Distilling the Essence of Proprietary Workloads into Miniature Benchmarks

AJAY JOSHI
University of Texas at Austin
LIEVEN EECKHOUT
Ghent University
ROBERT H. BELL JR.
IBM, Austin
and
LIZY K. JOHN
University of Texas at Austin

10

Benchmarks set standards for innovation in computer architecture research and industry product development. Consequently, it is of paramount importance that these workloads are representative of real-world applications. However, composing such representative workloads

A preliminary version of this paper entitled "Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks," by A. Joshi, L. Eeckhout, R. Bell Jr., and L. K. John appeared in the *IEEE International Symposium on Workload Characterization*, 2006 [Joshi et al. 2006a]. © IEEE 2006. This extended version makes the following new contributions over the previous paper: (1) It applies the application performance cloning technique to general-purpose and scientific programs, next to the embedded workloads considered in the original paper; (2) It characterizes the data locality of general-purpose and scientific benchmarks and shows that the memory access patterns of these programs can be modeled using a stride-based model; and (3) It studies the convergence properties of the synthetic benchmark clone and applies it to generate miniature synthetic clones that simulate five orders of magnitude faster than the proprietary application. The ability to generate miniature benchmarks is an improvement over the previous paper in which the synthetic clones had the same order of dynamic instruction count as the original application and, therefore, did not reduce simulation time.
Ajay Joshi was supported by an IBM Fellowship. Lieven Eeckhout is a Postdoctoral Fellow with the Fund for Scientific Research – Flanders, Belgium. This work is also supported in part through the NSF award numbers 0429806 and 0702694, an IBM Faculty Partnership Award, and the UGent-BOF project 01J14407. The findings presented in this paper are opinions of the authors and not of the National Science Foundation or other funding agencies.
Authors' addresses: Ajay Joshi and Lizy K. John, University of Texas at Austin, Austin, Texas 78712; email: ajoshi@ece.utexas.edu. Lieven Eeckhout, Ghent University, Belgium. Robert H. Bell, IBM, Austin, Texas 78712.

poses practical challenges to application analysis teams and benchmark developers (1) real-world workloads are intellectual property and vendors hesitate to share these proprietary applications; and (2) porting and reducing these applications to benchmarks that can be simulated in a tractable amount of time is a nontrivial task. In this paper, we address this problem by proposing a technique that automatically distills key inherent behavioral attributes of a proprietary workload and captures them into a miniature synthetic benchmark clone. The advantage of the benchmark clone is that it hides the functional meaning of the code but exhibits similar performance characteristics as the target application. Moreover, the dynamic instruction count of the synthetic benchmark clone is substantially shorter than the proprietary application, greatly reducing overall simulation time for SPEC CPU, the simulation time reduction is over five orders of magnitude compared to entire benchmark execution. Using a set of benchmarks representative of general-purpose, scientific, and embedded applications, we demonstrate that the power and performance characteristics of the synthetic benchmark clone correlate well with those of the original application across a wide range of microarchitecture configurations.

Categories and Subject Descriptors: B.8 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.4 [**Performance of Systems**]: Measurement Techniques, Modeling Techniques

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Benchmarks, workload characterization, benchmark cloning

ACM Reference Format:

Joshi, A., Eeckhout, L., Bell Jr., R. H., and John, L. K. 2008. Distilling the essence of proprietary workloads into miniature benchmarks. ACM. Trans. Architec. Code Optim. 5, 2, Article 10 (August 2008), 33 pages. DOI = 10.1145/1400112.1400115 http://doi.acm.org/10.1145/1400112.1400115

1. INTRODUCTION

The use of benchmarks for quantitatively evaluating novel ideas, analyzing design alternatives, and identifying performance bottlenecks has become the mainstay in computer systems research and development. A wide range of programs, ranging from microbenchmarks, kernels, hand-coded synthetic benchmarks, to full-blown real-world applications, have been used for the performance evaluation of computer architectures. Early synthetic benchmarks, such as Whetstone [Curnow and Wichman 1976] and Dhrystone [Weiker 1984], had the advantage of consolidating multiple application behaviors into one program. However, these benchmarks fell out of favor in the 1980s, because they were hand-coded and hence, difficult to upgrade and maintain, and were easily subject to unfair optimizations. Smaller benchmark programs, such as microbenchmarks and kernel codes, have the advantage that they are relatively easier to develop, maintain, and use. However, they only reflect the performance of a very narrow set of applications and may not serve as a general benchmark against which the performance of real-world applications can be judged. At the other end of the spectrum, the use of real-world applications as benchmarks offers several advantages to architects, researchers, and customers. They increase the confidence of architects and researchers in making design trade-offs and make it possible to customize microprocessor design to specific (sets of) applications. Also, the use of real-world applications for benchmarking greatly simplifies purchasing decisions for customers.

Although the proposition of using real-world applications for benchmarking seems very attractive, there are several challenges. Real-world applications tend to be intellectual property and application developers hesitate to share them with third-party vendors and researchers. Moreover, enormous time and effort is required for engineering real-world applications into portable benchmarks. The problem is further aggravated by the fact that real-world applications are diverse and evolve at a rapid pace. This makes it necessary to upgrade and maintain special benchmark versions very frequently. Another challenge in engineering benchmarks from real-world applications is to make them simulation friendly—a very large dynamic instruction count results in intractable simulation times even on today's fastest simulators running on today's fastest machines.

The objective of this paper is to explore automated synthetic benchmark generation as an approach to disseminate real-world applications as miniature benchmarks without compromising on the applications' proprietary nature. Moreover, automated synthetic benchmark generation significantly reduces the effort of developing benchmarks, making it possible to upgrade the benchmarks more often. In order to achieve this, we propose *benchmark cloning*, a technique that distills key behavioral characteristics of a real-world application and models them into a synthetic benchmark. The advantage of the synthetic benchmark clone is that it provides code abstraction capability, i.e., it hides the functional meaning of the code in the original application, but exhibits similar performance characteristics as the real application. Source code abstraction, prevents reverse engineering of proprietary code, which enables software developers to share synthetic benchmarks with third parties. Moreover, the dynamic instruction count of the synthetic benchmark clone is much smaller than the original application and significantly reduces its simulation time. These two key features of synthetic benchmark clones, source code abstraction, and a small dynamic instruction count, enable the dissemination of a proprietary real-world application as a miniature synthetic benchmark that can be used by architects and designers as a proxy for the original application. Computer architecture researchers have also recognized the importance of synthetic benchmarks and have expressed the need to develop scientific approaches to construct such benchmarks [Skadron et al. 2003].

The key novelty in our benchmark cloning approach compared to prior work in synthetic benchmark generation [Bell and John 2005] is that we characterize the performance of codes using inherent workload characteristics that are independent of the microarchitecture. Since the design of the synthetic benchmark is guided only by the application characteristics and is independent of any hardware specific features, the benchmark can be used across a wide range of microarchitectures. We show in our evaluation that the synthetic benchmark clone shows good correlation with the original application across a wide range of cache, branch predictor, and other microarchitecture configurations.

The remainder of this paper is organized as follows. In Section 2, we provide a high-level overview of the benchmark cloning approach. Section 3 then provides more details and describes the application characterization method and the algorithm used to generate the synthetic benchmark clone. In Section 4, we

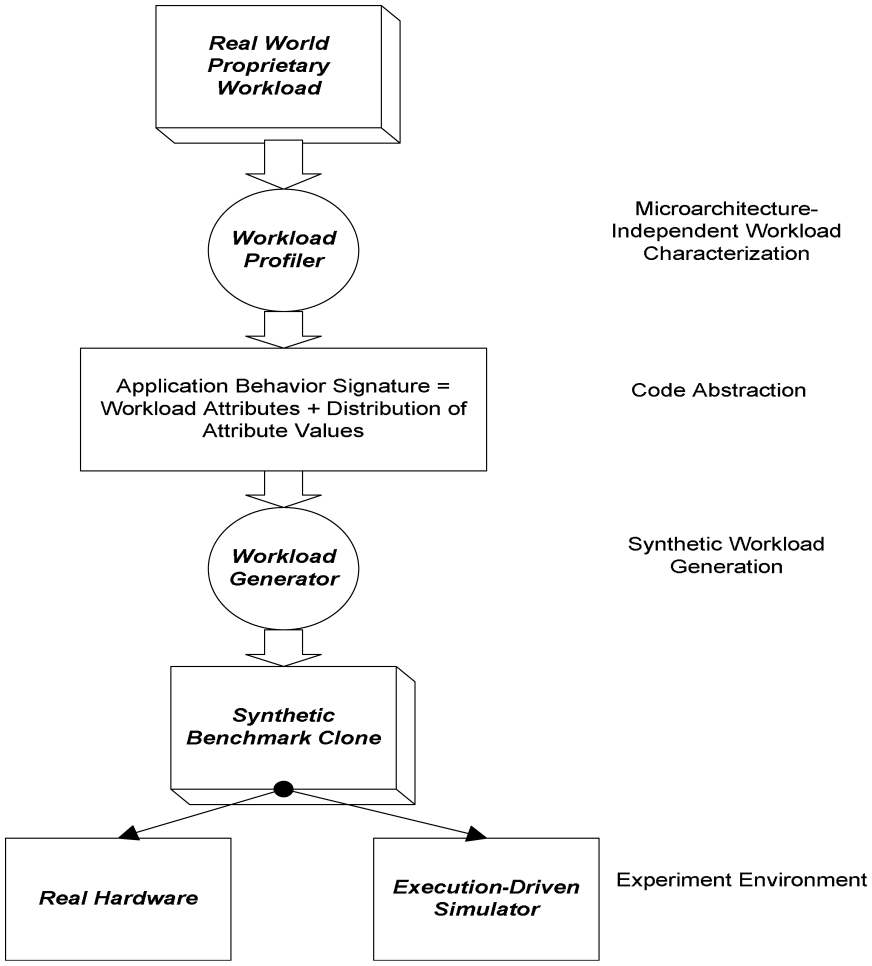


Fig. 1. Framework for constructing a synthetic benchmark clone from a real-world application.

describe our simulation environment, machine configuration, and the benchmarks that we use to evaluate the efficacy of synthetic benchmark clones for performance and power estimation. Section 5 presents a detailed evaluation of the synthetic benchmark cloning technique using various criteria. In Section 6, we provide a discussion on the strengths and limitations of the benchmark cloning technique. We summarize related research work in Section 7. Finally, in Section 8, we summarize the key results from this paper.

2. BENCHMARK CLONING

Figure 1 illustrates the benchmark cloning approach that we propose in this paper for generating synthetic benchmarks that abstract the functional meaning of a real-world application while mimicking its behavioral and performance characteristics. Benchmark cloning comprises of two steps: (1) Profiling the

real-world proprietary workload to measure its inherent behavior characteristics, and (2) modeling the measured workload attributes into a synthetic benchmark program.

The benchmark cloning approach is based on the premise that the behavior of programs can be characterized using a small set of inherent workload characteristics. This requires that we develop a parametric characterization of the behavior of programs to understand the set of inherent characteristics that correlate well with the performance exhibited by the program. The set of workload characteristics can be thought of as an *application behavior signature* that describes the workload's inherent behavior, independent of the microarchitecture. This increases the usefulness of the synthetic benchmark clone during computer architecture research and development as it serves as a useful indicator of performance even when the underlying microarchitecture is altered.

The next step in the benchmark cloning technique is to generate a synthetic benchmark that embodies the application behavior signature of the proprietary workload. Ideally, if all the key workload attributes of the real application are successfully replicated into the performance clone, the synthetic benchmark should exhibit similar performance characteristics.

3. BENCHMARK CLONING FRAMEWORK

We now describe in more detail the workload attributes that capture the inherent behavior signature of an application in Section 3.1. In Section 3.2, we provide details on the algorithm that models these characteristics into a synthetic benchmark clone.

3.1 Application Behavior Signature

We characterize the proprietary application by measuring its inherent microarchitecture-independent workload characteristics. The characteristics included in the application behavior signature are a subset of all the microarchitecture-independent characteristics that can be potentially modeled, but we believe that we model (most of) the important program characteristics that impact a program's performance; the results in the evaluation section, in fact, show that this is the case for the general-purpose, scientific, and embedded applications that we target. The microarchitecture-independent characteristics that we measure cover a wide range of program characteristics, namely instruction mix, control flow behavior, instruction stream locality, data stream locality, and instruction-level parallelism. We discuss these characteristics in detail now.

3.1.1 *Control-Flow Behavior and Instruction-Stream Locality.* It has been well observed that the instructions in a program exhibit the *locality of reference* property. The locality of reference is often stated in the rule of thumb called the 90/10 rule, which states that a program typically spends 90% of its execution time in only 10% of its static code. In order to model this program property in a synthetic benchmark clone, it is essential to capture the control-flow structure of the program, i.e., we need to model how basic blocks are traversed in

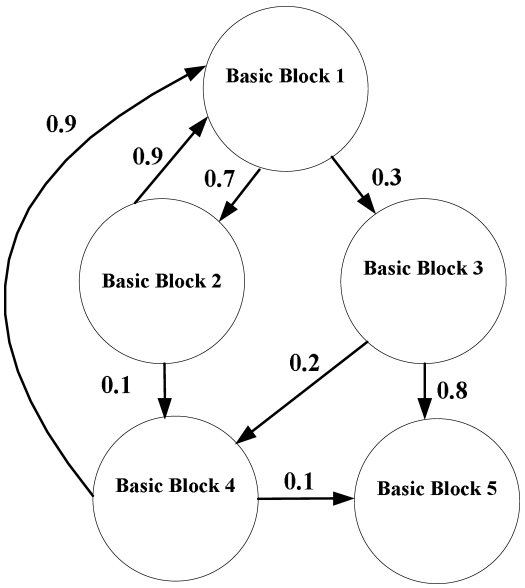


Fig. 2. An example SFG used to capture the control-flow behavior and instruction stream locality of a program.

the program and how branch instructions alter the direction of control flow in the instruction stream. During the application profiling phase we capture the control-flow information using the statistical flow graph (SFG) described in Eeckhout et al. [2004]. Each node in the SFG represents a unique basic block and the edges represent the transition probabilities between basic blocks. Figure 2 shows an example SFG that is generated by profiling the execution of a program. The probabilities marked on the edges of each basic block indicate the transition probabilities, e.g., if basic block 1 was executed, the probability that basic block 2 will be executed next equals 0.7.

We measure the workload characteristics described below per unique pair of predecessor and successor basic blocks in the SFG. For example, instead of measuring a single workload profile for basic block 4, we maintain separate workload characteristics profiles for basic block 4 dependent on its predecessor, basic block 2 or basic block 3. By doing so, the workload characteristics are correlated with the execution path of a program, which improves the modeling accuracy [Eeckhout et al. 2004].

Table I provides a summary of the information that can be gleaned from the SFGs for the complete executions of the SPEC CPU2000 integer and floating-point programs, as well as a set of MiBench and MediaBench embedded programs—we defer to later in this paper for a detailed description of the experimental setup. The table shows the number of unique basic blocks in each program, the number of basic blocks that account for 90% of the dynamic program execution, the average basic block size, and the number of successor basic blocks. Among all the SPEC CPU2000 programs, `gcc` has the largest footprint but still exhibits a very high instruction locality (only 9% of the basic blocks

Table I. Summary of the Information Captured by the SFG

(a) SPEC CPU2000 Integer and Floating-Point Benchmarks

Benchmark	Number of Basic Blocks	Number of Basic Blocks that Account for 90% of Program Execution	Average Basic Block Size	Average Number of Successor Basic Blocks
applu	348	104	112.3	1.8
apsi	262	127	28.6	3.3
art	69	6	7.7	2.7
bzip2	139	40	7.1	3.2
crafty	514	151	10.5	7.4
eon	303	119	9.3	4.7
equake	47	10	39.6	1.6
gcc	1088	98	6.9	10.5
gzip	163	30	8.7	2.3
mcf	178	23	4.2	4.4
mesa	211	74	16.4	3.2
mgrid	210	11	109.2	1.9
perlbmk	54	34	5.2	2.0
swim	63	17	41.5	1.1
twolf	155	55	8.2	5.1
vortex	626	44	4.9	6.6
vpr	84	21	7.2	2.2
wupwise	130	47	10.3	4.3

(b) Embedded Benchmarks from MiBench and MediaBench Benchmark Suites

Benchmark	Number of Basic Blocks	Number of Basic Blocks that Account for 90% of Program Execution	Average Basic Block Size	Average Number of Successor
basicmath	371	98	6.8	7.6
bitcount	240	8	8.3	3.0
crc32	311	146	6.2	11.2
dijkstra	366	15	6.9	2.3
fft	366	101	8.9	6.8
ghostscript	2549	62	6.7	4.9
gsm	312	142	6.2	10.3
jpeg	695	30	11.3	4.5
patricia	419	136	6.0	11.6
qsort	319	58	5.2	8
rsynth	536	22	10.2	2.9
stringsearch	160	46	6.7	5.1
susan	364	6	16.5	2.4
typeset	875	86	7.6	6.7
cjpeg	711	31	10.1	5.0
djpeg	702	45	23.5	5.1
epic	650	21	6.0	3.8
g721-decode	299	40	8.1	5.3
mpeg-decode	514	27	16.9	3.2
rasta	1089	215	10.6	9.7
rawaudio	119	3	19.1	2.2
texgen	886	71	10.7	3.9

account for 90% of the program execution). All the other benchmarks have fairly small instruction footprints, suggesting that they do not stress the instruction cache. The embedded benchmarks also have very modest footprints, with `ghostscript` having the largest footprint. Compared to the SPEC CPU benchmarks, the embedded benchmarks exhibit higher instruction locality; on average, 90% of the time is spent in 13% of the basic blocks, compared to SPEC CPU benchmarks where 90% of the time is spent in 27% of the basic blocks. The average size for the floating-point programs is 58.2, with `applu` (111.7), and `mgrid` (109.2) having very large basic blocks. On the other hand, the average basic block size for the integer programs is 7.9. The average basic block size for the embedded benchmarks, 10 instructions, is slightly larger than for the SPEC CPU integer programs—with `djpeg` (23.5) having a fairly large average basic block size.

The average number of successors for each basic block is a measure for the control-flow complexity in the program—the higher the number of successor basic blocks, the more complex the control-flow behavior. `crafty` and `gcc` are the two benchmarks which have a large number of successors, 7.4 and 10.5 basic blocks, respectively. This high number of basic blocks is because of returns from functions that are called from multiple locations in the code and multimodal switch statements. At the other end of the spectrum, basic blocks in programs, such as `applu`, `equake`, `gzip`, `mgrid`, `swim`, `perlbmk`, and `vpr` only have one or two successor basic blocks, suggesting that they execute in tight loops most of the time. Interestingly, some of the embedded benchmarks, `crc32`, `gsm`, `patricia`, `qsort`, and `rasta` have a large number of successor basic blocks, suggesting complex control flow. For the other embedded benchmarks the average number of successor basic blocks is 4, which is nearly the same as for the SPEC CPU benchmark programs.

The SFG thus captures a picture of the program structure and its control-flow behavior. We will use the SFG to mimic an application's control-flow structure in its synthetic benchmark clone.

3.1.2 *Instruction Mix.* The instruction mix of a program measures the relative frequency of operation types appearing in the dynamic instruction stream. We measure the percentage integer arithmetic, integer multiply, integer division, floating-point arithmetic, floating-point multiply, floating-point division operations, load, store, and branch instructions. The instruction mix is measured separately for every basic block.

3.1.3 *Instruction-Level Parallelism.* The dependency distance is defined as the number of instructions in the dynamic instruction stream between the production (write) and consumption (read) of a register and/or memory location. The goal of characterizing data-dependency distances is to capture a program's inherent ILP. For every instruction, we measure the data-dependency distance information on a per-operand basis as a distribution organized in eight buckets: percentage of dependencies that have a dependency distance of 1 instruction and the percentage of dependencies that have a distance of up to 2, 4, 6, 8, 16, 32, and greater than 32 instructions. This dependency-distance distribution is

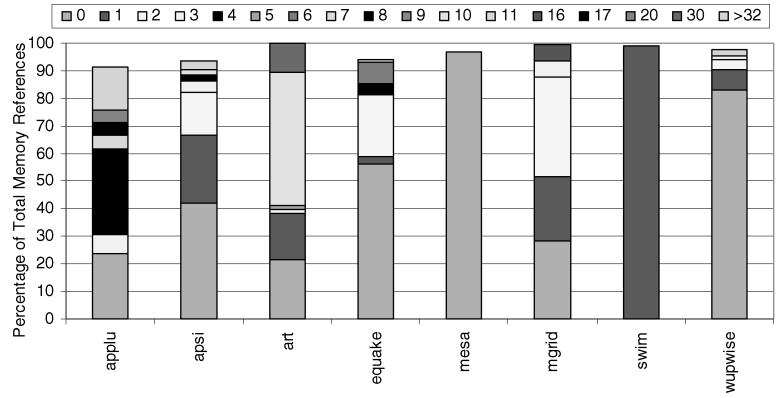
maintained separately for each unique pair of predecessor and successor basic blocks in the program.

3.1.4 *Data-Stream Locality.* The principle of data-reference locality is well known and recognized for its importance in affecting an application's performance. Traditionally, data locality is considered to have two important components, temporal and spatial locality. Temporal locality refers to locality in time and is because of the fact that data items that are referenced now tend to be referenced soon in the near future. Spatial locality refers to locality in space and is because of the program property that when a data item is referenced, nearby items tend to be referenced soon. Previous work [Sorenson and Flanagan 2002] shows that these abstractions of data locality and their measures are insufficient to replicate the memory-access pattern of a program. Therefore, instead of quantifying temporal and spatial locality by a single number or a simple distribution, our approach for mimicking the temporal and spatial data locality of a program is to characterize the memory accesses of a program to identify the patterns with which a program accesses memory on a per-instruction basis and then replicate that in the synthetic benchmark.

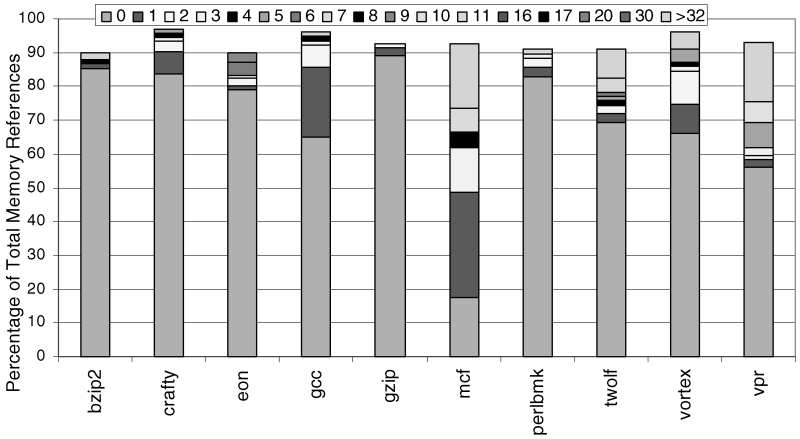
The order in which a program accesses memory locations is a function of its dynamic traversal of the control flow graph and the memory access patterns of its individual load and store instructions. One may not be able to easily identify patterns or sequences when observing the global data access stream of the program. This is because several memory-access patterns coexist in the program and are generally interleaved with each other. Thus, the problem that we are trying to address is how to efficiently extract patterns from the global sequence of memory addresses issued by the program. When a compiler generates a memory-access instruction, load or store, it has a particular functionality—it accesses a global variable, stack, array, or a data structure. This functionality of the memory access instruction is consistent and stable and determines how the instruction generates effective addresses. This suggests that rather than studying the global memory-access stream of a program, it may be better to view the data access patterns at a finer granularity of individual memory access instructions.

We profile general-purpose, scientific, and embedded benchmark programs (described later), and measure the stride values (differences between two consecutive effective addresses) per static load and store instruction in the program. We use this information to calculate the most frequently used stride value for each static load and store instruction and the percentage of the memory references for which it was used. If a static memory access instruction uses the same stride more than 80% of the time, we classify the instruction as a *strongly strided instruction*. We then plot a cumulative distribution of the stride patterns for the most frequently used stride values for all of the strongly strided memory access instructions and the percentage of the dynamic memory access instructions that they represent.

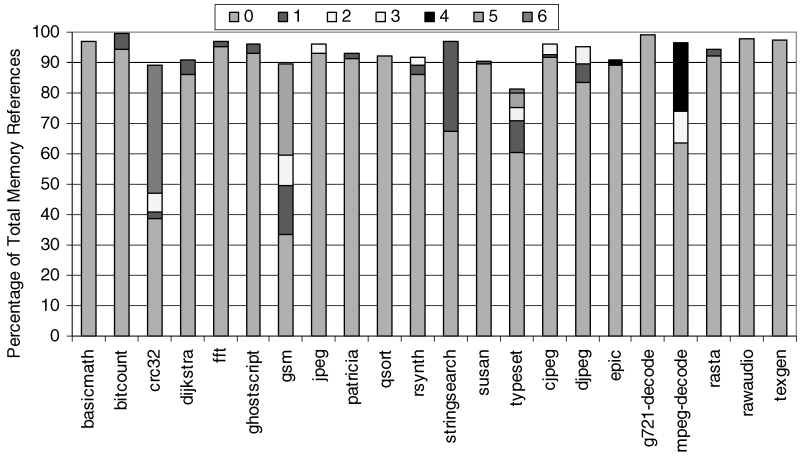
Figures 3(a), (b), and (c) show this cumulative distribution for the SPEC CPU2000 floating-point, integer, and embedded benchmarks, respectively. The stride values are shown at the granularity of a 64-byte block (analogous to a



(a) SPEC CPU 2000 Floating-Point Programs.



(b) SPEC CPU 2000 Integer Programs.



(c) Embedded Programs from MediaBench & MiBench Suites.

Fig. 3. Percentage breakdown of stride values per static memory access.

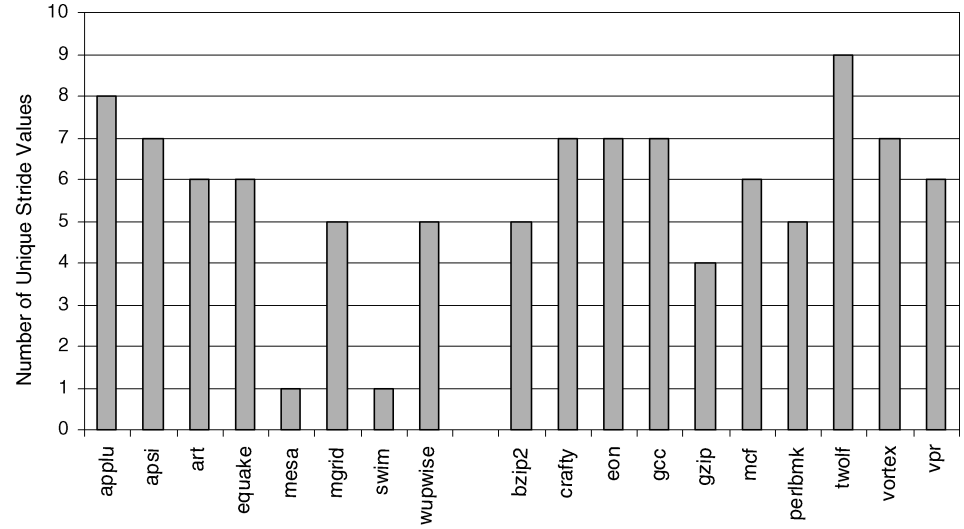
cache line), i.e., if the most frequently used stride value for an instruction is between 0 and 63 bytes, it is classified as a zero stride (consecutive addresses are within a single 64-byte block), between 64 and 127 bytes as stride 1, etc.

The results from Figure 3(a) show that for floating-point benchmarks almost all the references issued by a static load or store instruction can be classified as strides. This observation is consistent with the common understanding of floating-point program behavior—the data structures in these programs are primarily arrays and, hence, the memory instructions are expected to generate very regular access patterns. The access patterns of `swim` and `mesa` are very unique—all the static memory access instructions in `swim` walk through memory with a stride of one cache line size, and those in `mesa` with a zero stride. Also, `wupwise` has the zero stride as its dominant stride. The number of prominent unique stride values in other floating-point program varies between 5 (`mgrid`) and 8 (`applu`). Benchmark `art`, a program that is known to put a lot of pressure on the data cache, also has a very regular access pattern with almost all the load and store instructions accessing memory with stride values of 0, 1, 10, and 30.

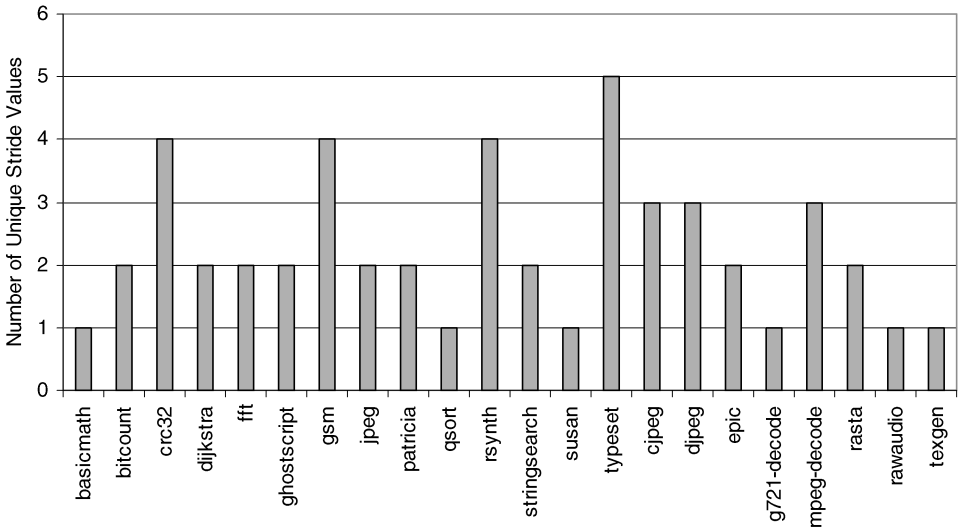
The static load and store memory access characterization behavior of the SPEC CPU2000 integer benchmarks, shown in Figure 3(b), appears to be rather surprising at first sight. The results show that a large number of programs have just one single dominant access stride pattern—more than 80% of the accesses from static loads and stores for `bzip2`, `crafty`, `eon`, `gzip`, and `perlbmk` have the zero stride as their dominant stride. Also, a large percentage (more than 90%) of all memory references in programs, such as `mcf`, `twolf`, `vpr`, and `vortex` appear to exhibit regular stride behavior. These programs are known to execute pointer-chasing code and it is expected that this results in irregular data access patterns. The observation made from Figure 3(b) though suggests that in these pointer-intensive programs, linked list, and tree structure elements are frequently allocated at a constant distance from each other in the heap. As a result, when linked lists are traversed, a regular pattern of memory accesses with constant strides emerges. Recent studies aimed at developing prefetching schemes have made similar observations (see for example Collins et al. [2001], Stoutchinin et al. [2001], and Wu [2002]); they developed stride-based prefetching schemes that improve the performance of pointer-intensive programs, such as `mcf`, by as much as 60%.

We observe in Figure 3(c) that for embedded benchmarks, too, more than 90% of the dynamic memory accesses originate from strongly strided static memory access instructions. Most of the embedded benchmarks have the zero stride as their dominant stride. The exceptions are `crc32`, `dijkstra`, `rsynth`, and `typeset` for which there are four to five different unique stride values.

From this characterization study of the memory access patterns we can infer that the memory-access patterns of programs (both general-purpose integer, embedded, and floating-point) are amenable to be modeled using a stride-based model on a per-instruction basis. We, thus, record the most frequently used stride value for every static memory access instruction in every node in the statistical flow graph. Also, during the profiling phase, the starting address per static load or store instruction and the average length of the stride stream with



(a) SPEC CPU Floating-Point and Integer Benchmarks.



(b) Embedded Benchmarks from MiBench & MediaBench Suites.

Fig. 4. Number of different dominant memory-access stride values per program.

which each static load or store instruction accesses data is recorded—this is measured by calculating the number of consecutive positive stride references issued by a static-memory instruction before seeing a break in the stride pattern. Note that the dominant stride value and the average stream length can be different for different static memory-access instructions.

Figure 4 summarizes the number of different dominant strides with which static load and store instructions in integer, floating-point, and embedded

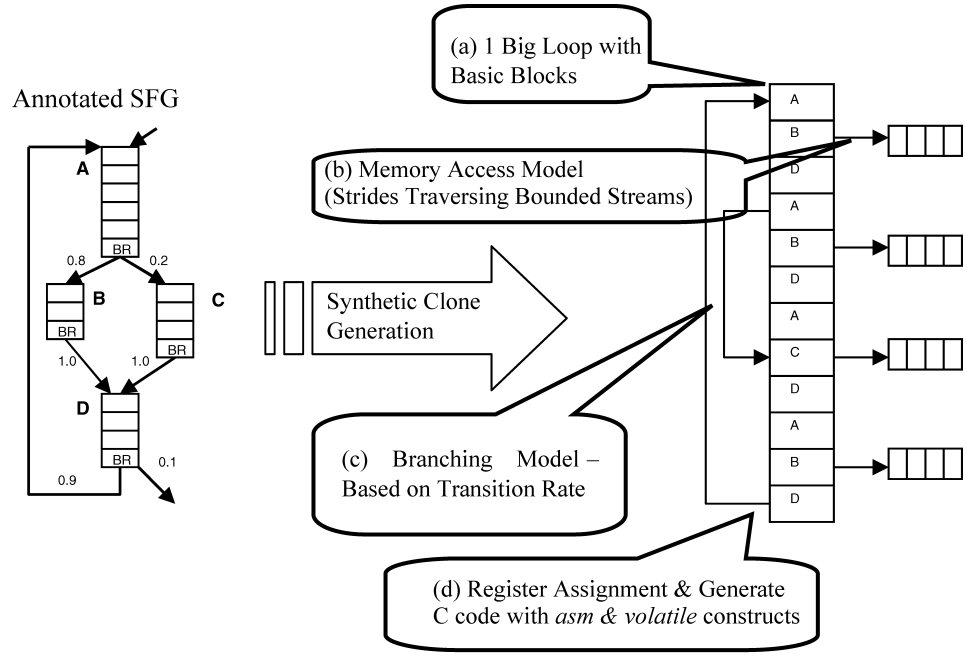
programs access data. The vertical axis shows the number of different strongly strided values with which memory instructions access data. We observe that a maximum of 9 stride values (`twolf`) are seen across the strongly strided memory-access instructions. When constructing a synthetic benchmark clone, we model each static load/store instruction as accessing a bounded memory region with its most frequent stride value and its average stream length obtained from this workload characterization. It will be clear when we describe the synthesis algorithm (Section 3.2), that having a small number of dominant stride values makes it feasible to incorporate the memory-access model in the synthetic benchmark clone—only 9 registers are required to store the stride values.

3.1.5 *Control-Flow Predictability.* In order to incorporate inherent branch predictability in the synthetic benchmark clone, it is essential to understand the property of branches that makes them predictable. The predictability of branches stems from two sources: (1) most branches are highly biased toward one direction, i.e., the branch is either taken or not taken most of the time, and (2) the outcome of branches may be correlated.

In order to capture the inherent branch behavior of a program, the most popular microarchitecture-independent metric is to measure the taken rate per static branch, i.e., the fraction of the times that a static branch was taken during the complete run of the program. Branches that have a very high or very low taken rate are biased toward one direction and are considered to be highly predictable. However, merely using the taken rate of branches is insufficient to actually capture the inherent branch behavior. Consider two sequences of branches, one has a large number of taken branches followed by an equally long number of not-taken branches, whereas the other sequence does not have such a regular pattern and switches randomly between taken and not-taken directions. Both sequences have the same taken rate (50%), but may have different branch misprediction rates. It is clear that the former sequence is better predictable than the latter. This suggests that branch predictability depends more on the sequence of taken and not-taken directions and not just on the taken rate.

Therefore, in our control-flow predictability model, we also measure an attribute called the branch *transition rate* [Haungs et al. 2000] for capturing the branch behavior in programs. The transition rate of a static branch is defined as the number of times it switches between taken and not-taken directions as it is executed, divided by the total number of times that it is executed. By definition, branches with low transition rates are always biased toward either taken or not taken. It has been well observed that such branches are easy to predict. Also, branches with a very high transition rate always toggle between taken and not taken directions and are also highly predictable. However, branches that transition between taken and not-taken sequences at a moderate rate are relatively more difficult to predict.

In order to incorporate synthetic branch predictability we annotate every node in the statistical flow graph with its transition rate. When generating the synthetic benchmark clone, we ensure that the distribution of the transition



(a) Steps in benchmark clone synthesis

```
_asm_ _volatile_ ("$LINSTR126: addq $23, 64, $26" : "=r" (vout_26), "=r" (vout_23): "r" (vout_26), "r" (vout_23));
_asm_ _volatile_ ("$LINSTR127: addq %0, 0, %0" : "=r" (vout_12): "r" (vout_12));
_asm_ _volatile_ ("$LINSTR128: ldl $27,0(%1)" : "=r" (vout_27), "=r" (vout_12): "r" (vout_27), "r" (vout_12));
_asm_ _volatile_ ("$LINSTR129: addq %0, 0,%0": "=r" (vout_11): "r" (vout_11));
_asm_ _volatile_ ("$LINSTR130: addq %0, 4,%0" : "=r" (vout_14) : "r" (vout_14));
_asm_ _volatile_ ("$LINSTR131: ldl $25,0(%1)": "=r" (vout_25), "=r" (vout_14): "r" (vout_25), "r" (vout_14)) ;
_asm_ _volatile_ ("$LINSTR132: addq %0, 0,%0": "=r" (vout_12) : "r" (vout_12));
_asm_ _volatile_ ("$LINSTR133: beq $12, $LINSTR149": "=r" (vout_12): "r" (vout_12));
```

(b) Code snippet from one basic block of the synthetic clone

Fig. 5. Illustration of the synthetic benchmark synthesis process.

rates for static branches in the synthetic clone is similar to that of the original program. We achieve this by configuring each basic block in the synthetic stream of instructions to alternate between taken and not-taken directions, such that the branch exhibits the desired transition rate. The algorithm for generating the synthetic benchmark program in the next section describes the details of this mechanism.

3.2 Benchmark Clone Synthesis

The second step in the benchmark cloning framework is to generate a synthetic benchmark by modeling all the microarchitecture-independent workload characteristics from the previous section into a synthetic clone. The basic structure

of the algorithm used to generate the synthetic benchmark program is similar to the one proposed by Bell and John [2005]. However, the memory and branching model is replaced with the microarchitecture-independent models described in the previous section.

The clone generation process comprises five substeps: SFG analysis, memory-accessing pattern modeling, branch-predictability modeling, register assignment, and code generation. Figure 5(a) illustrates each of these steps.

3.2.1 *Control-Flow Graph Generation.* In this step, the SFG profile is used for generating the basic template for the benchmark. The SFG is traversed using the branching probabilities for each basic block and a linear chain of basic blocks is generated. This linear chain of basic blocks forms the spine of the synthetic benchmark program (refer to step (a) in Figure 5). The spine is the main loop in the synthetic clone that will be iterated multiple times during its execution. The length of the spine should be long enough to reflect the average basic block size and the representation of the most frequently traversed basic blocks in the program. The average basic block size and the number of basic blocks in the program, shown in Table I, is used as a starting point to decide on the number of basic blocks in the spine. We then tune the number of basic blocks to match the overall instruction mix characteristics by iterating through the synthesis a small number of times. The number of iterations over which the main loop is executed is set such that performance characteristics of the synthetic clone converge to a stable value. Our experiments, discussed in Section 5.2, show that a total of approximately 10 million dynamic instructions are required for convergence of the synthetic clone. This dynamic instruction count is used to determine the number of times the main loop of the synthetic clone is executed.

The algorithm used for instantiating the basic blocks in the synthetic clone is as follows:

1. Generate a random number in the interval [0,1] and use this value to select a basic block in the statistical flow graph (SFG) based on the cumulative distribution function that is built up using the occurrence frequencies of each basic block.
2. Output a sequence of instructions per basic block. Assign instruction types to the instructions using the instruction mix distribution for that basic block. Depending on the instruction type, assign the number of source operands for each instruction.
3. For each source operand, assign a dependency distance using the cumulative dependency distance distribution. This step is repeated until a real dependency is found that ensures syntactical correctness, i.e., the source operand of an instruction cannot be dependent on a store or branch instruction. If this dependency cannot be satisfied after 100 attempts, the dependency is simply squashed.
4. A cumulative distribution function based on the probabilities of the outgoing edges of the nodes in the SFG is then used to determine the next node. If the node does not have any outgoing edges, go to step 1.

5. If the target number of basic blocks (equal to the total number of basic blocks in the original program) has not been generated, go to step 2. If the target number of basic blocks has been generated, the algorithm terminates.

3.2.2 *Modeling Memory-Access Pattern.* For each memory-access instruction in the synthetic clone we assign its most frequently used stride along with its stream length. A load or store instruction is modeled as a memory operation that accesses a circular and bounded stream of references, i.e., each memory access walks through an array using its dominant stride value and then restarts from the first element of the array (step (b) in Figure 5). An arithmetic instruction in each basic block is assigned to increment the stride value for the memory walk. The stride value itself is stored in a register. Since the maximum number of stride values in the program is 9 (see Section 3.1.4), we do not need a large number of registers to store the various stride values in the synthetic benchmark.

3.2.3 *Modeling Branch Predictability.* For each static branch in the spine of the program, we identify branches with very high or very low transition rates (branches with a transition rate of less than 10% or greater than 90%) and model them as always taken or not taken. Branches with moderate transition rates are configured to match the transition rate of the corresponding static branch. A register variable is assigned for each transition rate category (a maximum of eight categories). For every iteration of the master loop, a modulo operation is used to determine whether the static branches belonging to a particular transition rate category will be taken or not taken (step (c) in Figure 5). The static branches in the program use these register variables as a condition to determine the branch direction. If the branch direction is taken, the target is address is determined from the SFG.

3.2.4 *Register Assignment.* In this step we use the dependency distances that were assigned to each instruction to assign registers. A maximum of eight registers are needed to control the branch transition rate and a maximum of nine registers are used for controlling the memory access patterns. The number of registers that are used to satisfy the dependency distances is typically kept to a small value (typically around 10) to prevent the compiler from generating stack operations that store and restore the values.

3.2.5 *Code Generation.* During the code-generation phase, the instructions are emitted out with a header and footer. The header contains initialization code that allocates memory using the *malloc* library call that the memory operations will access and, in addition, it assigns memory stride values to variables. Each instruction is then emitted out with assembly code using *asm* statements embedded in C code. The instructions are targeted toward a specific ISA, Alpha, in our case. However, the code generator can be modified to emit instructions for an ISA of interest. The *volatile* directive is used to prevent the compiler from reordering the sequence of instructions and changing the dependency distances between instructions in the program.

Table II. SPEC CPU 2000 Programs, Input Sets, and Simulation Points Used in this Study

Program	Input	Type	SimPoint
applu	ref	FP	46
apsi	ref	FP	3408
art	110	FP	340
equake	ref	FP	812
mesa	ref	FP	1135
mgrid	ref	FP	3292
swim	ref	FP	2079
wupwise	ref	FP	3237
bzip2	graphic	INT	553
crafty	ref	INT	774
eon	rushmeier	INT	403
gcc	166.i	INT	389
gzip	graphic	INT	389
mcf	ref	INT	553
perlbmk	perfect-ref	INT	5
twolf	ref	INT	1066
vortex	lendian1	INT	271
vpr	route	INT	476

Figure 5(b) shows a snippet of code for one basic block from the synthetic clone targeted toward the Alpha ISA. Each instruction comprises of an assembler instruction template comprising of a label (e.g., $LINSTR126), an instruction type (e.g., addq), the input and output registers in assembly language (e.g., $23) or operands corresponding to C-expressions (e.g., %0, %1), operand constraint for register type (e.g., "r" for integer and "f" for floating-point), and register variables in C-expressions (e.g., vout_22). The read-after-write dependencies between instructions are enforced through the register variables, e.g., $LINSTR127 and $LINSTR128 through register variable vout_12. Please refer to Coleman [1998] for details on the syntax assembler format and techniques for specifying the operand constraint string.

4. EXPERIMENTAL SETUP

We use a modified version of the SimpleScalar [Burger and Austin 1997] functional simulator `sim-safe` to measure the workload characteristics of the programs. An alternative to simulation is to measure these characteristics using a binary instrumentation tool, such as ATOM [Srivastava and Eustace 1994] or PIN [Luk et al. 2005]. In order to evaluate and compare the performance characteristics of the real benchmark and its synthetic clone, we use SimpleScalar's `sim-outorder`. We use Wattch [Brooks et al. 2000] to measure the power characteristics of the benchmarks, and consider the most aggressive clock-gating mechanism in which an unused unit consumes 10% of its maximum power and a unit that is used only for a fraction n consumes only a fraction n of its maximum power.

In most of our experiments, we use one 100M-instruction simulation point selected using SimPoint [Sherwood et al. 2002] for the SPEC CPU2000

Table III. MediaBench and MiBench Programs and Their Embedded Application Domain

Program	Application Domain
basicmath, qsort, bitcount, susan	Automotive
crc32, dijkstra, patricia	Networking
fft, gsm	Telecommunication
ghostscript, rsynth, stringsearch	Office
jpeg, typeset	Consumer
cjpeg, djpeg, epic, g721-decode, mpeg, rasta, rawaudio, texgen	Media

Table IV. Baseline Processor Configuration

L1 I-cache and D-cache	16 KB/two-way/64 B line size
Fetch, decode, and issue width	Four-wide out-of-order
Branch predictor	Combined (2-level and bimodal), 4 KB
L1 I-cache & D-cache – size/assoc/latency	32 KB / four-way / 1 cycle
L2 Unified cache – size/assoc/latency	4 MB / eight-way / 10 cycles
RUU / LSQ size	128 / 64 entries
Instruction fetch queue	32 entries
Functional units	Two Integer ALU, two floating-point, one FP multiply/divide, and one Integer multiply/divide unit
Memory bus width, access time	8 B, 150 cycles

benchmarks (see Table II); we also consider complete simulation of the benchmarks to assess the synthetic benchmark cloning method for longer-running benchmarks. As a representative of the embedded application domain we use benchmarks from the MiBench and MediaBench suite (see Table III). The MiBench and MediaBench programs were run to completion. All benchmark programs were compiled on an Alpha machine using the native Compaq cc v6.3-025 compiler with the –O3 optimization setting.

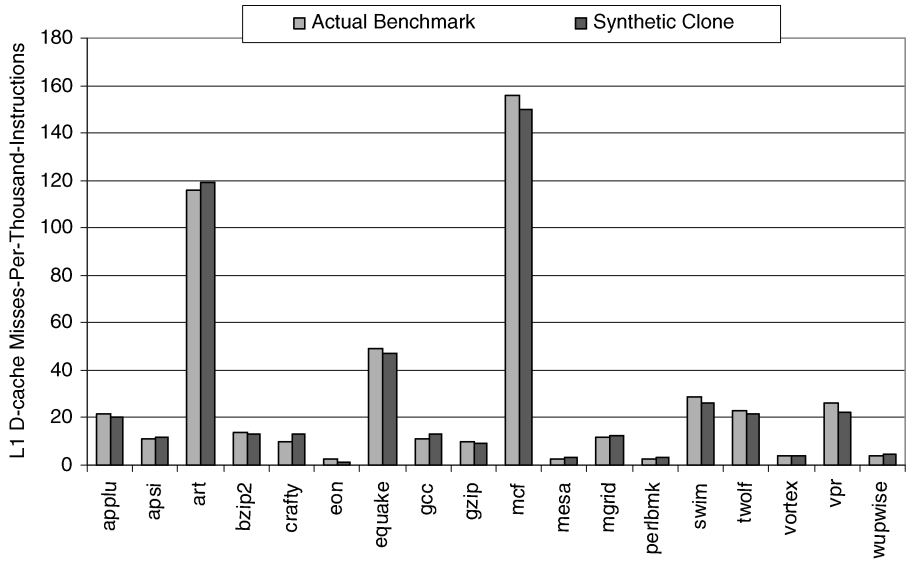
In order to evaluate the representativeness of the synthetic clones, we use a four-way issue out-of-order superscalar processor as our baseline configuration (Table IV).

5. EVALUATION OF SYNTHETIC BENCHMARK CLONE

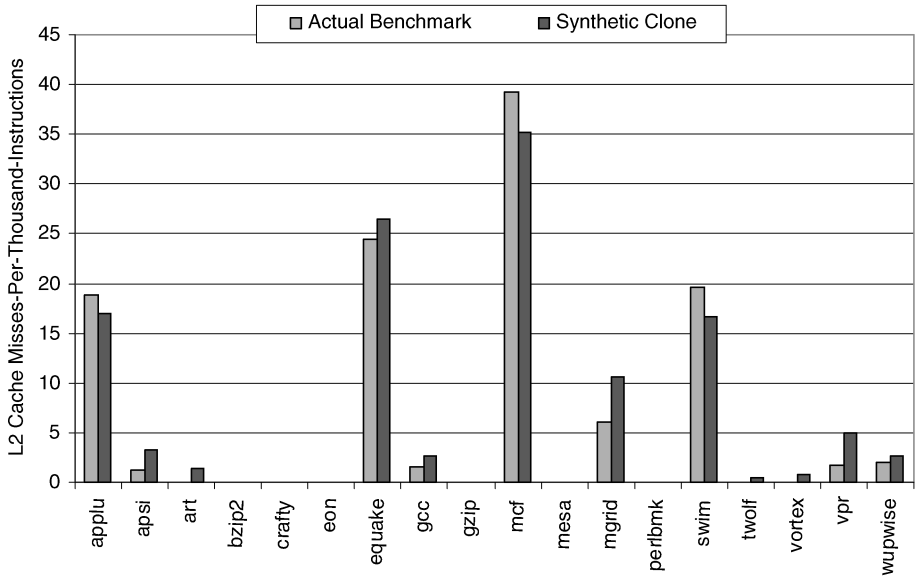
We evaluate the accuracy and usefulness of the synthetic benchmark cloning approach by applying the technique to generate clones that are representative of general-purpose, scientific, and embedded domain benchmarks. In our evaluation, we compare the workload characteristics of the synthetic clone with those of the original program, absolute, and relative accuracy in estimating performance and power, convergence characteristics of the synthetic benchmark clone, and the ability of the synthetic benchmark to assess design changes.

5.1 Workload Characteristics

In this section we evaluate the proposed memory-access and branching models proposed in this paper, by comparing the cache misses-per-instruction and branch direction-prediction rates of the synthetic clones with those of the original programs.



(a) L1 D-cache misses



(b) L2 cache misses

Fig. 6. Cache misses-per-thousand-instructions for the SPEC CPU2000 benchmarks and their synthetic clones.

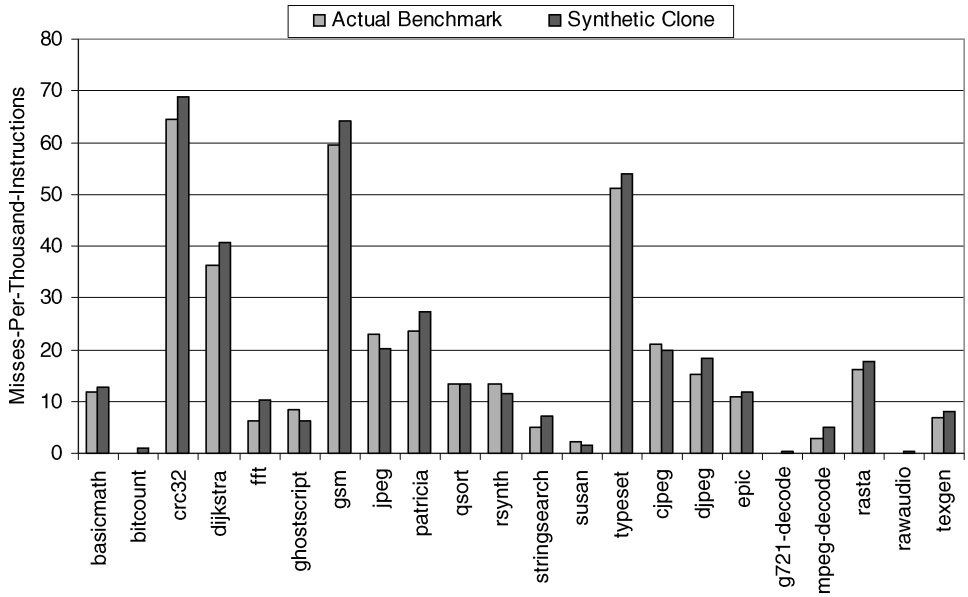


Fig. 7. Cache misses-per-thousand-instructions per benchmark and its synthetic clone for the embedded benchmarks.

5.1.1 *Cache-Behavior.* Figure 6 shows the L1 and L2 data cache misses-per-thousand-instructions for the original SPEC CPU benchmarks and their synthetic benchmark clones on the base configuration.

The average absolute difference in the number of L1 misses-per-thousand-instructions between the actual benchmark and the synthetic clone is two misses-per-thousand-instructions, with `mcf` having a maximum error of six misses-per-thousand-instructions. Looking at the L2 cache misses (see Figure 6(b)), we observe that `mcf`, `equake`, `swim`, and `applu` are the only programs that cause a significant number of L2 cache misses—the rest of the programs have a footprint that is small enough to fit in the L2 cache. For the four programs that show a relatively high L2 miss rate, the difference between the misses estimated by the synthetic benchmark clone and the actual program is only about three misses-per-thousand-instructions.

For the embedded benchmarks, the number of L1 data cache misses-per-thousand-instructions is negligibly small, with a maximum of eight misses-per-thousand-instructions for benchmark `typeset`. The base configuration that we use in our evaluation has a 32 KB four-way L1 data cache, and the footprints of the embedded benchmarks are small and fit into this cache. Therefore, in order to make a meaningful comparison in terms of the number of misses-per-thousand-instructions between the synthetic clone and the actual benchmark for the embedded benchmarks, we use a 4-KB, two-way set-associative cache. Figure 7 shows the L1 data cache misses-per-thousand-instructions for this cache configuration. The maximum absolute error is 4.7 misses-per-thousand-instructions, for the `gsm` benchmark.

5.1.2 *Branch Behavior.* Figure 8 shows the branch prediction rate of the synthetic clone and the original benchmark on the hybrid branch predictor considered in the baseline processor configuration. The average error shown by the synthetic clone in estimating the branch prediction rate is 1.2%, with a maximum error of 5.2% for `bzip2`. For the embedded benchmarks, the average error in the branch prediction rate is 2.1% with the maximum error of 5.3% for `crc32`, which has the lowest branch prediction rate in the entire suite.

In summary, based on the results presented in this section, we can conclude that the proposed memory access and branch models are fairly accurate, and capture the inherent workload characteristics into the synthetic benchmark clone.

5.2 Accuracy in Performance and Energy Estimation

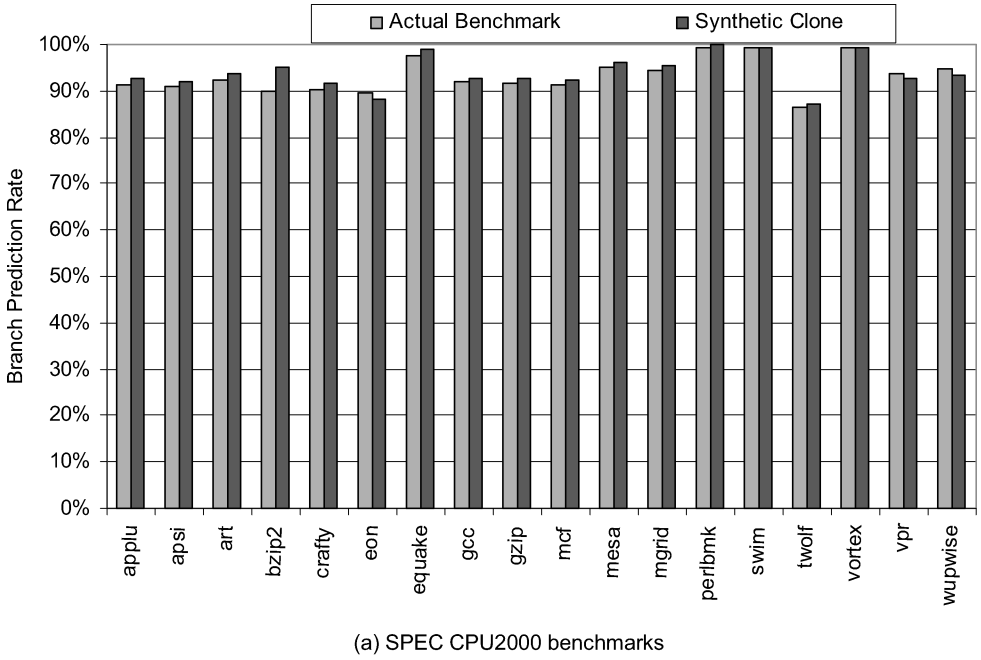
We now evaluate the accuracy of synthetic cloning in estimating overall performance and energy consumption. To this end, we simulate the actual benchmark and its synthetic clone on the baseline processor configuration outlined in Table IV. Figures 9 and 10 show the cycles-per-instruction (CPI) and energy-per-instruction (EPI) metrics, respectively. The average absolute error in CPI for the synthetic clone across all the SPEC CPU2000 benchmark configurations is 6.3%, with maximum errors of 9.9% for `mcf` and 9.5% for `swim`. The average absolute error in estimating the EPI is 7.3%, with maximum errors of 10.6% and 10.4% for `mcf` and `swim`, respectively. For the embedded benchmarks, the average error in estimating CPI and EPI using the synthetic clone is 3.9 and 5.5%, respectively; the maximum error is observed for `gsm` (9.1% in CPI and 10.3% in EPI).

In general, we conclude that memory-intensive programs, such as `mcf`, `swim`, `art`, and `twolf` have a higher than average absolute error in estimating CPI and EPI. Also, for programs with very poor branch predictability, such as gsm, the errors are higher than average. On the other hand, the errors for control-intensive programs with moderate or high-branch predictability, such as `bzip2`, `crafty`, `gcc`, `gzip`, and `perlbmk` are relatively smaller. Overall, from these results we can conclude that the synthetic benchmark clone can accurately estimate performance and power characteristics of the original application.

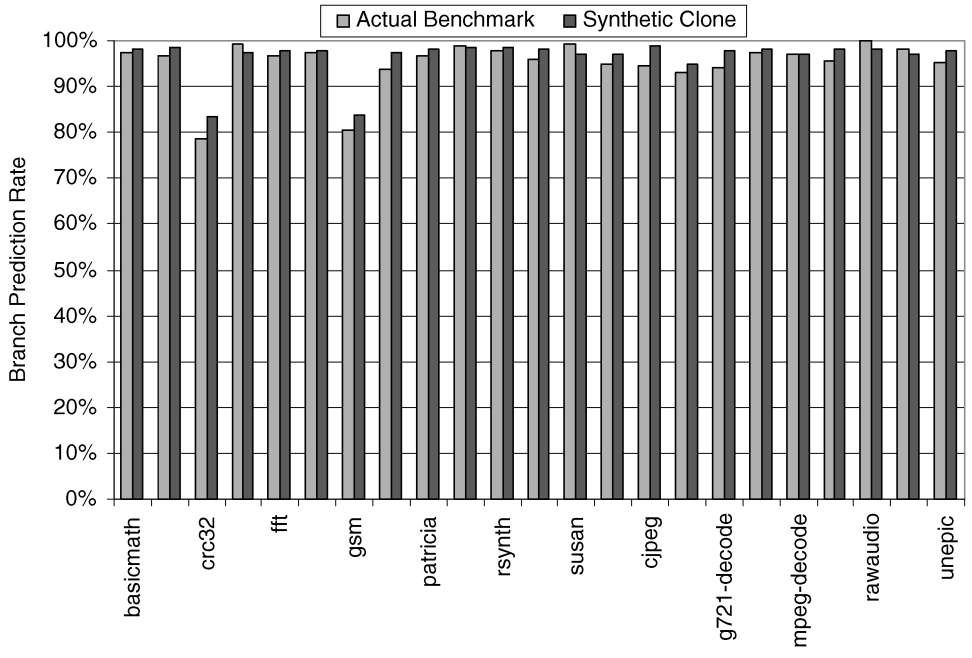
5.3 Convergence Property of the Synthetic Benchmark Clone

The instructions in the synthetic benchmark clone are generated by probabilistically walking through the SFG. In addition, the memory accesses are modeled as strides that traverse fixed-length arrays. Also, the branch instructions are configured to match a preset transition rate. As a result, if the entire spine of the program is executed in a big loop with a sufficient number of iterations, it will eventually reach steady state where the performance and power characteristics, such as CPI and EPI, converge.

Compared to the pipeline core structures, large caches will take a relatively longer time to reach steady state. Hence, in order to understand the upper bound on the number of instructions required for the program to converge, we select a memory-intensive benchmark that exhibits poor temporal locality.

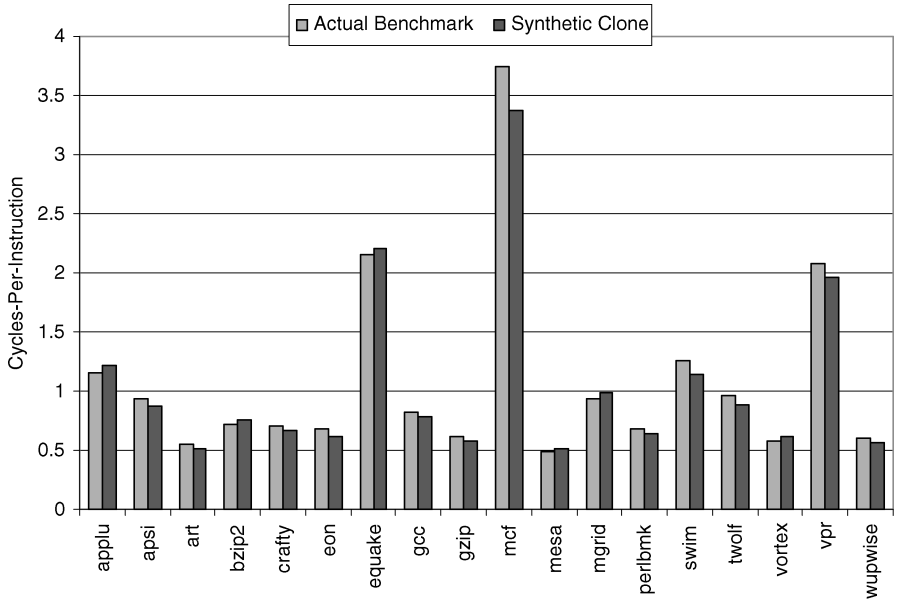


(a) SPEC CPU2000 benchmarks

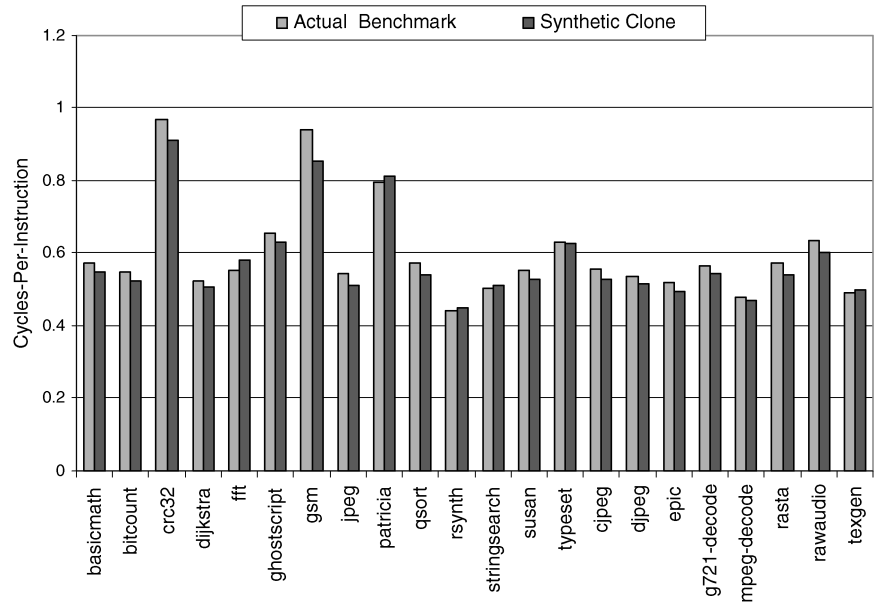


(b) Embedded benchmarks from MediaBench & MiBench suites

Fig. 8. Branch prediction rate per benchmark and its synthetic clone.

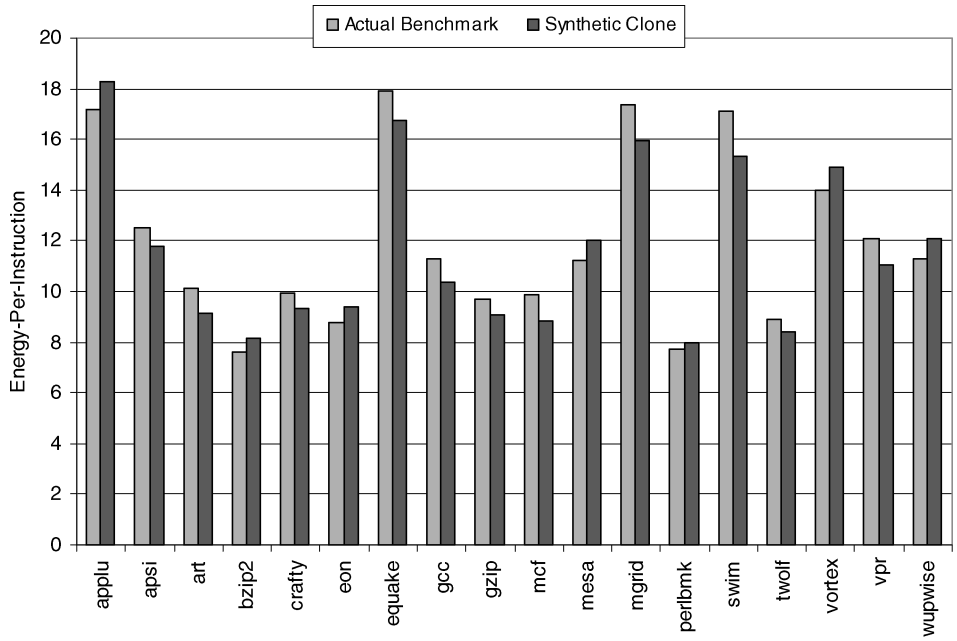


(a) SPEC CPU2000 benchmark programs

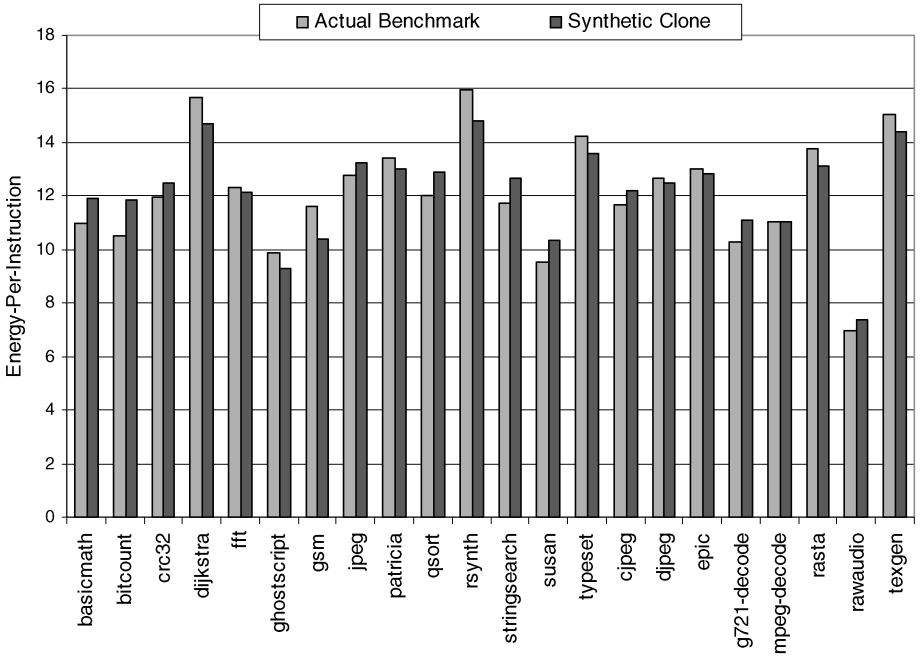


(b) Embedded Benchmarks from MediaBench & MiBench benchmark Suite

Fig. 9. Comparison of CPI for the synthetic clone versus the original benchmark.



(a) SPEC CPU2000 benchmark programs



(b) Embedded Benchmarks from MediaBench & MiBench benchmark Suite

Fig. 10. Comparison of energy-per-cycle for the synthetic clone versus the original benchmark.

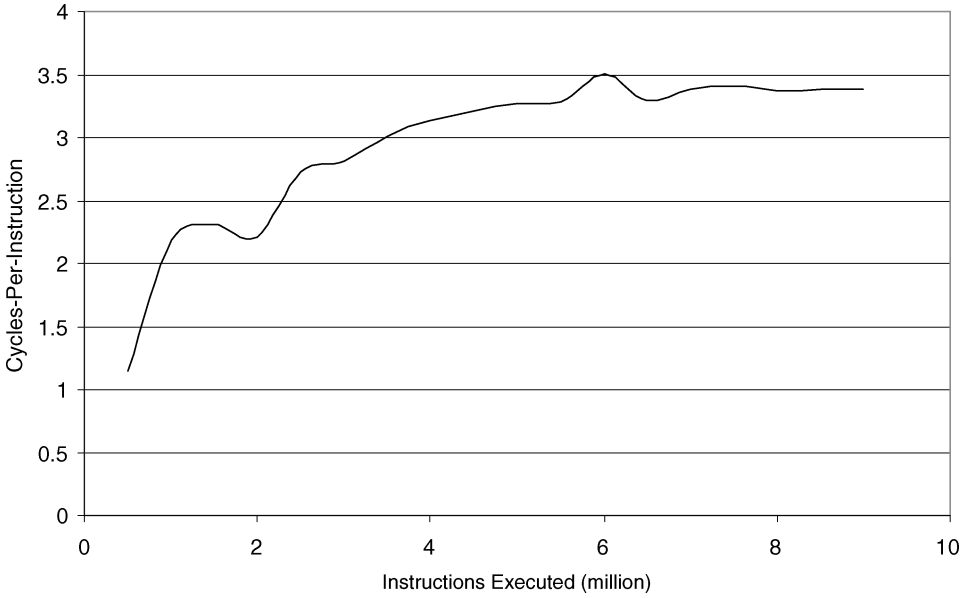


Fig. 11. CPI versus instruction count for the synthetic clone of `mcf`.

We have performed a similar analysis for all programs, but present the benchmark that took the largest number of instructions to converge. Benchmark `mcf` is one of the most memory-intensive programs with a very poor temporal locality, so we use this benchmark as an example. We simulated the synthetic benchmark clone for `mcf` on the base configuration described in Table IV, in a large outer loop and plotted the CPI against the dynamic instruction count (see Figure 11). The CPI initially increases and then stabilizes after around 9 million instructions; simulating more instructions only changes the CPI value by about 1%. The data cache misses in `mcf` are dominated by capacity misses and the misses-per-instruction increases during the course of the synthetic clone execution, which eventually stabilizes at a steady state value. We experimented with other programs and all of them converged to a steady state value within 9 million instructions. Thus, for the benchmark programs that we studied, we set 10 million instructions as an upper bound on the number of instructions required to converge. We can set this number as a guideline when selecting the outer loop count for the benchmark program—we typically set the value to 10 million instructions divided by the number of static instructions in the synthetic clone spine. If the L1 and L2 cache sizes are smaller than the one used in the configuration, the benchmark will converge faster, requiring less than 10 million instructions.

The synthetic benchmark clones that we generate can, therefore, be considered as representative miniature versions of the original applications. In our setup, where we use one 100 million instruction trace, we obtain a simulation speedup of an order of magnitude. If larger streams of instructions are considered, as shown the Section 5.5, the saving in simulation time is over five orders of magnitude.

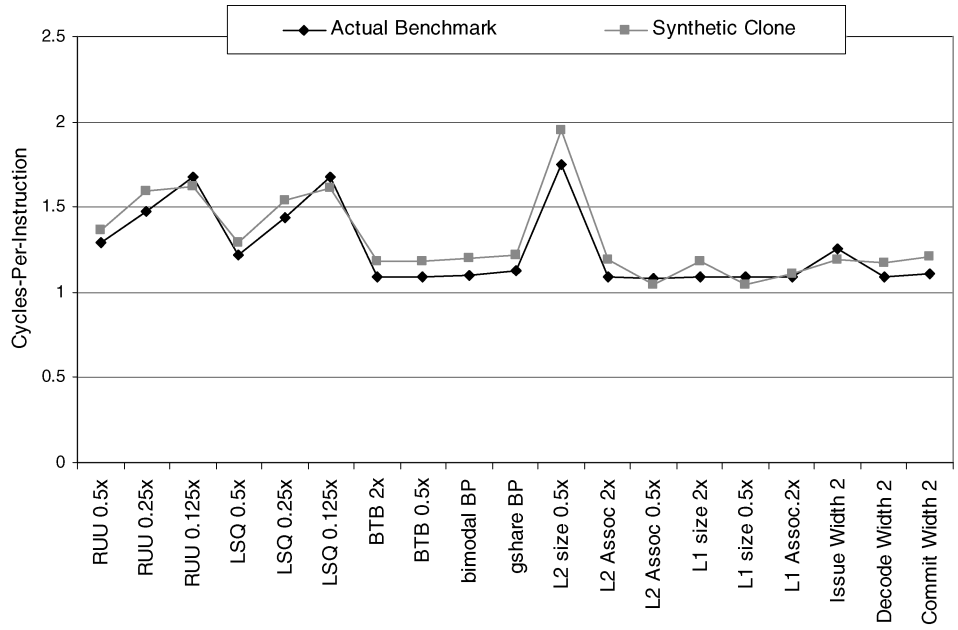


Fig. 12. Response of synthetic benchmark clone to design changes in base configuration.

5.4 Relative Accuracy in Assessing Design Changes

In our prior evaluation, we only measured the absolute accuracy of the synthetic benchmark clone in predicting performance and power for a single microarchitecture design point. However, in many empirical studies, predicting the performance trend is more important than predicting absolute performance. To evaluate the effectiveness of the synthetic benchmark clone in assessing design changes, we measure the relative accuracy by changing the cache sizes and associativity, reorder buffer size, processor width, and branch predictor configuration. In each experiment, all parameters have the baseline value, except for the parameter that is being changed. When changing the register update unit (RUU) and the load/store queue (LSQ) size, we ensure that the LSQ size is never larger than the RUU size. Figure 12 shows the design change on the horizontal axis and the CPI of the actual benchmark and its the synthetic clone on the vertical axis. The average error for the synthetic clone across all the design changes is 7.7%, with a maximum average error of 11.3% for the design change in which L2 cache size is reduced to one-half. As such, we conclude that the synthetic benchmark clone is able to effectively track design changes to the processor core and the memory hierarchy.

5.5 Modeling Long-Running Applications

In the prior sections, we showed that the synthetic benchmark clone exhibits good accuracy when the performance characteristics of the original program are measured from one representative 100-M instruction-simulation point. In order to evaluate the proposed synthetic benchmark generation methodology for

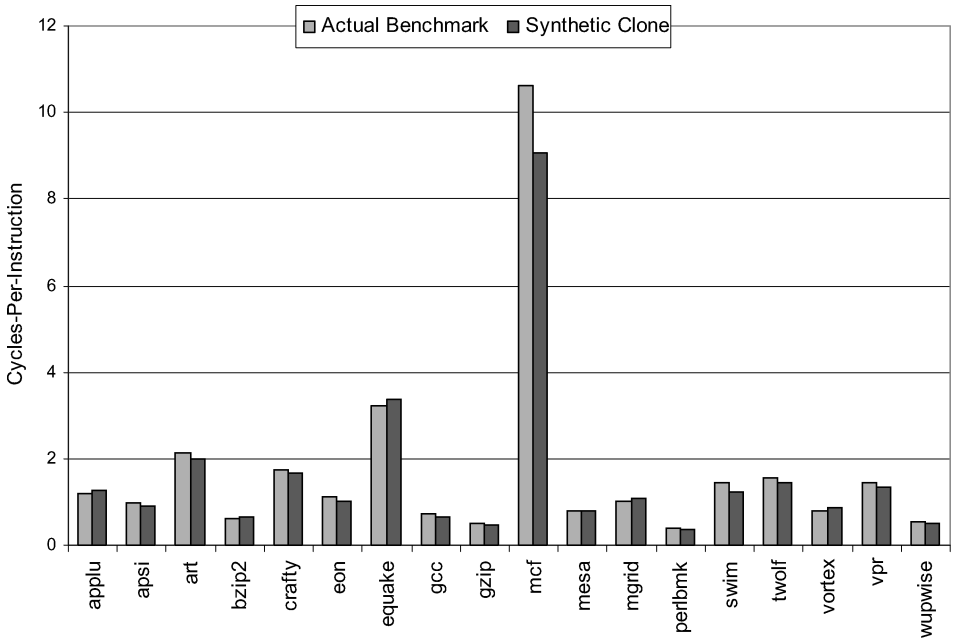


Fig. 13. Comparing the CPI of the synthetic clone and the actual benchmark for entire SPEC CPU2000 benchmark executions.

modeling longer running benchmarks, we now generate a synthetic clone that is representative of the complete run of each program and evaluate its representativeness and accuracy for the eight-way superscalar processor configuration from the SimPointwebsite using the published CPI numbers [SimPoint Website]. Figure 13 shows the CPI estimated by the synthetic clone and that of the actual benchmark program. The trend in errors is the same as for the 100-M instruction-simulation points, with `mcf` and `swim` having the highest maximum CPI prediction errors of 14.8 and 13.5%, respectively; the average CPI error is 8.2%. Table V shows the static instruction count of the synthetic clone and the simulation speedup through benchmark cloning compared to the original benchmark. These results show that the synthetic benchmark clones exhibit good accuracy even for long-running benchmarks and result in a simulation time reduction by more than five orders of magnitude.

6. DISCUSSION

As mentioned before, the advantage of the benchmark cloning approach proposed in this paper, as compared to previously proposed workload synthesis techniques [Bell and John 2005, 2006], is that all the workload characteristics modeled into the synthetic benchmark clone are microarchitecture-independent. This makes the benchmarks portable across a wide range of microarchitecture configurations. However, a limitation of the proposed technique is that the synthetic benchmark clone is dependent on the compiler technology that was used to compile the original proprietary application. Therefore,

Table V. Speedup from Synthetic Benchmark Cloning

Benchmark	Dynamic Instruction Count of Original Benchmark (Billion Instructions)	Speedup from Synthetic Benchmark Clone
applu	223	22,300
apsi	347	34,700
art	45	4,500
bzip2	128	12,800
crafty	191	19,100
eon	80	8,000
equake	131	13,100
gcc	46	4,600
gzip	103	10,300
mcf	61	6,100
mesa	141	14,100
mgrid	419	41,900
swim	225	22,500
twolf	346	34,600
vortex	118	11,800
vpr	84	8,400
wupwise	349	34,900

the generated synthetic benchmark clone may have limited application to the compiler community for studying the effects of various compiler optimizations. Also, the synthetic benchmark clone only imbibes the characteristics that were modeled, i.e., other characteristics such as value locality are not modeled and the benchmark clone cannot be used for studies exploiting these characteristics. However, if these characteristics are important and need to be modeled, one can always develop microarchitecture-independent metrics to capture their behavior and augment the benchmark cloning framework to mimic these characteristics into the synthetic benchmark clone.

A second note that we would like to make is that the synthetic benchmark clones that we generate contain instruction set architecture (ISA) specific assembly instructions embedded in C-code. Therefore, a separate benchmark clone would have to be synthesized for all target architectures of interest (e.g., Alpha, PowerPC, and IA-32). Typically, every designer and researcher would be interested only in his particular architecture and, therefore, this may not be a severe problem in practice. However, if the synthetic benchmark clone is to be made truly portable across ISAs, it would be important to address this concern. One possibility could be to generate the synthetic benchmark clone using a virtual instruction set architecture that can then be consumed by different back-end compilers for different ISAs. Another possibility would be to perform binary translation of the synthetic benchmark clone binary to the ISA of interest.

A final note is that the abstract workload model presented in this paper is fairly simple by construction, i.e., the characteristics that serve as input to the synthetic benchmark generation, such as the branching model and the data locality model, are far from being complicated. We have shown that even the observed behavior of pointer-intensive programs can be effectively modeled

using simple stride-based models. This was our intention: we wanted to build a model that is simple, yet accurate enough, for predicting performance trends of workloads.

7. RELATED WORK

7.1 Statistical Simulation

Oskin et al. [2000], Eeckhout and De Bosschere [2001], and Nussbaum and Smith [2001] introduced the idea of statistical simulation, which forms the foundation of synthetic workload generation. The approach used in statistical simulation is to generate a short synthetic trace from a statistical profile of workload attributes, such as basic block size distribution, branch misprediction rate, data/instruction cache miss rate, instruction mix, dependency distances, etc., and then simulate the synthetic trace using a statistical simulator. Eeckhout et al. [2004] improved statistical simulation by profiling the workload attributes at a basic block granularity using the statistical flow graph (SFG). Recent improvements include more accurate memory data flow modeling for statistical simulation [Genbrugge and Eeckhout 2008]. The important benefit of statistical simulation is that the synthetic trace is extremely short, in comparison to real workload traces—1 million synthetically generated instructions are typically sufficient. Moreover, various studies have demonstrated that statistical simulation is capable of identifying a region of interest in the early stages of the microprocessor design cycle while considering both performance and power consumption [Genbrugge and Eeckhout 2008; Eyerman et al. 2006; Joshi et al. 2006b]. As such, the important application for statistical simulation is to cull a large design space in limited time in search for a region of interest. The limitation of statistical simulation though is that it generates synthetic traces rather than synthetic benchmarks; synthetic traces, unlike synthetic benchmarks, cannot be executed on execution-driven simulators or real hardware.

Sorenson and Flanagan [2002] evaluate various approaches to generating synthetic address traces using locality surfaces. Wong and Morris [1998] use the hit-ratio in fully associative caches as the main criterion for the design of synthetic workloads. They also use a process of *replication* and *repetition* for constructing programs to simulate a desired level of locality of a target application.

7.2 Constructing Synthetic Benchmarks

Early synthetic benchmarks, such as Whetstone [Curnow and Wichman 1976] and Dhrystone [Weiker 1984], were hand coded to consolidate application behaviors into one program. Several approaches [Ferrari 1984; Curnow and Wichman 1976; Sreenivasan and Kleinman 1974] have been proposed to construct a synthetic workload that is representative of a real workload under a multiprogramming system. In these techniques, the characteristics of the real workload are obtained from the system accounting data and a synthetic set of jobs are constructed that places similar demands on the system resources.

Hsieh and Pedram [1998] developed a technique to construct assembly programs that, when executed, exhibit the same power consumption signature as the original application.

The work most closely related to our approach is the one proposed by Bell and John [2005]. They present a framework for the automatic synthesis of miniature benchmarks from actual application executables. The key idea of this technique is to capture the essential structure of a program using statistical simulation theory and generate C-code with assembly instructions that accurately model the workload attributes, similar to the framework proposed in this paper. Our performance cloning technique improves the usefulness of this workload synthesis technique by developing microarchitecture-independent models to capture data stream locality and control flow predictability of a program into synthetic workloads so that the synthetic clones can be used across microarchitectures.

7.3 Workload Synthesis in Other Computer Systems

Approaches to generate synthetic workloads have been investigated for the performance evaluation of I/O subsystems, file system, networks, and servers. The central idea in these approaches is to model the workload attributes using a probability distribution, such as Zipf's law and, binomial distribution and to use these distributions to generate a synthetic workload. For example, Chen and Patterson [1993] developed an approach to generate parameterized self-scaling I/O benchmarks that can dynamically adjust the workload characteristics according to the performance characteristic of the system being measured.

7.4 Workload Characterization Techniques

Weicker [1990] used characteristics such as statement distribution in programs, distribution of operand data types, and distribution of operations, to study the behavior of several stone-age benchmarks. Saveedra and Smith [1996] characterized Fortran applications in terms of the number of various fundamental operations and predicted their execution time. Source code-level characterization has not gained popularity because of the difficulty in standardizing and comparing the characteristics across various programming languages. There has been research on microarchitecture-independent locality and ILP metrics. For example, locality models researched in the past include working set models, least recently used stack models, independent reference models, temporal density functions, spatial density functions, memory reuse distance, and locality space (see for example, Conte and Hwu [1990], Lafage and Seznec [2001], and Chandra et al. [2005]). Generic measures of ILP based on the dependency distance in a program have been used by Noonburg and Shen [1994] and Dubey et al. [1994].

7.5 Reducing Simulation Time

Statistical sampling techniques [Conte et al. 1996; Wunderlich et al. 2003] have been proposed for reducing the cycle-accurate simulation time of a program. The SimPoint project [Sherwood et al. 2002] proposed basic block distribution

analysis for finding program phases that are representative of the entire program. The SimPoint approach can be considered orthogonal to our approach, because one can generate a synthetic benchmark clone for each phase of interest. Iyengar et al. [1996] develop a concept of fully qualified basic blocks and apply it to generate representative traces for processor models with infinite cache. This work was later extended [Iyengar and Trevillyan 1996] to generate address traces to match a particular cache miss rate. Ringenberg et al. [2005] developed intrinsic checkpointing, which is a checkpoint-implementation technique that loads the architectural state of a program by instrumenting the simulated binary rather than through explicit simulator support. The synthetic benchmarks obtained in this paper have the advantages over these simulation points of (1) hiding the functional meaning of the original application, and (2) having even shorter dynamic instruction counts.

8. CONCLUSIONS

In this paper we explored a workload synthesis technique that can be used to clone a real-world proprietary application into a synthetic benchmark that can be made available to architects and designers. The synthetic benchmark clone has similar performance/power characteristics as the original application, but generates a very different stream of dynamically executed instructions. By consequence, the synthetic clone does not compromise the proprietary nature of the application. In order to develop a synthetic clone using pure microarchitecture-independent workload characteristics, we develop memory-access and branching models to capture the inherent data locality and control-flow predictability of the program into the synthetic benchmark clone. We developed synthetic benchmark clones for a set of benchmarks from the SPEC CPU2000 integer and floating point, and MiBench and MediaBench benchmark suites, and showed that the synthetic benchmark clones exhibit good accuracy in tracking design changes. Also, the synthetic benchmark clone runs more than five orders of magnitude faster than the original benchmark and thus significantly reduces simulation time on cycle-accurate performance models.

The technique proposed in this paper will benefit architects and designers to gain access to real-world applications, in the form of synthetic benchmark clones, when making design decisions. Moreover, the synthetic benchmark clones will help the vendors to make informed purchasing decisions, because they now have the ability to benchmark a processor using the synthetic benchmark clone as a proxy of their application of interest.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive and valuable feedback.

REFERENCES

BELL, R. AND JOHN, L. 2005. Improved automatic test case synthesis for performance model validation. In *Proceedings of the International Conference on Supercomputing*. 111–120.

BELL, R. AND JOHN, L. 2006. Efficient power analysis using synthetic testcases. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 110–118.

BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the Annual International Symposium on Computer Architecture*. 83–94.

BURGER, D. AND AUSTIN, T. 1997. The SimpleScalar toolset, version 2.0, University of Wisconsin-Madison, Computer Sciences Department Tech. Rep. #1342.

CHANDRA, D., GUO, F., KIM, S., AND SOLIHIN, Y. 2005. Predicting inter-thread cache contention on a chip multiprocessor architecture. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 1–12.

CHEN, P. AND PATTERSON, D. 1993. A new approach to I/O performance evaluation—Self-scaling I/O benchmarks. In *Proceedings of the ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*. 340–351.

COLEMAN, C. 1998. Using Inline Assembly with Gcc. See www.cs.virginia.edu/~clc5q/gcc-inline-asm.pdf.

COLLINS, J., WANG, J., CHRISTOPHER, H., TULLSEN, D., HUGHES, C., LEE, Y., LAVERY. D., AND SHEN, J. 2001. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the Annual International Symposium on Computer Architecture*. 14–25.

CONTE, T. AND HWU, W.-M. 1990. Benchmark characterization for experimental system evaluation. In *Proceedings of the 1990 Hawaii International Conference on System Sciences (HICSS)*. Architecture Track, vol. I. 6–16.

CONTE, T., HIRSCH, M., AND MENEZES, K. 1996. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the International Conference on Computer Design*. 468–477.

CURNOW, H. AND WICHMAN, B. 1976. A synthetic benchmark. *Comput. J.* 19, 1, 43–49.

DUBEY, P., ADAMS, G., III, AND FLYNN, M. 1994. Instruction window trade-offs and characterization of program parallelism. *IEEE Trans. Comput.* 431–442.

EECKHOUT, L. AND DE BOSSCHERE, K. 2001. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 25–34.

EECKHOUT, L., NUSSBAUM, S., SMITH, J., AND DE BOSSCHERE, K. 2003. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*. *23*, 5, 26–38.

EECKHOUT, L., BELL, R., STOUGIE, B., DE BOSSCHERE, K., AND JOHN, L. 2004. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *Proceedings of the Annual International Symposium on Computer Architecture*. 350–361.

EYERMAN, S., EECKHOUT, L., AND DE BOSSCHERE, K. 2006. Efficient design space exploration of high performance embedded out-of-order processors. In *Proceedings of the design, Automation and Test in Europe*. 351–356.

FERRARI, D. 1984. On the foundations of artificial workload design. In *Proceedings of ACM SIGMETRICS Conference on Modeling and Measurement of Computer Systems*. 8–14.

GENBRUGGE, D. AND EECKHOUT, L. 2008. Memory data flow modeling in statistical simulation for the efficient eploration of microprocessor design spaces. *IEEE Trans. Comput. 57*, 1, 41–54.

HAUNGS, M., SALLEE, P., AND FARRENS, M. 2000. Branch transition rate: A new metric for improved branch classification analysis. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 241–250.

HENNING, J. 2000. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Comput. 33*, 7, 28–35.

HSIEH, C. AND PEDRAM, M. 1998. Microprocessor power estimation using profile-driven program synthesis. *IEEE Trans. Comput. Aided Design Integrated Circ. Sys. 17*, 11, 1080–1089.

IYENGAR, V. AND TREVILLYAN, L. 1996. Evaluation and generation of reduced traces for benchmarks. Tech. Rep. RC20610. IBM Research Division. T. J. Watson Research Center.

IYENGAR, V., TREVILLYAN, L., AND BOSE, P. 1996. Representative traces for processor models with infinite cache. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 62–73.

JOSHI, A., EECKHOUT, L., BELL, R., AND JOHN, L. 2006a. Performance cloning: A technique for disseminating proprietary applications as benchmarks. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 105–115.

JOSHI, A., YI, J., BELL, R., EECKHOUT, L., JOHN, L., AND LILJA, D. 2006b. Evaluating the efficacy of statistical simulation for design space exploration. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 70–79.

LAFAGE, T. AND SEZNEC, A. 2001. Choosing representative slices of program execution for microarchitecture simulations: A preliminary approach to the data stream. *Kluwer International Series in Engineering and Computer Science Series, Workload Characterization of Emerging Computer Applications*. 145–163.

LUK, C., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V., AND HAZELWOOD, K. 2005. PIN: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 190–200.

NOONBURG, D. AND SHEN, J. 1994. Theoretical modeling of superscalar processor performance. In *Proceedings of the International Symposium on Microarchitecture.* 52–62.

NUSSBAUM, S. AND SMITH, J. 2001. Modeling superscalar processors via statistical simulation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 15–24.

OSKIN, M., CHONG, F., AND FARRENS, M. 2000. HLS: Combining statistical and symbolic simulation to guide microprocessor design. In *Proceedings of the Annual International Symposium on Computer Architecture*. 71–82.

RINGENBERG, J., PELOSKI, C., OEHMKE, D., AND MUDGE, T. 2005. Intrinsic checkpointing: A methodology for decreasing simulation time through binary modification. In *Proceedings of the International Symposium on Performance Analysis of System and Software.* 78–88.

SAVEEDRA, R. AND SMITH, A. 1996. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Trans. Comput. Syst. 14*, 4, 344–384.

SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 1–13.

SimPoint Website. http://www-cse.ucsd.edu/~calder/simpoint/

SKADRON, K., MARTONOSI, M., AUGUST, D., HILL, M., LILJA, D., AND PAI, V. 2003. Challenges in computer architecture evaluation. *IEEE Comput. 36*, 8, 30–36.

SORENSON, E. AND FLANAGAN, J. 2002. Evaluating synthetic trace models using locality surfaces. In *Proceedings of the IEEE International Workshop on Workload Characterization*. 23–33.

SREENIVASAN, K. AND KLEINMAN, A. 1974. On the construction of a representative synthetic workload. *Commun. ACM*. 127–133.

SRIVASTAVA, A. AND EUSTACE, A. 1994. ATOM: A system for building customized program analysis tools. Tech. Rep. 94/2, Western Research Lab, Compaq (Mar.).

STOUTCHININ, A., AMARAL, J., GAO, G., DEHNERT, J., JAIN, S., AND DOUILLET, A. 2001. Speculative prefetching of induction pointers. In *Proceedings of Compiler Construction 2001*, *European Joint Conferences on Theory and Practice of Software*. 289–303.

WEICKER, R. 1990. An overview of common benchmarks. *IEEE Comput. 23*, 12, 65–75.

WEICKER, R. 1984. Dhrystone: A synthetic systems programming benchmark. *Communi. ACM*. 1013–1030.

WUNDERLICH, R., WENISCH, T., FALSAFI, B., AND HOE, J. 2003. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the Annual International Symposium on Computer Architecture*. 84–95.

WONG, W. AND MORRIS, R. 1998. Benchmark synthesis using the LRU cache hit function. *IEEE Trans. Comput. 37*, 6, 637–645.

WU, Y. 2002. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 210–221.

Received March 2007; revised August 2007; accepted January 2008