal to an estimate of how many blocks are needed to match the I-cache miss rate of the application given a default I-cache configuration [2][3]. We then tune the number of synthetic basic blocks to match the I-cache miss rate and instruction mix characteristics by iterating through synthesis a small number of times. In practice, anywhere from one to 1000 basic blocks may be necessary to meet the I-cache miss rate of a particular application. Typically less than 4000 instructions are synthesized.

For each basic block, we assign instruction input dependencies (also Figure 2b). The starting dependence for each instruction is taken from the average found for the instruction during workload characterization. If the dependency is not compatible with the input operand type of the dependent instruction, then another instruction is chosen. The algorithm is to move forward and backward from the starting dependency through the list of instructions until the dependency is compatible. In practice, the average number of moves per instruction input is small, usually less than one. For loads and stores, the data address register must be of integer type. When found, it is attributed as a memory access counter for special processing during the code generation phase.

When all instructions have compatible dependencies, a search is made for an additional integer instruction that becomes the loop counter (Figure 2c). The branch in the last basic block in the program checks the loop counter to determine when the program is complete. The number of executed loops is chosen to be large enough to assure IPC convergence given the memory accesses of the load and store instructions in the benchmark. 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/4000* for most benchmarks.

The data access counter instructions are assigned a stride based on the D-cache hit rate found for the corresponding load and store accesses during workload characterization (Figure 2d). The memory accesses for data are modeled using the sixteen simple stream access classes shown in Table 1. The table was generated based on a default cache configuration [2][3], and the stride is shown in four byte increments. The stride for a memory access is determined first by matching the L1 hit rate of the load or store that is fed by the access counter, after which the L2 hit rate for the stream is predetermined.

By treating all memory accesses as streams, the memory access model is kept simple. This reduces changes to the instruction sequences and dependencies, which have been shown to be critical for correlation with the original workload [1]. On the 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 of the table. Section 4 shows that the simple model is responsible for correlation error when the cache hierarchy changes from the default. More complicated models might walk cache congruence classes or pages (to model TLB misses), or move, add, or convert instructions to implement specific functions. Adding a few instructions to implement a more complicated model will not impact instruction mix and behavior in most cases. There are many models in the literature that can be investigated as future work [24][29][9][15].

In some cases, we found that additional manipulation of the streams was necessary for correlation of the benchmarks because of the cumulative errors in stream selection. Parameters were added to adjust the basic block

Table 1: 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

and overall miss rates of the synthetic data accesses to compensate. A small number of synthesis iterations is usually necessary. Details are given in [2][3].

We superimpose a branch predictability model onto the set of basic blocks that already represent the instruction mix, dependencies and data access patterns of the original workload (Figure 2e). A number of branches in the trace are configured to branch past the next basic block or a number of instructions based on the global branch predictability of the original application. An integer instruction that is not used as a data access counter or a loop counter is converted into an *invert* instruction that operates on a particular register every time it is encountered. If the register is set, the branch jumps past the next basic block. The *invert* mechanism causes a branch to have a predictability of 50% for predictors that use 2-bit saturating counters. Benchmarks like *mgrid* and *applu* have average basic block sizes much longer than other benchmarks. In those cases, parameters are used to adjust the synthetic branch predictability. Details are given in [2][3].

The configured branches, invert instruction, and loop counter must not be skipped over by a taken branch, or loop iterations may not converge, or the branch predictability may be incorrect. Code regions containing these instructions are carefully avoided.

In practice, there are many synthetic benchmarks that more or less satisfy the metrics obtained from the workload characterization and overall application IPC. As mentioned in several places above, the usual course of action is to iterate through synthesis a number of times until the metric deltas are as small as desired. Usually less than ten iterations are needed to obtain reasonably small errors.

3.3. Register Assignment

All architected register usages in the synthetic benchmark are assigned exactly during the register assignment phase. Most ISAs specify dedicated registers that should not be modified without saving and restoring. In practice, not all registers need be used to achieve a good synthesis result. In our experiments, only 20 or so general-purpose registers divided between data access counters and code use are necessary.

Data access streams are pooled according to their stream access characteristics and a register is reserved for each class. All data access counters in the same pool increment the same register, so new stream data are accessed similarly whether there are a lot of counters in the pool and few loop iterations or few in the pool but many iterations. The exact numbers of data access and stream registers assigned for each benchmark are given in [2][3].

For applications with large numbers of stream pools, synthesis consolidates the least frequent pools together until the total number of registers is under the register use limit. 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 takes the representative instructions and the attributes from graph analysis and register assignment and outputs a single module of C-code that contains calls to assembly-language instructions in the target language [2][3]. Figure 1 shows the three targets currently supported. Each instruction in the representative trace maps one-to-one to a single *asm* call in the C-code. Ordinary C-code is emitted for functions not related to the trace, as, for example, to instantiate and initialize data structures and variables.

We emit a C-code *main* header and variable declarations to link output registers to data access variables for the stream pools, the loop counter variable, and the branching variable. Pointers to the correct memory type for each stream pool are declared, and *Malloc* calls for the stream data are generated with size based on the number of loop iterations. Each stream pool register is initialized to point to the head of its *malloced* data structure.

The loop counter register initialization is emitted, and the instructions associated with the original graph walk are emitted as volatile calls to assembly language instructions. The data access counters are emitted as integer additions of its output register value to the associated stride for the stream. The loop counter is emitted as an integer subtraction of one to its output register. 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 emitted.

Additional detailed synthesis information for the SPEC95, SPEC2000 and STREAM benchmarks can be found in [2] and [3].

4. Synthesis Results

In this section we present the SPEC2000 results for the benchmark synthesis system described in the last section.

4.1. Experimental Setup and Benchmarks

We start with an experimental system that exhibits good synthetic simulation correlation against actual application simulations. Our system is derived from the statistical simulation system HLS [19][20], which we updated with the statistical flow graph to improve correlation [1][12]. SimpleScalar 3.0 [7] was downloaded and *sim-cache* was modified to carry out the workload characterization. The twelve SPECint2000 and fourteen SPECfp2000 *Alpha* binaries were executed in *sim-outorder* on the first reference dataset for the first billion instructions. Single-precision versions of eight STREAM and STREAM2 benchmarks [16] with a ten million-loop limit were also simulated. We use the default SimpleScalar configuration in Table 2, as in [19]. SimpleScalar does not model an L3, but the memory latency estimates a fast L3.

Code generation was enabled and C-code was produced using the synthesis methods of Section 3. The synthetic benchmarks were compiled using *gcc* with optimization level *–O2* and executed to completion in SimpleScalar on an IBM p270 (400 MHz).

4.2. Synthesis Results

The synthetic benchmarks have an execution speed advantage over the original applications. Most simulations of the SPEC2000 synthetics take less than four seconds to execute an average of 325K instructions, compared to about 25K seconds for the original codes. On average, the applications simulate 6000 times slower than the synthetics [3].

The STREAM synthetics must also execute for a large number of instructions (283K on average) in order to represent to be represent the data access patterns of the original codes. As a result, the original codes of 10M dynamic instructions execute only 35 times slower than the synthetic codes. Dynamic executions of at least 300K instructions provide data access convergence and also limit the code overhead of the synthetic to less to 1% of the total dynamic instructions.

Table 2: 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 mispredict penalty

Table 3: Percent Execution Error, Synthetics vs. Applications

Metric	Avg. %Error	Max. %Error
IPC	2.4	8.0 (facerec)
Instruction Frequencies	3.4	7.3 (branches)
Dependency Distances	11.1	34.9 (integers)
Dispatch Occupancies	4.1	8.7 (floats)
Basic Block Sizes	7.2	21.1 (mgrid)
L1 I-cache Miss Rate (>1%)	8.6	22.9 (sixtrack)
L1 D-cache Miss Rate (>1%)	12.3	55.7 (mgrid)
L2 Cache Miss Rate (>15%)	18.4	61.2 (applu)
Branch Predictability	1.5	6.4 (art)

Table 3 compares the execution characteristics of the synthetic benchmarks to those of the SPEC2000 and STREAM codes. The average percent errors for all the metrics are generally less than 15%, with most below 10%, although some of the maximum errors are high. The error in IPC remains low because errors in the metrics offset each other for particular benchmarks, or the absolute values of the metrics are very low and have little effect. As an example, the large percent error for the *mgrid* L1 D-cache miss rate corresponds to a reduction for the synthetic that is offset by a 0.3% increase in I-cache miss rate and a 21.1% decrease in basic block size. As another example, the 22.9% L1 I-cache miss rate error for *sixtrack* is a decrease taken against a miss rate of just 1.1%, so the effect of the error is small. Also, the large increase in L2 cache misses for *applu* is offset by a 31.5% decrease in its L1 D-cache miss rate, to 6.5%. The various errors in Table 3 are broken out for each benchmark in [3].

The overall average IPC error for the synthetic benchmarks is 2.4%, with a maximum error of 8.0% for *facerec*. The error in IPC expresses the average effect of small or offsetting errors among the workload characteristics of the synthetics as described below.

The average error in instruction frequencies over the five classes of instructions for the synthetic benchmarks is 3.4% with a maximum of 7.3% for branches. The basic block size varies per synthetic with an average error of 7.2% and a maximum error of 21.1% 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 average I-cache miss rate error is 8.6% for benchmarks with miss rates above 1%. However, the number of synthetic instructions is within 2.8% of the expected number given the I-cache configuration. The errors are 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 the maximum number that fits in the default cache is typically used, up to the number needed to give an appropriate instruction mix for the benchmark. For the STREAM loops, only one basic block is needed to meet the instruction mix and miss rate requirements. For all synthetic benchmarks there is a small but non-zero miss rate, versus an essentially zero miss rate for some of the applications. This is because the synthetic benchmarks 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, the impact, when combined with the miss penalty, is also small.

The average branch predictability error is 1.9%, with a maximum error for *art* of 6.4%. *Mgrid*, with its large basic block size error, has the third largest error at 4.9%.

For L1 data cache miss rates greater than 1%, the average error is 12.3%. Despite this error, the trends in D-cache miss rates generally correspond with those of the original workloads [3]. There is some variation for smaller miss rates, but, as with many I-cache miss rates, the execution impact is also small.

The unified L2 miss rates have a large average error of 18.4%. The large error is due to the simple streaming memory access model. However, the errors are often mitigated by small L1 miss rates. A good example is *gcc*, which has a 15% L2 miss rate but only a 2.6% L1 miss rate. The 61.2% L2 miss rate error for *applu* is offset by I-cache and L1 D-cache miss rates that are smaller than those of the original workload. *Art* and *ammp* have large L1 miss rates (41% and 44%), but their L2 miss rates are offset by relatively larger I-cache miss rates and smaller branch predictabilities. The main cause of the 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. A large L2 miss rate error for *ammp* (46%) is explained by the fact that our small data-footprint synthetic benchmarks 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.

Table 4: Design Change Percent Error and Percent Relative Error Synthetics vs. Applications

Design Change	Avg. %Error	Avg. %Rel. Err.
Dispatch Window 8, LSQ 4	2.8	2.4
Dispatch Window 32, LSQ 16	3.7	2.1
Dispatch Window 48, LSQ 24	4.9	3.8
Dispatch Window 64, LSQ 32	6.1	5.1
Dispatch Window 96, LSQ 48	8.3	7.5
Dispatch Window 128, LSQ 64	9.0	8.3
Machine Width 2	2.7	1.6
Machine Width 6	2.6	1.1
Machine Width 8	2.6	1.1
Machine Width 10	2.6	1.1
Issue Width 1	1.9	2.3
Issue Width 8	2.7	1.0
Commit Width 1	2.8	2.1
Commit Width 8	2.4	0.2
Instruction Fetch Queue 8	2.6	0.5
Instruction Fetch Queue 16	2.7	0.8
Instruction Fetch Queue 32	3.0	1.1
Caches 0.25x	20.1	19.4
Caches 0.5x	24.8	23.9
Caches 2x	4.1	3.3
Caches 4x	4.7	3.8
L1 I-cache 2x	3.0	1.3
L1 D-cache 2x	3.1	1.0
L1 D-cache Latency 8	9.5	9.7
BP Table 0.25x	2.5	1.1
BP Table 0.5x	2.3	0.3
BP Table 2x	2.3	0.3
BP Table 4x	2.3	0.4

The average dependency distances have 11.1% error on average. The largest components of error are the integer dependencies (at 34.1%), caused by the conversion of many integer instructions to data access counters. A data access 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. The movement is usually less than one instruction position, as mentioned earlier, but *mgrid* and *applu*, the benchmarks with the largest average block sizes at 100.1 and 93.4, respectively, show significant movement. The branching model also contributes errors to the integer instruction class.

In spite of the dependency distance errors, the average dispatch window occupancies are similar to those of the original benchmarks with an average error of 4.1%.

4.3. Using the Synthetic Benchmarks to Assess Design Changes

We now study design changes using the same synthetic benchmarks; that is, we take the benchmarks described in the last section, change the machine parameters in SimpleScalar and re-execute them. Table 4 gives the average IPC prediction error and relative IPC error [12] when executing various design changes on the benchmarks synthesized from the default configuration. A change in machine width implies that the decode width, issue width and commit width all change by the same amount from the base configuration in Table 2. When the caches are increased or decreased by a factor, the number of sets for the L1 I-cache, D-cache and L2 cache are increased or decreased by that factor. Likewise, when the bimodal branch predictor is multiplied by a factor, the table size is multiplied by that factor from the default size. The *L1 D-cache 2x* and *L1 I-cache 2x* specify a doubling of the L1 D-cache (to 256 sets, 64B cache line, 8-way set associativity), and a doubling of the L1 I-cache configuration (to 1024 sets, 64B cache line, 2-way set

associativity). The numbers here 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 benchmarks do not model data-TLB misses.

Looking at the *Dispatch Window* rows, when the synthetics are executed on configurations close to the default configuration, the average IPC prediction error and average relative errors are below 5%. However, as the configuration becomes less similar to the configuration used to synthesize the benchmarks, the errors increase. One conclusion is that the synthetics are most useful for design studies and validations closer to the synthesis configuration, and that the miniature benchmarks should be resynthesized when the configuration strays farther away - in this case, when the dispatch window rises to four times the default size.

When the errors are small, the changes in IPC for the applications and synthetics are very similar. If the IPC changes are significantly larger than the errors in IPC due to synthesis modeling, the changes using the synthetics should be large enough to trigger additional studies using a detailed cycle-accurate simulator. For early design studies, chip designers are looking for cases in a large design space in which a design change may improve or worsen a design. Example analyses are given in [2][3].

The *Machine Width* results differ from the dispatch window results in that the errors are small regardless of the width change. For the dispatch studies, the absolute change in IPC from the default configuration for both synthetics and applications is greater than 17% for each case (for a dispatch window of 128 the change is over 56%). Likewise, when the width is reduced to 2, the absolute change in IPC is over 23%, which indicates that the low average prediction error and relative error are meaningful. But when the width increases to 6, 8 and 10, the change is never more than 2.3%, which is on the order of the IPC prediction errors of the synthetics versus the applications. However, the fetch queue size did not change from the default and it supplies too little ILP to stress the wider machine width. These configurations therefore cannot test the accuracy of the synthetics.

Similarly, the absolute IPC change from the default IPC for the *Issue Width 8* and *Commit Width 8* rows never gets greater than 1.4%, and likewise for the instruction fetch queue and branch predictability rows, it never gets greater than 1.6%. Simply changing the IFQ size, issue, or commit width without addressing the other pipeline bottlenecks does not improve performance, as expected. Other workloads may be needed to stress the branch predictor.

The remaining studies yield changes in IPC significantly greater than the error of the synthetics versus the applications, except for the *Caches 0.25x* and *Caches 0.5x* studies. The synthetics underestimate performance when the cache is significantly reduced due to capacity misses among the synthetic data access streams.

For the L1 D-cache latency study, the average absolute IPC error is 9.5% and the relative error is 9.7%, significantly less than the 22.1% change in IPC from the default configuration. On further investigation, the error is mostly due to the SPECint synthetics, with an average relative error of 19.9%, versus 4.2% for the others. Despite these errors, the IPC changes in the actual benchmarks executing one billion instructions are still visibly reflected in the synthetic benchmarks that run in seconds [3].

Again, all of these runs use the same miniature benchmarks synthesized from the initial SimpleScalar configuration, not re-synthesized benchmarks.

5. Drawbacks and Discussion

The main drawback of the approach is that the microarchitecture independent workload characteristics, and thus the synthetic workload characteristics, are dependent on the particular compiler technology used. However, since the process is automatic, resynthesis based on workload characterization from new compiler technology is simplified. It also avoids 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].

One objection is that the synthetics are comprised of machine-specific assembly calls. However, use of low-level operations are a simple way to achieve true representativeness in a much shorter-running benchmark, and the *asm* calls are easily retargeted to other ISAs that follow the RISC philosophy.

Another drawback is that only features specifically modeled among the workload characteristics appear in the synthetic benchmark. This will be addressed over time as researchers uncover additional features needed to correlate with execution-driven simulation or hardware, although the present state-of-the-art is quite good [12][1]. In the future, synthesis parameters could be used to incorporate or not incorporate features as necessary.

One consequence of the present method is that dataset information is assimilated into the final instruction sequence of the synthetic benchmark. For applications with multiple datasets, a family of synthetic benchmarks must be created. The automatic process makes doing so possible, but future research could seek to find the workload features related to changes in the dataset and model those changes as runtime parameters to the synthetic benchmark.

Ideally, our miniature programs would be benchmark replacements, but the memory and branching models used in their creation introduce significant errors. This makes

them a solution in the "middle" between micro-benchmarks and applications. However, as shown in Section 4, many of the characteristics of the original applications are maintained, and the synthesis approach provides a framework for the investigation of advanced cache access and branching models each independently of the other.

Our benchmarks use a small number of instructions in order to satisfy the I-cache miss rate. This small number causes variations in workload characteristics, including basic block size, with corresponding changes in instruction mix, dependency relationships, and dispatch window occupancies. One solution is to instantiate additional basic blocks using *replication* [32]. Multiple sections of representative synthetic code could be synthesized and concatenated together into a single benchmark. Each section would satisfy the I-cache miss rate, but the number of basic blocks would increase substantially to more closely duplicate the instruction mix. Similarly, multiple sections of synthetic code, and possibly initialization code, could be concatenated together to recreate program phases [22]. Additionally, phases from multiple benchmarks could be consolidated together and configured at runtime through user parameters.

6. Conclusions

In this paper we explore the possibility of automatically synthesizing reduced, miniature benchmarks from the dynamic workload characteristics of executing applications. Two major uses of the miniature benchmarks are for rapid early design studies and pre-silicon performance validation. We discuss the advantages and challenges of automatic synthesis and describe an example system in which the target application's executable is analyzed in detail and sequences of instructions are instantiated as in-line assembly-language instructions inside C-code.

Unlike prior synthesis efforts, we focus on the low-level workload characteristics of the executing binary to create a workload that behaves like a real application executing on the machine. Multiple synthetic benchmarks 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 benchmarks and enables consolidation of multiple representative phases into a single small benchmark. Other benefits include portability to various platforms and flexibility with respect to benchmark modification. Future work includes more accurate memory access models and branching models.

Aknowledgements

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, by the IBM Center for Advanced Studies (CAS), and an IBM SUR grant.

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] R. H. Bell, Jr. and L. K. John, "Improved Automatic Testcase Synthesis for Performance Model Validation," to appear in the Proceedings of the ACM International Conference on Supercomputing, June 2005.

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

[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," Proceedings of the 24th Annual 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," Proceedings of 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," 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," 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, November 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, December 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," Proceedings of the 27th Annual 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,” Proceedings of the International 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 International 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 Journal of Research and Development, 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 Transaction 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,” The International Symposium on Computer Architecture, June 2002.

[34] 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.

[35] J. Ringenberg, C. Pelosi, D. Oehmke and T. Mudge, “Intrinsic Checkpointing: A Methodology for Decreasing Simulation Time Through Binary Modification,” International Symposium on Performance and Simulation Systems, March 2005, pp. 78-88.

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