Experiments in Automatic Benchmark Synthesis

Robert H. Bell, Jr. Lizy K. John
The University of Texas at Austin

Abstract

In the nineteen-eighties, synthetic workloads such as Whetstone and Dhrystone fell out of favor as benchmarks of computer performance because they became unrepresentative of the performance of continuously-evolving applications. Hand-coded synthetic benchmarks take work to develop and maintain, are language feature specific, and are subject to compiler optimizations that eliminate code meant to make a significant contribution to performance.

We present an automatic benchmark synthesis method that addresses these problems. The method automatically creates C-code that, when compiled and executed, is representative of the features of a target application but executes in a fraction of the original runtime. Our benchmark synthesis technique takes an actual executable, performs control flow analysis and workload characterization, and generates a representative synthetic benchmark. The representative sequences of instructions are instantiated as in-line assembly-language instructions in the synthetic benchmark.

We synthesize versions of the SPEC95 and STREAM benchmarks with both perfect branching and a simple branching model. We find that benchmarks can be synthesized to an average IPC within 3.9% of the average IPC of the target benchmarks with remarkably similar instruction mix, cache access characteristics, RUU occupancies, and dependency characteristics. In addition, the change in IPC for a synthetic benchmark due to a design change is found to be proportional to the change in IPC for the original application. The synthesized versions of the SPEC95 benchmarks execute in 0.1% of the original execution time.

1. Introduction

Hand-coded synthetic benchmarks such as Whetstone [CURN76] and Dhrystone [WEIK84] were developed to represent the new language features of ever-evolving workloads running on increasingly complex microarchitectures. The benchmarks were *portable* to multiple platforms, but they were quickly outmoded as the falling price of system memory enabled the proliferation of applications using lots of memory, unusual and inefficient coding styles, new language paradigms, standard code libraries and object-oriented features. A major problem was that compilers would *cheat* and eliminate code specifically included to test machine performance but not contributing to a functional result [WEIK95]. Other problems included a lack of standards related to compiler parameters, input dataset use, and performance metrics [HENN96].

Microbenchmarks were developed from representative computational units [WONG88] [SREE74] [WILL76] but the techniques are ad-hoc in nature and still subject to compiler elimination, and like synthetic *kernel* programs [MCMA86], were not representative of any complete application. The *consolidation* of multiple applications into a small representative synthetic benchmark was hampered by rapid obsolescence and maintenance difficulties.

As a result, researchers have relied heavily on the runtimes of real applications to assess computer performance [HENN96]. Recently there has been an *explosion* of applications in use as benchmarks. A partial list is given in [LILJ00]. A designer would like to use all benchmarks to evaluate his design, but execution times even on the fastest event-driven simulators can amount to days to evaluate a single design choice for one application [SHER02] [WUND03] [EECK03]. Reduced input datasets have been studied [KLEI00], but results to date have been mixed [TODI01].

Recent work has shown that running benchmarks in suites like SPEC cover the same *redundant* workload characteristics [SAAV92] [DUJM98] [EECK03] [VAND04]. Redundancy internal to programs, in the form of *phases* [SHER02] and sufficient trace *samples* [WUND02], has also been identified. Nevertheless, even the fastest trace sampling techniques require hours to evaluate a single design choice [WUND03]. Trace-driven simulation reduces runtime, but traces can be prohibitively large and are not easily modified for design spaces exploration. So *statistical simulation* has been developed [CARL98] [OSKI00] [NUSS01] [EECK03] [EECK04] [BELL04].

Statistical simulation systems use workload characteristics from execution-driven simulations to generate statistics that are then used to create a dynamic input trace for a flexible execution engine. Studies have focused on achieving *absolute correlation* with respect to instructions-per-cycle (IPC) versus execution-driven simulators, and absolute errors below 5% have been achieved with much reduced runtimes [EECK04]. Good *relative* accuracy has also been achieved; i.e. the difference in machine performance found for a microarchitectural change in statistical simulation is proportional to the difference found using execution-driven simulation [EECK03].

The traces used in statistical simulation are not easily portable to various platforms, they cannot be executed on hardware, emulation systems, functional models, or execution-driven simulators, and they cannot accurately model specific locking mechanisms and data sharing in the increasingly sophisticated designs of multiprocessor systems. Chip complexity [TEND02] is driving a trend toward more accurate execution-driven and functional model simulation, but the benchmark explosion and long simulation times make comprehensive design studies prohibitive.

What is needed is a method to synthesize benchmarks with the flexibility of source-code, the representativeness of actual applications, and the efficiency of statistical simulation. An automatic method would eliminate the problem of benchmark obsolescence. The design community recognizes the need for synthetic benchmarks and an automatic way to generate them [SKAD03], but no such method has been forthcoming.

The major contribution of this paper is just such an automatic method. The method generates synthetic benchmarks from actual application executables. The core of the technique is to use graph analysis and workload characterization to capture the essential structure of the program. The benchmarks are generated in C-code with low-level instructions instantiated as *asm* statements. We show that synthetic benchmarks created using this method have an absolute IPC within 3.9% on average of the IPCs of the SPEC95 and STREAM benchmarks. Also, the IPC change for a synthetic benchmark due to a design change is proportional to the IPC change for the original application. The runtimes are generally three orders of magnitude shorter than those of the original application.

The rest of this paper is organized as follows. Section 2 presents the conceptual framework for our benchmark synthesis method. Section 3 describes the synthesis approach in detail. Section 4 presents experimental results using the system. Section 5 presents related work, Section 6 presents the drawbacks of the method and future work, and the last sections present conclusions and references.

2. Representative Synthetic Benchmarks

We make a distinction between benchmark representativeness at a high *functional* level and representativeness at a low *execution* level. The most popular synthetic benchmarks were written in a high-level language to be representative of both the static high-level language features and dynamic low-level instruction frequencies of an application [WEIK95]. The fact that they were written at the same functional level as the original application had advantages: the code could be ported to multiple platforms, rewritten in different languages, and it would respond to compiler optimizations. None of these attributes, however, is relevant to the main purpose of the synthetic benchmark, which is to represent the machine response of the original workload. As soon as representative code is ported to another machine or language, or compiled with new compiler technology, even if the static high-level language characteristics are maintained, the code is most likely no longer representative of the low-level execution characteristics of the application undergoing the same transformation. A better outcome would be obtained by first transforming the application, executing it, then writing a new synthetic benchmark to represent the new workload characteristics of the application.

We propose that low-level, execution-based representativeness is a more useful focus for the development of synthetic benchmarks. The execution characteristics used in the generation of the synthetic benchmark can be built from a low-level workload characterization of the compiled and executing application. We make the following key observation: the statistical flow graphs in [OSKI00][EECK04][BELL04] are reduced representations of the control flow instructions of the application – a compact representative program. We combine the reduced representative trace from a detailed statistical simulation analysis with novel synthesis algorithms to automatically generate a simple but flexible benchmark in the C language, using *asm* statements to support low-level operations. Some flexibility is lost because the code targets a particular machine language, but the benchmark can easily be transformed for use on machines with similar ISAs.

We then benefit from the speed and flexibility of statistically generated traces while solving the shortcomings associated with the presence of traces, including portability to execution-driven simulators and hardware and simulation of multiprocessor locking mechanisms. At synthesis-time, parameters can be used to modify workload characteristics to study predicted trends of *future* workloads. At runtime, parameters can be used to switch between sections of code, consolidating multiple benchmark phases into a single benchmark.

The synthesis method can also be thought of as a *code abstraction* capability. Gone are questions of high-level programming style, language, or library routines that plagued the representativeness of the early synthetic benchmarks. Code abstraction also motivates increased code sharing between industry and academia, effectively hiding the functional meaning of proprietary code.

3. Automatic Benchmark Synthesis Approach

With reference to Figure 1, a code generator was built into a modified version of HLS [BELL04] that augments the data structures and information that already exist. After workload characterization analyzes the workload and produces a sequence of basic blocks that gives good simulation correlation as in [BELL04], the code generator takes the representative instructions and outputs a single module of C-code that contains calls to assembly-language instructions in the *pisa* language [BURG97]. Each instruction, including branches, maps one-to-one to a single assembly language call in the C-code.

The default synthesis mode is for perfect branch prediction. In that case, the basic blocks are synthesized into a single loop with both taken and not-taken targets of any branch configured to be the first instruction in the next sequential basic block.

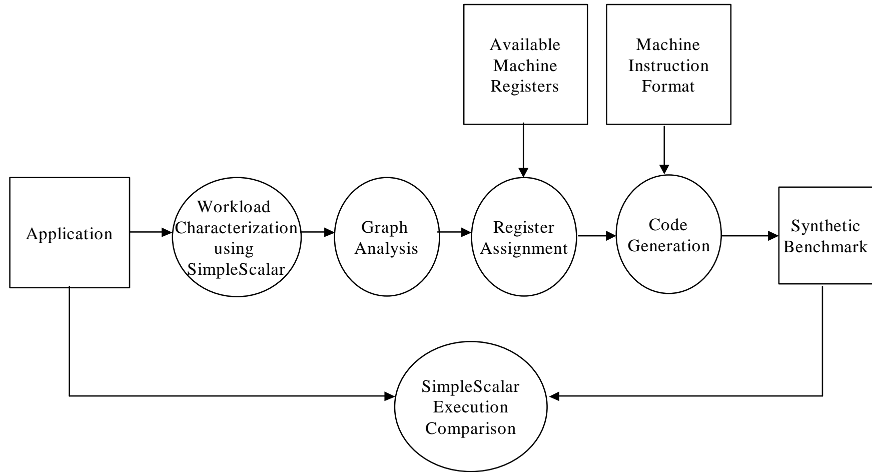

Figure 1: Benchmark Synthesis and Simulation Overview

We also synthesize with a simple branching model. In this scheme, we calculate the branches that will have taken-targets based on the global branch predictability, *BR*, of the original application. An integer instruction that is not used as a memory access counter or a loop counter is converted into an *invert* instruction operating on a particular register every time it is encountered. If the register is set, the branch jumps past the following basic block in the default loop. The *invert* mechanism causes a branch to have a predictability of 50% for most branch predictors, so the target *BR* must be equal to $(F*N + (1-F)*N*(0.5))/N$, where $(1-F)$ is the fraction of branches in the synthetic benchmark that are configured to use the *invert* mechanism and *N* is the total number of synthesized branches. Solving for $(1–F)$, we find the fraction of branches that must be configured to be $(2*BR – 1)$. We use a uniform random variable over this fraction to decide which branches are configured.

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

For both perfect branch prediction and the branching model, we use the configured I-cache size to implement the largest number of basic blocks such that the Icache miss rate for sequential code access is equivalent to the miss rate in the original workload. Synthesizing the Icache miss rate, *IMR*, depends on the cache configuration and the instruction size. An example *IMR* calculation is given in Section 4. The basic blocks and number of instructions synthesized for each benchmark are shown in Table 3.

The memory accesses for data are modeled using the 16 simple stream access classes shown in Table 1. The stride for a memory access is determined first by matching the L1 hit rate, after which the L2 hit rate for the stream is predetermined. The table was generated based on an L1 line size of 32 bytes, and an L2 line size of 64 bytes, and the stride is shown in 4 byte increments.

After the workload has been characterized, there are three major phases to the synthesis approach: *graph analysis*, *register assignment* and *code generation*. In the graph analysis phase, all instruction input dependencies are assigned. The starting dependence is exactly the dependent instruction chosen for input during statistical simulation. The issue then becomes operand *compatibility*: if the dependency is not compatible with the input type of the dependent instruction, then another instruction must be chosen. The algorithm is to move forward and backward from the starting dependency through the list of instructions in sequence order until the dependency is compatible. The average number of moves per instruction input is shown in Table 3 in column *dependency moves*. If more than a certain number of instructions are checked and a dependency cannot be made compatible, the program ends with an error. The exception is a store or branch that is operating on external data that was not generated in the program. An additional variable of the correct data type is created for this case.

Table 2 shows the compatibility of instructions in the *pisa* assembly language. The *dependent-inputs* column gives the *pisa* instruction inputs that are being tested for compatibility. For loads and stores, the memory access register must be an integer type. When found, it is labeled as a memory access counter for special processing during the code generation phase.

Table 2: Dependence Compatibility Chart

Dependent Instruction	Inputs	Dependence Compatiblity	Comment
Integer	0/1	Integer, Load-Integer	
Float	0/1	Float, Load-Float	
Load-Integer/Float	0	Integer	dep0 is addr resolution input
Store-Integer	0	Integer, Load-Integer	dep0 must be integer
Store-Float	0	Float, Load-Float	dep0 must be float
Store-Integer/Float	1	Integer	dep1 is addr resolution input
Branch-Integer	0/1	Integer, Load-Integer	
Branch-Float	0/1	Float, Load-Float	

When all instructions have compatible dependencies, a search is made for an additional integer instruction that is labeled as the loop counter and assigned a unique output register. 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, *loop iterations* in Table 3, is chosen to be large enough to assure IPC convergence. In general, this means that the number of loops must be larger than the longest memory access stream pattern of any memory operation. In practice, the number of loops does not have to be much larger than one hundred to characterize simple stream access patterns. When the branching model is enabled, an additional integer instruction is chosen to invert the branching register on each loop iteration and another register is reserved for it.

All register usages in the program are assigned exactly during the register assignment phase. The compiler requires that registers 1 to 5 and 28 to 31 of the 32 available registers be reserved, and register 0 is zero, so the rest of the 22 registers are split between memory access stream counters or code use. Memory access streams are *pooled* according to their stream access characteristics and a register is reserved for each class (*stream pools* in Table 3). For the benchmarks under study, the number of registers available for code use averages about 10 (*code registers* in Table 3). The minimum is 4, since there are only 16 possible stream classes and two registers are reserved for the loop counter and the branching register. Synthesis has the capability of reducing the number of stream pools and thus increasing the number of available registers for code use by combining the pools for the least frequently used streams, but in practice this procedure does not improve *quality* for the synthetic benchmarks in this study. 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.

In the code generation phase, the C-code main header is generated. Sequencing through all of the instructions, special variable declarations are generated to link registers to memory access variables, the loop variable, the branching variable, and pointers to the correct memory type for the memory access instructions. Then *malloc* calls for the memory access stream data are generated with size based on the number of iterations per program loop. Then we generate initializations for each memory access register to the head of the memory access stream data. The register assigned to a stream access will be incremented in the code after each access according to the stride appropriate for the stream class.

The loop counter register is initialized to the number of times the instructions will be executed. The instructions are then generated as calls to *pisa* assembly language instructions.

Each call is given an associated unique label. Memory access counters are generated using *addiu*, adding the stride to its variable value. The loop counter is generated as an *addi* with *-1* as the decrement value. Long latency floating point operations are generated using *mul.s* and short latency operations are generated using *add.s*. Loads and stores use *lw*, *sw*, *l.s* or *s.s* depending on the type. Branches use the *beq* type, and can have either integer or float operands. The basic blocks are analyzed and code is generated to print out unconnected output registers depending on a switch value. The switch is never set, so no code is eliminated during compilation. Code to free the *malloced* memory is generated, and finally a C-code footer is generated.

Figure 2 shows the synthetic version of the *saxpy* benchmark [MCCA95] used in this work. For comparison, Figures 3 and 4 show the original source code and disassembled loop after *gcc* compilation with optimization *–O*. *Saxpy* is relatively simple, but the same automatic process

```
int main(int argc, char* argv[]) { /* saxpy */
  int doprint;
  float* data_8; float* data_9;
  register int vout_8 asm ("8"); /* integer */
  float vout_22; float vout_24; float vout_26; float vout_28; float vout_30;
  register int vout_7 asm ("7"); /* branch invert cntr */
  register int vout_6 asm ("6"); /* loop cntr */
  register int vout_9 asm ("9"); /* integer */

  data_8 = (float*)malloc(553333 * sizeof(float));
  data_9 = (float*)malloc(553333 * sizeof(float));

  vout_8 = (int)&(data_8[0]);
  __asm__ __volatile__ ("add $8,%0,$0" : "=r" (vout_8) : "r" (vout_8));
  vout_9 = (int)&(data_9[0]);
  __asm__ __volatile__ ("add $9,%0,$0" : "=r" (vout_9) : "r" (vout_9));

  if(!strcmp(argv[0], "print")) doprint = 1; else doprint = 0;

  vout_7 = 0; __asm__ __volatile__ ("add $7,%0,$0" : "=r" (vout_7) : "r" (vout_7));
  vout_6 = 33334; __asm__ __volatile__ ("add $6,%0,$0" : "=r" (vout_6) : "r" (vout_6));

  instr0: /* Index 0 */ __asm__ __volatile__ ("addiu %0,%0,4" : "=r" (vout_8) : "r" (vout_8));
  instr1: __asm__ __volatile__ ("l.s $f2,0(%1)" : "=f" (vout_24), "=r" (vout_8) : "f" (vout_24), "r" (vout_8));
  instr2: __asm__ __volatile__ ("mul.s $f4,$f0,$f2"
    : "=f" (vout_26), "=f" (vout_22), "=f" (vout_24)
    : "f" (vout_26), "f" (vout_22), "f" (vout_24));
  instr3: __asm__ __volatile__ ("l.s $f6,0(%1)" : "=f" (vout_28), "=r" (vout_9) : "f" (vout_28), "r" (vout_9));
  instr4: __asm__ __volatile__ ("add.s $f8,$f4,$f6"
    : "=f" (vout_30), "=f" (vout_26), "=f" (vout_28)
    : "f" (vout_30), "f" (vout_26), "f" (vout_28));
  instr5: __asm__ __volatile__ ("nor $7,$7,$0" : "=r" (vout_7) : "r" (vout_7));
  instr6: __asm__ __volatile__ ("addi $6,$6,-1" : "=r" (vout_6) : "r" (vout_6));
  instr7: __asm__ __volatile__ ("s.s $f8,0(%1)" : "=f" (vout_30), "=r" (vout_9) : "f" (vout_30), "r" (vout_9));
  instr8: __asm__ __volatile__ ("addiu %0,%0,4" : "=r" (vout_9) : "r" (vout_9));
  instr9: if (vout_6 > 0) goto instr0;

  if(doprint) {
    printf("vout_%d %d\n", 9, vout_9); printf("vout_%d %d\n", 6, vout_6);
  }
  free(data_8); free(data_9);
}
```

Figure 2: S*AXPY* Synthetic Benchmark

```
int main() {
#define LIM 1000000
int k; float q, z[LIM], z[LIM];
   q = 3.0;
   for (k = 0; k < LIM; k++)
      z[k] = z[k] + q*x[k];
   printf("%f", q);
}
```

Figure 3: S*AXPY* Source Code

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

Figure 4 : Disassembled *SAXPY* Loop

analyzes and synthesizes code for the complex SPEC95 benchmarks.

Table 3 gives the synthesis information described in this section for each of the benchmarks. The *ratio* is the runtime of the original benchmark for one billion instructions divided by the runtime of the synthetic benchmark. The *loop iterations* has not been tuned for the benchmarks, i.e. *loop iterations* is approximately *400K/(number of instructions)*, so the *ratio* can probably be increased without affecting representativeness by decreasing *loop iterations*. This is definitely the case for the STREAM benchmarks. Future work will seek to tune this variable.

Table 3: Synthetic Benchmark Properties

Name	Number of Basic Blocks	Number of Instructions	Stream Pools	Code Registers	Loop Iterations	Dependency Moves	Actual Runtime (s)	Synthetic Runtime (s)	Ratio
gcc	677	2481	6	8	111	0.661	5777.09	1.38	4186.30
perl	438	2274	5	6	93	1.83	5754.99	1.07	5378.50
m88ksim	422	2174	6	6	143	0.299	5076.3	1.41	3600.21
ijpeg	368	1924	5	6	175	0.208	2644.69	1.47	1799.11
vortex	468	2512	6	6	76	0.911	6382.39	1.1	5802.17
compress	421	2130	4	8	501	0.42	4746.88	4.35	1091.24
go	477	2360	9	6	162	0.217	5256.64	1.76	2986.73
li	391	1650	11	8	147	0.935	902.78	1.38	654.19
tomcatv	650	2549	5	8	159	1.644	5785.63	1.99	2907.35
su2cor	633	2537	4	6	130	0.83	5537.71	1.61	3439.57
hydro2d	322	2201	9	6	192	0.49	12823.84	2.27	5649.27
mgrid	21	2031	8	10	131	1.025	5679.83	1.34	4238.68
applu	104	2151	9	8	215	0.218	6719.31	1.91	3517.96
turb3d	107	2088	9	8	234	0.217	4950.21	1.91	2591.73
apsi	97	2152	13	6	164	0.826	6611.47	1.93	3425.63
wave5	376	2184	8	6	223	0.91	5403.51	2.42	2232.86
fpppp	52	2862	6	6	65	1.203	7176.41	1.17	6133.68
swim	53	1213	9	6	276	0.321	6037.54	1.49	4052.04
saxpy	1	10	2	12	33334	0	124.41	2.09	59.53
sdot	1	9	2	12	50001	0	546.07	2.55	214.15
sfill	1	5	1	12	100001	0	21.22	1.72	12.34
scopy	1	7	2	12	50001	0	43.98	1.89	23.27
ssum2	1	6	1	12	100001	0	22.2	2.99	7.42
sscale	1	8	2	12	50001	0	91.05	2.24	40.65
striad	1	11	3	12	33334	0	65.59	2.12	30.94
ssum1	1	10	3	12	33334	0	36.61	1.85	19.79

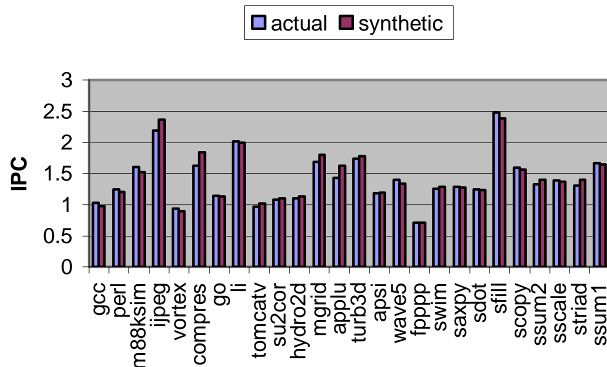

Figure 5: Actual vs. Synthetic Benchmark IPC Perfect Branch Prediction

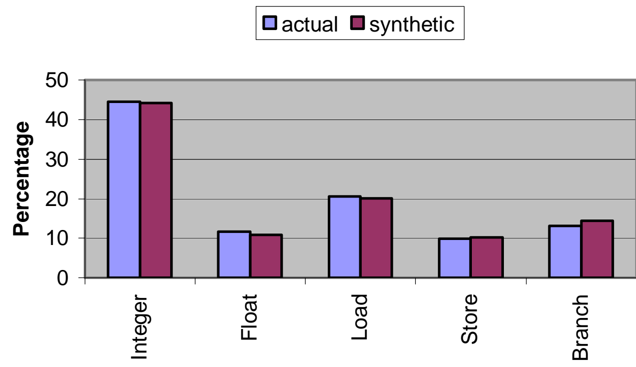

Figure 6: Actual vs. Synthetic Benchmark Average Instruction Percentage Perfect Branch Prediction

4. Benchmark Synthesis Results

In this section we present the benchmark synthesis experimental results, first with perfect branch prediction and later using a synthetic branching function.

4.1. Experimental Setup and Benchmarks

The modified HLS system used in this study was presented in [BELL04]. We extend their system with the benchmark synthesis capability described here. SimpleScalar Release 2.0 [BURG97] was downloaded as well as the SPEC95 *pisa* binaries found at [SOHI03]. The applications were executed with the default SimpleScalar configuration in *sim-outorder* on the first reference dataset for up to one billion instructions. In addition, single-precision versions of the STREAM and STREAM2 benchmarks [MCCA95] with a one million-loop limit were compiled using the SimpleScalar *pisa* cross-compiler and executed.

Code generation was enable and C-code was produced using the synthesis method of Section 3. The synthetic benchmarks were cross-compiled to the *pisa* language [BURG97] using *gcc* with optimization level *-O* and executed to completion in SimpleScalar.

4.2. Synthesis Results using Perfect Branch Prediction

The following figures show results for both the original applications, *actual*, and the synthetic benchmarks, *synthetic*. Figure 5 shows the IPC for the benchmarks. The average error for the synthetic benchmarks is 3.9%, with a maximum error of 13.2% for *compress*. We discuss the reasons for the errors in the context of the figures below.

Figure 6 compares the average instruction percentages over all benchmarks for each class of instructions. The average error is 4.2%. Figure 7 shows that the basic block size varies per benchmark with an average error of 8.8%. The errors are caused by variations in the fractions of specific basic block types in the synthetic benchmark with respect to the original workload. This is a direct consequence of selecting a limited number of basic blocks during synthesis.

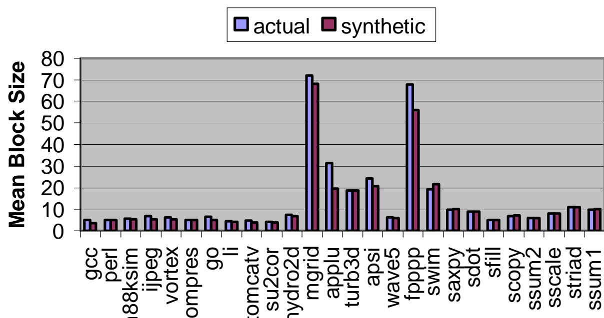

Figure 7: Actual vs. Synthetic Basic Block Size
Perfect Branch Prediction

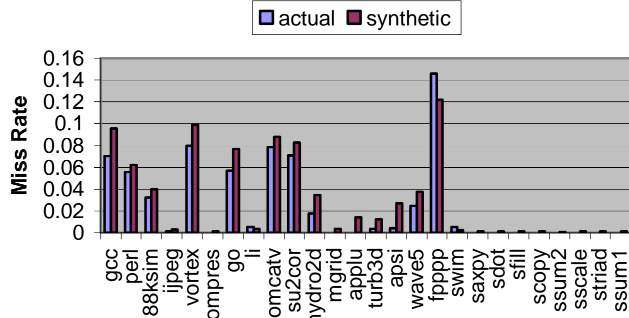

Figure 8: Actual vs. Synthetic IL1 Miss Rates
Perfect Branch Prediction

The number of synthetic basic blocks is determined by the I-cache configuration. In our experiments, the I-cache is a 16KB direct mapped cache with 32B lines, giving 512 lines. In the *pisa* language, each instruction is 8 bytes, so the I-cache can contain 2048 instructions without missing. Since there are four instructions per cache block, there will be two misses per loop for every four instructions over 2048 in the workload. To generate a specific benchmark miss rate, *IMR*, the synthetic benchmark must therefore be composed of 2048 instructions plus an additional *(2*2048*IMR)/(1-2*IMR)* instructions. Using this calculation with *IMR* set to the miss rate of the original benchmark, the instruction counts for the synthetic benchmarks are within 2% on average of the expected instruction counts. The miss rates are shown in Figure 8.

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 2048 is 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 *IMR* and instruction mix requirements.

One consequence of synthesizing a small number of basic blocks to meet an *IMR* is shown in Figure 9. The synthetic benchmarks are generated from runs with perfect branch prediction, but

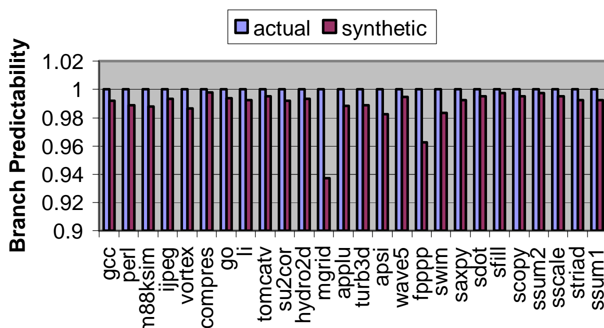

Figure 9: Actual vs. Synthetic
Branch Predictability
Perfect Branch Prediction

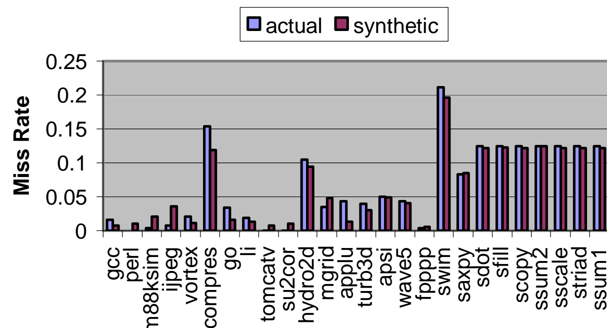

Figure 10: Actual vs. Synthetic
DL1 Miss Rates
Perfect Branch Prediction

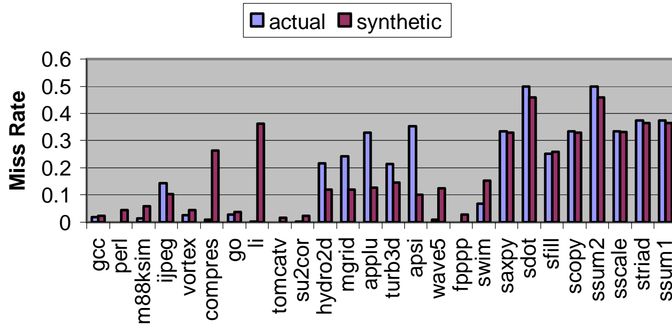

Figure 11: Actual vs. Synthetic UL2 Miss Rates Perfect Branch Prediction

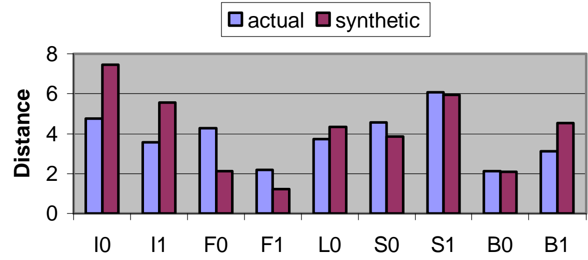

Figure 12: Average Dependency Distance for Instruction Inputs Perfect Branch Prediction

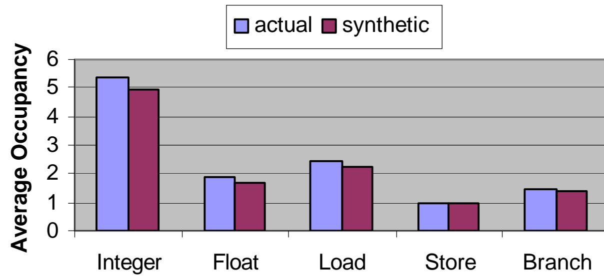

Figure 13: Average RUU Occupancy per Cycle by Type Perfect Branch Prediction

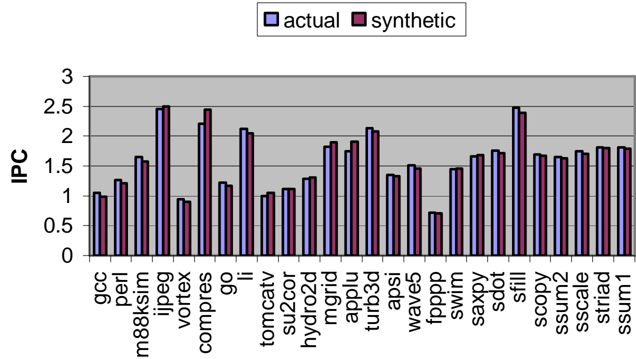

Figure 14: Actual vs. Synthetic IPC Dispatch Window 32 Perfect Branch Prediction

they are executed with a real branch predictor in order to assess the capability of synthesizing a benchmark with perfect predictability when running on hardware. The branch predictability of the synthetic benchmarks can vary depending on the interaction of their few branches with a real branch predictor. The lower predictabilities for *mgrid* through *swim* (except for *wave5*) are due to the relatively larger basic block sizes for those benchmarks as shown in Figure 7.

The L1 data cache miss rates are shown in Figure 10. In spite of using a very simple cache access model with only 16 different possible access patterns, the correlation with the original workloads is quite similar. For miss rates greater than 0.05, the trends using the synthetic benchmarks clearly correspond with those of the original workloads. Again, there is some variation for smaller miss rates, but the execution impact is also small.

In Figure 11, the unified L2 miss rates are shown. The large errors due to the simple memory access model are often mitigated by small L1 miss rates. A good example is *li*. Exceptions include *compress*; but *compress* has the highest fraction of integer instructions among the benchmarks, and its high L2 miss rate is offset by the relatively long integer dependency distances in the synthetic benchmarks.

Figure 12 shows the effect. The increased dependency distances are due to the conversion of many integer instructions to memory access stride counters. A stride counter overrides the original function of the integer instruction and causes dependency relationships to change. Another source of error is the movement of dependencies during the search for compatible dependencies in the synthesis process. The average movement is less than one position, as shown in the *dependency moves* column of Table 3.

In spite of the dependency distance errors, Figure 13 shows that the average register update unit (RUU) occupancies are similar to those of the original benchmarks with an average error of 6.3%.

4.3. Using Synthetic Benchmarks to Assess Design Changes

We now study design changes using the same synthetic benchmarks. Figures 14 and 15 show the absolute IPCs using an RUU size of 32 and 64 with average errors of 3.2% and 3.1%, respectively. Figure 16 graphs the IPC prediction errors for each benchmark. Most errors are below 5%.

Figures 17 and 18 show the absolute change in IPC, *delta IPC*, as the same benchmarks are executed first with the default configuration (RUU size of 16) and then with the RUU sizes changed to 32 and 64 respectively. The average relative errors [EECK04] are 2.1% and 2.7%, respectively. The graphs show that, when an application change is large with respect to the changes in the other applications, the synthetic benchmark change is also large relative to the change in the other synthetic benchmarks. These IPC changes would be large enough to trigger additional studies using a detailed cycle-accurate simulator. Chip designers are looking for cases in a large design space in which a design change may improve or worsen a design. In the case of the RUU studies, the results would trigger further cycle-accurate studies of *ijpeg*, *compress*, and

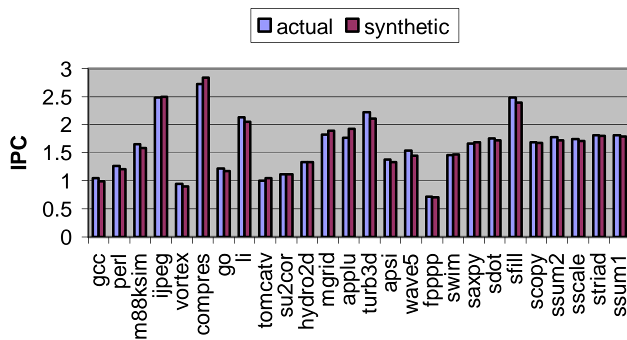

Figure 15: Actual vs. Synthetic IPC for Dispatch Window 64 Perfect Branch Prediction

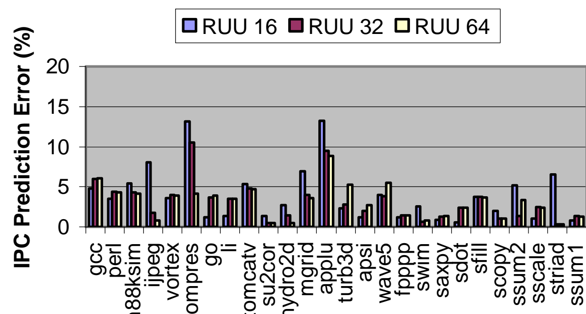

Figure 16: Actual vs. Synthetic IPC Prediction Error for Dispatch Window 16, 32 and 64 Perfect Branch Prediction

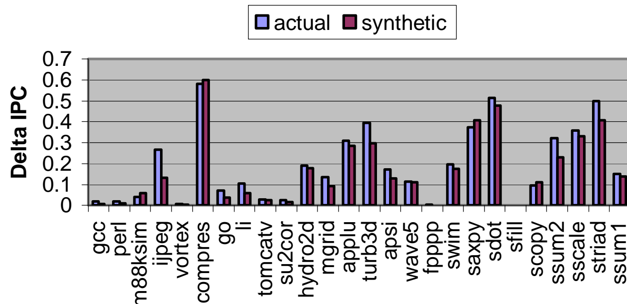

Figure 17: Actual vs. Synthetic Delta IPC Dispatch Window Increased from 16 to 32 Perfect Branch Prediction

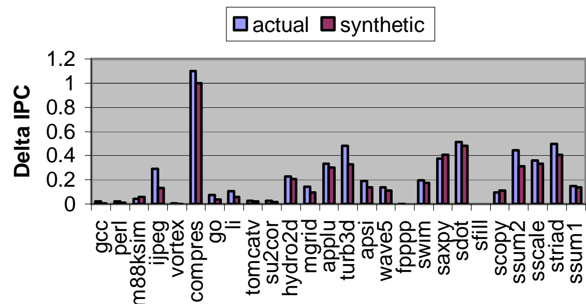

Figure 18: Actual vs. Synthetic Delta IPC Dispatch Window Increased from 16 to 64 Perfect Branch Prediction

several of the SPECfp applications. Alternatively, the designers might be curious why the change did not help a benchmark like *gcc*, resulting in an additional study.

Figure 19 shows the delta IPC error as the L1 D-cache latency is decreased from 8 to 1. The average absolute IPC error is 8.9% and the delta IPC relative error is 6.9%. Figure 20 shows the delta IPC as the issue width increases from 1 to 4. The average absolute error is 2.6%, and the relative error is 4.1%. Similar studies for a commit width change from 4 to 8 give an absolute error of 3.7% and a relative error of 1.1%. Doubling the L1 D-cache configuration gives an absolute error of 4.3% and a relative error of 2.1%. Doubling the L1 I-cache configuration gives an absolute error of 9.8% and a relative error of 8.2%. Again, all of these runs use the same benchmark synthesized from the default SimpleScalar configuration, not a resynthesized benchmark.

4.4. Synthesis Results using the Simple Branch Predictor Model

Additional studies were carried out using the simple branch predictor model described in Section 3. Figure 21 shows the absolute IPC error for the default SimpleScalar configuration. The average error is 2.4%. Because only the branch predictability changes, the I-cache and D-cache miss rates are the same as before. The average instruction mix error increases from 4.2% to 4.8%

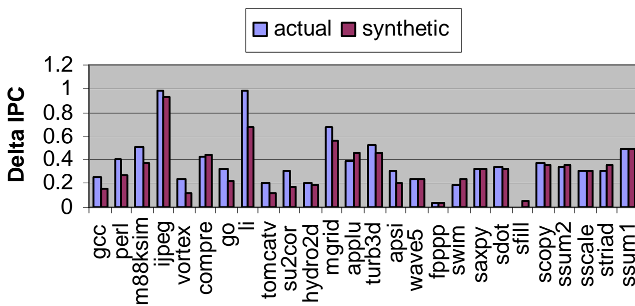

Figure 19: Actual vs. Synthetic Delta IPC as DL1 Latency Decreases from 8 to 1 Perfect Branch Prediction

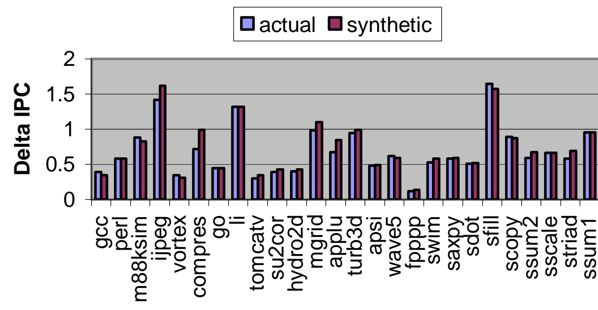

Figure 20: Base vs. Synthetic Delta IPC as Issue Width Increases from 1 to 4 Perfect Branch Prediction

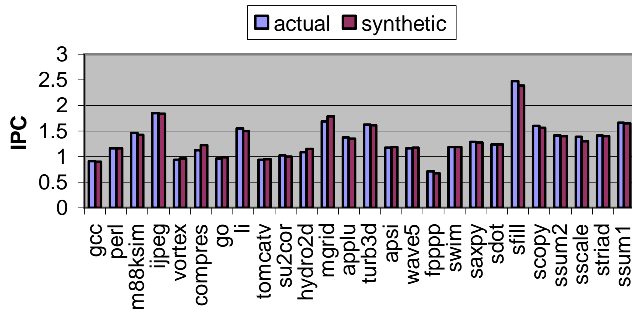

Figure 21: Actual vs. Synthetic IPC Branch Prediction Function

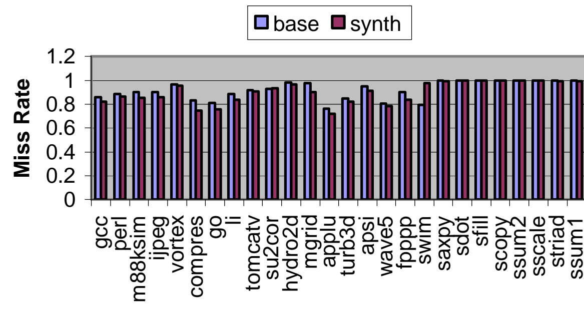

Figure 22: Actual vs. Synthetic Overall Branch Predictability Branch Prediction Function

due to branching around basic blocks in the branch predictability model. For the same reason, the average RUU occupancy error increases from 6.3% to 9.2%. The graph of dynamic dependency distances per input changed only slightly.

Figure 22 shows the branch predictability results using the simple model. The model does well, with an average absolute error of 3.9%.

As design elements are changed, the IPC prediction errors are low, similar to the perfect branch predictability case, and in some cases lower. Tables 4 and 5 summarize the results for both synthetic benchmark models and include the delta IPC relative errors versus the base case (default SimpleScalar configuration). The errors are similar but not identical to those found with perfect branch prediction. As an example, Figure 23 shows the delta IPC as the RUU size is changed from 16 to 32.

Table 4: Average Synthetic IPC Errors and Delta IPC Errors vs. Actual Applications

Model	Commit Width 8		Commit Width 1		L1 D-cache 256:64:8		L1 I-cache 1024:64:2	
	IPC	Delta IPC	IPC	Delta IPC	IPC	Delta IPC	IPC	Delta IPC
Perfect Branch Prediction	3.7%	1.1%	2.8%	4.2%	4.3%	2.1%	9.8%	8.2%
Branch Prediction Function	2.6%	1.4%	3.2%	3.9%	3.2%	2.4%	8.7%	7.5%

Table 5: Average Synthetic IPC Errors and Delta IPC Errors vs. Actual Applications

Model	RUU					DL1 Latency 8		Issue Width 1	
	16	32		64					
	IPC	IPC	Delta IPC	IPC	Delta IPC	IPC	Delta IPC	IPC	Delta IPC
Perfect Branch Prediction	3.9%	3.2%	2.1%	3.1%	2.7%	8.9%	6.9%	2.6%	4.1%
Branch Prediction Function	2.4%	3.1%	2.2%	3.3%	2.4%	11.1%	10.4%	2.1%	2.2%

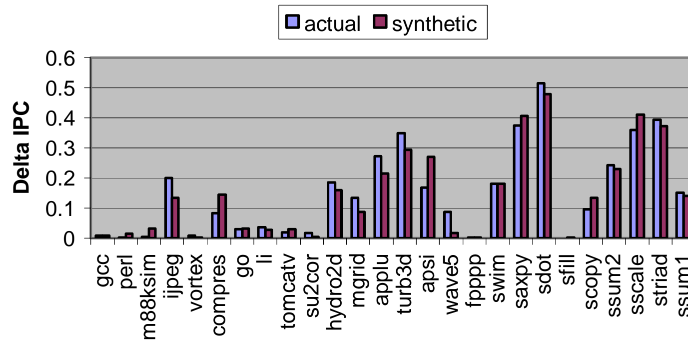

Figure 23: Actual vs. Synthetic Delta IPC as Dispatch Window Increased from 16 to 32 Branch Prediction Function

5. Related Work

Several ad-hoc techniques to synthesize workloads have been developed [WONG88] [SREE74] [WILL76]. In [WONG88], a linear combination of microbenchmarks is found that, when combined in a process called *replication* and executed, duplicates the LRU hit function of the target benchmark. There is no clear way to incorporate other execution characteristics like instruction mix into the technique.

In [HSIE98], 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 [EECK03]. In particular, the instruction sequences and dependency relationships of the synthetic programs are not representative of the original workload, unlike in the present work. The cache access and branch predictor models in [HSIE98] are useful as high-level ideas or starting points, but the specific implementations in that work allow and rely on modifications to the workload features shown to be required for representative performance.

The primary focus in *program synthesis* is on mathematical theorem-provers or frameworks for synthesizing high-performance programs from formal specifications [MANN80][BAUM02]. Workload characteristics of existing programs are not considered.

The well-organized microarchitectures of ASICs and DSPs lend themselves to automatic *code scheduling* from simple code specifications [CHEN94]. The goal is to schedule instructions to maximize performance with complete knowledge of the machine resources and pipeline structure, and no attempt is made to generate programs based on the workload characteristics of other programs.

6. Drawbacks of the Approach and Future Work

The main drawback of the approach is that the microarchitecture independent workload characteristics, and thus the synthetic workload, are dependent on the particular compiler technology used. As discussed earlier, this is a natural result of requiring the execution characteristics of the synthetic benchmark to be representative of those of the original application. However, if the synthetic benchmark targets an ISA similar to the ISA used for the compilation, the *asm* statements can be easily substituted for the target machine. As instruction sets continue the current trend towards RISC-like primitive operations [HENN96], retargeting for another ISA becomes less of an issue.

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 [EECK04][BELL04]. In the future, synthesis parameters could be used to incorporate or not incorporate features as needed.

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. We argue that this is a requirement for representativeness, and 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.

Our cache access and branch predictor models are simplistic. Models with less impact on dependency distances need to be developed. More research is needed to make use of advanced models [SORE02][THIE89], but those that have been developed to date may not model all access streams well. The benchmark synthesis approach presented in this paper 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 *IMR*. The small number causes small but noticeable variations in workload characteristics, including basic block size, with corresponding changes in instruction mix, dependency relationships, and RUU occupancies. One solution is to instantiate additional basic blocks using *replication* and *repetition* [WONG89] [DJUM04]. Multiple sections of representative synthetic code could be synthesized and concatenated together into a single benchmark. Each section would satisfy the *IMR*, 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 [SHER02]. Similarly, phases from multiple benchmarks could be consolidated together.

7. Conclusions

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

Unlike prior benchmark synthesis efforts, we focus on the low-level workload characteristics of the compiled and executing binary to create workloads that are truly representative of the effects of the application in the machine. Multiple synthetic benchmarks are necessary if the application is executed on multiple machines or significantly different ISAs, but we argue that representativeness cannot be ensured otherwise. 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, future workload generation, and code abstraction.

We use the method to synthesize representative benchmarks for the SPEC95 and STREAM benchmarks with both perfect branching and a simple branching model. We find that benchmarks can be synthesized to an average IPC within 3.9% of the average IPC of the target applications with remarkably similar instruction mix, cache access characteristics, RUU occupancies, and dependency characteristics, while runtimes are often three orders of magnitude shorter. In addition, the change in IPC for a synthetic benchmark due to a design change is found to be proportional to the IPC change for the original application and relative errors are small. The resulting synthetic benchmarks are flexible and can be parameterized at synthesis-time and run-time.

References

[BAUM02] G. Baumgartner, et al., "A High-Level Approach to Synthesis of High Performance Codes for Quantum Chemistry," Proceedings of the ACM/IEEE Conference on Supercomputing, Nov. 2002.

[BELL04] 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, June20, 2004.

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

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

[CHEN94] W. K. Cheng and Y. L. Lin, "Code Generation for a DSP Processor, " Proceedings of

the Seventh International Symposium on High Level Synthesis," May 1994, pp. 82-87.

[CURN76] H. J. Curnow and B.A. Wichman, "A Synthetic Benchmark," Computer Journal, vol. 19, No. 1, February 1976, pp, 43-49.

[DUJM98] J. J. Dujmovic and I. Dujmovic, "Evolution and Evaluation of SPEC Benchmarks," ACM Sigmetrics Performance Evaluation Review, Vol. 26, Issue 3, Dec. 1998, pp. 2-9.

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

[EECK04] L. Eeckhout, R. H. Bell, Jr., B. Stougie, L. K. John and K. De Bosschere, "Improved Statistical Simulation for Power/Performance Modeling," International Symposium on Computer Architecture, June 2004, to appear.

[HENN96] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 2nd Edition, Morgan Kauffman, 1996.

[HSIE98] 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.

[KLEI02] A.J. KleinOsowski and D.J. Lilja, "MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research," Computer Architecture Letters, June 2002.

[LILJ00] D. J. Lilja, Measuring Computer Performance, Cambridge University Press, 2000.

[MANN80] Z. Manna and R. Waldinger, "A Deductive Approach to Program Synthesis," ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 2 Issue 1, January 1980.

[MCMA86] F.H. McMahon, "Livermore FORTRAN Kernels: A Computer Test of the Numerical Performance Range," Lawrence Livermore National Laboratories, Livermore, CA, 1986.

[MCCA95] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," IEEE Technical Committee on Computer Architecture newsletter, December 1995.

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

[OSKI00] M. Oskin, F. T. Chong and M. Farrens, "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Design," Proceedings of the 27th Annual International Symposium on Computer Architecture, June 2000, pp. 71-82.

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

[SHER02] T. Sherwood, E. Perleman, H. Hmaerly 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.

[SKAD03] 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.

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

[SORE02] E. S. Sorenson and J. K. Flanagan, "Evaluating Synthetic Trace Models Using Locality Surfaces," In Proceedings of the IEEE International Workshop on Workload Characterization," Nov. 2002, pp. 23-33.

[SREE74] K. Sreenivasan and A.J. Kleinman, "On the Construction of a Representative Synthetic Workload," Communications of the ACM, March 1974, pp.127-133.

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

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

[THIE89] 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.

[TODI01] R. Todi, "SPEClite: Using Representative Samples to Reduce SPEC CPU2000 Workload," IEEE Workshop on Workload Characterization, December 2001, pp. 15-23.

[VAND04] H. Vandierendonck and K. De Bosschere, "Many Benchmarks Stress the Same Bottlenecks," IEEE CAECW 2004.

[WEIK84] R. P. Weiker, "Dhrystone: A Synthetic Systems Programming Benchmark," Communications of the ACM, October 1984, pp. 1013-1030.

[WEIK95] R. P. Weicker, "An Overview of Common Benchmarks," IEEE Computer, December 1995, pp. 65-75.

[WILL76] 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.

[WONG88] 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 1998, pp. 637-645.

[WUND02] 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.