# The Return of Synthetic Benchmarks

Ajay M. Joshi*, Lieven Eeckhout**, and Lizy K. John*

*ECE, The University of Texas at Austin
**ELIS, Ghent University, Belgium
{ajoshi, ljohn}@ece.utexas.edu, leeckhou@elis.ugent.be

## Abstract

*This paper describes a framework, BenchMaker, for constructing parameterized, scalable, synthetic benchmarks from a set of hardware-independent program characteristics. We show that with a suitable choice of a few inherent program characteristics related to the instruction mix, instruction-level parallelism, control flow behavior, and memory access patterns, it is possible to generate a synthetic benchmark whose performance directly relates to that of a real-world application. The parameterized nature of this framework enables the construction of synthetic benchmarks that allow researchers to explore a wider range of the application behavior space, even when no benchmarks yet exist. We evaluate the applicability and the usefulness of BenchMaker for studying the impact of program characteristics on performance and how they interact with processor microarchitecture.*

## 1. Introduction

Estimating and comparing the performance of computer systems has always been a challenging task faced by computer architects and researchers. One of the classic and most popular techniques to measure the performance of a computer system is to characterize its behavior when executing a representative workload. Typically, the representative workload is a benchmark program or a set of benchmark programs that is believed to be representative of typical applications that could be executed on the computer system. Since the early days of computer development, benchmark programs have evolved from simple hand-coded synthetic benchmarks, such as Whetstone [CURN76] and Dhrystone [WEIC84], to standardized benchmark suites such as SPEC CPU, SPECjbb, EEMBC, TPC, etc.

Although the advent of standardized benchmark suites has streamlined the process of performance comparison between different computer systems, architects and researchers face several challenges when using benchmarks in industry product development and academic research:

- *Benchmarks only represent a sample of the application behavior space* – The application programs that are being run on computer systems constantly evolve, and given the diversity of these application domains, benchmark programs only represent a sample of the performance spectrum. There may be several application characteristics for which standardized benchmarks do not exist. This makes it difficult to project the processor performance for such applications.

- *Benchmarks are rigid and measure performance at a single-point* – A benchmark typically measures the performance of a computer system for a set of workload characteristics. This may make it difficult to get statistical confidence in the evaluation. Typically, it is not easy to vary the benchmark characteristics to understand whether a performance anomaly is an artifact of the benchmark or a characteristic of the underlying system. Moreover, the rigid nature of benchmarks makes it difficult to isolate and study the effect of individual workload characteristics on performance.

- *Benchmark suites are costly to develop, maintain, and upgrade* – Typically, architects and researchers use prevailing benchmarks to make processor design decisions. However, it is known that as emerging applications evolve, benchmark characteristics drift with time and an optimal design using benchmarks of today may not be optimal for applications of tomorrow. This problem has been aptly described as: "Designing tomorrow's microprocessors using today's benchmarks built from yesterday's programs" [WEIC97] [YI06]. Therefore, it is important for architects and researchers to analyze the effect of workload behavior drift on microprocessor performance. However, developing new benchmark suites and upgrading existing benchmark suites is extremely time-consuming and by consequence very costly. Therefore, it is not possible for the benchmark development process to keep pace with the rate at which new applications emerge.

- *Benchmarks that are standardized are open-source where as applications of interest are typically proprietary* – Being able to run benchmarks on a variety of platforms requires that these benchmarks can be compiled to each of these platforms. As a result, industry-standard benchmarks such as SPEC CPU are typically open-source benchmarks that are easily portable across platforms. However, they may not be representative for the applications of interest. One solution to this problem would be to use the applications of interest as benchmarks. Unfortunately, in many cases, applications of interest cannot be distributed to third parties because of their proprietary nature.

One of the approaches for addressing these challenges is to complement application benchmark suites with synthetic benchmarks. An approach to automatically generate synthetic benchmarks can help in: (1) constructing synthetic benchmarks to represent application characteristics for which benchmarks do not (yet) exist, (2) isolating individual program characteristics into microbenchmarks, (3) altering hard-to-vary benchmark characteristics, and (4) serving as proxies for proprietary applications of interest. The aim of this paper is to propose a framework, called BenchMaker, for constructing such synthetic benchmarks whose code properties can be easily altered.

Recently, the computer architecture research community has recognized the need for rigorous benchmark generation techniques [SKAD03] and expended some effort in developing synthetic benchmarks that can mimic the behavior of real world applications. The primary motivation of recent research work in developing synthetic benchmarks has been to reduce simulation time of longer-running benchmarks and to enable sharing of proprietary applications as benchmarks. The central idea of these proposed techniques is to replicate detailed workload characteristics of a real world application into a synthetic trace [OSKI00] [NUSS01] [EECK04], or a synthetic benchmark program [BELL05] [JOSH06].

However, each of these approaches has at least one shortcoming that limits its ability to study the application behavior space by varying program characteristics. Firstly, in most of these approaches [NUSS01] [EECK04] [BELL05] [JOSH06], an application is characterized using detailed workload characteristics – a statistical flow graph captures the control flow behavior of a program and characteristics such as instruction mix, register dependency distribution, control flow predictability, and memory access pattern – that are measured at the granularity of a basic block. This involves specifying a large number of probabilities to describe a workload, which is highly impractical when using these frameworks for exploring workload behavior spaces by varying workload characteristics. Secondly, although some of the approaches for generating synthetic workloads [OSKI00] [EECK01] show that applications can be modeled using a limited of number of program characteristics, they use a combination of microarchitecture-dependent and microarchitecture-independent program characteristics. Microarchitecture-dependent characteristics, such as branch misprediction rate and cache miss rate, do not capture the inherent program characteristics and make it difficult to explore the entire application behavior space independently from the underlying hardware. Finally, a shortcoming of some of these techniques [OSKI00] [NUSS01] [EECK00] is that they generate synthetic workload traces, precluding their use on real hardware, execution-driven simulators, and RTL models.

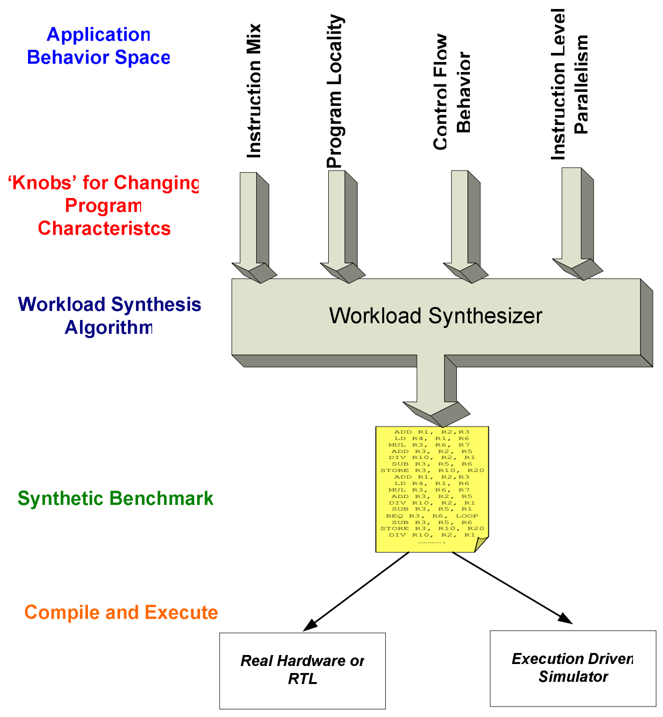
The approach proposed in this paper overcomes these shortcomings. Unlike prevailing approaches to generating synthetic benchmarks, the BenchMaker framework that we propose makes it possible to alter inherent workload characteristics of a program by varying a limited number of key microarchitecture-independent program characteristics in a synthetic benchmark – changing the workload behavior is done by simply 'turning knobs'. This ability to vary program characteristics makes it possible to efficiently explore the application behavior space. Specifically, we make the following contributions in this paper:

1) We show that it is possible to fully characterize a workload with just a few microarchitecture-independent workload characteristics. This is much more efficient than the collection of distributions that need to be specified in prevailing workload synthesis techniques. In addition, unlike previous approaches, the use of microarchitecture-independent characteristics makes it possible to explore the entire application behavior space.
2) We implement this approach into a framework, called BenchMaker, which is parameterized to generate synthetic benchmarks. The generation of synthetic benchmarks instead of traces makes it possible to use these parameterized synthetic workloads on real hardware, execution-driven architectural simulators and low-level cycle-accurate RTL simulators.
3) We evaluate the usefulness of the BenchMaker framework by demonstrating its applicability to three different areas: (a) Studying the effect of inherent workload characteristics on performance, (b) Studying the interaction of microarchitecture-independent workload characteristics with the microarchitecture features of a processor, and (c) Accounting for workload drift during microprocessor design.

The remainder of this paper is structured as follows. In Section 2, we provide an overview of the proposed technique for constructing synthetic benchmarks from program characteristics and describe features of the BenchMaker framework that we propose in this paper. In Section 3, we describe our simulation environment, machine configuration, and the benchmarks used to evaluate the BenchMaker framework. In Section 4, we evaluate the BenchMaker framework by demonstrating how it can be used to generate synthetic benchmarks that exhibit similar behavior to SPEC CPU2000 Integer benchmarks. In Section 5, we demonstrate the application of the BenchMaker framework to three challenging problems. In Section 6, we summarize related research work and prior art. Finally, in Section 7, we conclude with the key results from this paper.

## 2. BenchMaker Framework

Figure 1 illustrates the approach used by the BenchMaker framework that we propose in this paper for generating synthetic benchmarks from a set of microarchitecture-independent program characteristics. The program characteristics measure the inherent properties of the program that are independent from the underlying hardware. Collectively, these characteristics form an abstract workload model. This abstract workload model serves as input to the synthetic benchmark generator. Our intention is to develop a workload model that is simple yet accurate enough for predicting performance trends across the workload space. Keeping the workload model simple makes it possible to not only accurately model the characteristics of an existing workload into a synthetic benchmark, but also provides the ability to conduct 'what-if' studies by varying program characteristics. In the following sections we describe the workload characteristics that serve as input to the synthetic workload generator and we also describe the algorithm used for modeling these characteristics into a synthetic workload.


Application Behavior Space
Instruction Mix
Program Locality
Control Flow Behavior
Instruction Level Parallelism
'Knobs' for Changing Program Characteristcs
Workload Synthesis Algorithm
Workload Synthesizer
Synthetic Benchmark
Compile and Execute
Real Hardware or RTL
Execution Driven Simulator

```
ADD R1, R2,R3
LD R4, R1, R6
MUL R3, R6, R7
ADD R3, R2, R5
DIV R10, R2, R1
SUB R3, R5, R6
STORE R3, R10, R20
ADD R1, R2,R3
LD R4, R1, R6
MUL R3, R6, R7
ADD R3, R2, R5
DIV R10, R2, R1
SUB R3, R5, R1
BEQ R3, R6, LOOP
SUB R3, R5, R6
STORE R3, R10, R20
DIV R10, R2, R1
………….
```

**Figure 1.** The BenchMaker framework for constructing synthetic benchmarks.

## 2.1 Workload Characteristics

The characteristics that we propose to drive the benchmark synthesis process are a subset of all the microarchitecture-independent characteristics that can be modeled. However, we believe that our abstract workload model captures (most of) the important program characteristics that potentially impact a program's performance; the results from the evaluation of the synthetic benchmarks in this paper in fact show that this is the case, at least for the benchmarks that we used.

Recall that the key goal of this paper is to show that it is possible to maintain good representativeness and good accuracy with a *limited* number of key workload characteristics. For limiting the number of program characteristics, we capture them at a coarse granularity using average statistics over the entire program. This is in contrast to prior work on synthetic benchmark generation [BELL05] [JOSH06] which models program characteristics at a fine granularity by capturing program characteristics at the basic block level. Although measuring program characteristics at a coarse granularity likely reduces the representativeness of the synthetic benchmarks compared to fine grained characteristics, this is key to enable the flexibility in BenchMaker for generating benchmarks with characteristics of interest. This will enable one to easily vary workload characteristics by 'turning knobs' and make it possible to answer 'what-if' questions. We propose to measure the following workload characteristics at the program level.

**Instruction Mix.** The instruction mix of a program measures the relative frequency of various operations performed in the program; namely the percentage of integer small latency, integer long latency, floating-point small latency, floating-point long latency, integer load, integer store, floating-point load, floating-point store, and branches in the dynamic instruction stream of a program.

**Basic Block Size.** A basic block is a section of code with one entry and one exit point. We measure the basic block size as the average number of instructions between two consecutive branches in the dynamic instruction stream of a program. We assume that the basic block sizes in the program have a normal distribution, and characterize them in terms of the average and standard deviation in the basic block size distribution of a program.

**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 the data dependency distances is to capture a program's inherent ILP. We measure the data dependency distance information on a per instruction basis and summarize it as a cumulative distribution organized in eight buckets: percentages of dependencies that have a dependency distance of 1 instruction, and the percentage of dependency dependencies that have a distance of up to 2, 4, 6, 8, 16, 32, and greater than 32 instructions. Longer dependency distances permit more overlap of instructions in a superscalar out-of-order processor.

**Data Footprint.** We measure the data footprint of a program in terms of the total number of unique data addresses referenced by the program. The data footprint of a program gives an idea of whether the data set fits into the level-1 or level-2 caches.

**Data Stream Strides.** The principle of data locality is well known and recognized for its importance in determining an application's performance. Instead of quantifying temporal and spatial locality by a single number or a simple distribution, our approach for measuring the data locality of a program is to identify the streams (regular sequences of arithmetic progressions) in a program, their length, and how they intermingle with each other. Once these stream attributes have been correctly identified and instantiated into the synthetic benchmark clone, the resulting program should show similar inherent temporal and spatial locality characteristics [SORE02].

One may not be able to easily identify such stride sequences when observing the global data access stream of the program. This is because several streams co-exist in the program and are generally interleaved with each other. In order to identify the streams and their related attributes, we profile every static load and store instruction to identify the stride with which it accesses data. This is based on the observation [JOSH06] that the memory access pattern appears more regularly when viewed at a finer granularity of static load/store instructions than at a coarser granularity of the global access stream.

In order to capture the data access pattern of a program we measure a distribution of local strides in the program. The local stride value is the difference between two consecutive effective addresses generated by the same static load or store instruction. We measure the local strides in terms of 32-byte block sizes (analogous to a cache line size), i.e., if a local stride is between 0 or 31 bytes, it is classified as stride 0 (consecutive addresses are within one cache line distance), between 32 and 63 bytes as stride 1, etc. We summarize the local stride distance for the entire program as a histogram showing the percentage of memory access instructions with stride value of 0, 1, 2, etc. Figure 2 shows the distribution of the data stride values for the SPEC CPU2000 integer programs. From this figure we observe that for the `bzip2`, `crafty`, `gzip`, and `perlbmk` benchmarks, more than 80% of the local stride references are within a 32-byte block size, indicating very good spatial data locality. The `gcc`, `twolf`, and `vortex` benchmarks only have 60% of local stride values that are within a 32-byte block size, and exhibit moderate spatial data locality. The `vpr` benchmark shows two extremes, with approximately 50% of local strides accessing the same 32-byte block, and the other 50% with extremely large local stride values, indicating a mix of references with extremely poor and extremely high spatial locality. The `mcf` benchmark is an outlier and has very poor data locality, with most of the local stride values being extremely large.

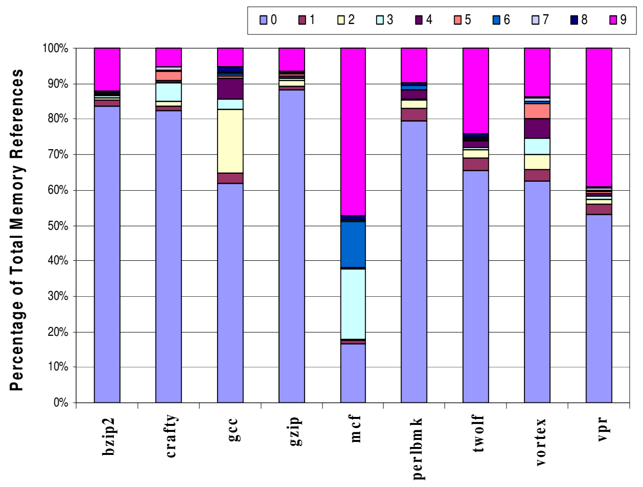


**Figure 2**.Percentage breakdown of local stride values.

The combination of data footprint and the stride value distribution captures the inherent data locality in the program. These two characteristics are typically very difficult to modify in standard benchmarks. In synthetic benchmarks it is easy to fix one of these parameters and study the effect of the other. For example, using BenchMaker, we can easily study the impact of changing stride values while keeping the data footprint the same. Or, if of interest, one can explore the combined effect of varying footprint and access patterns.

**Instruction Footprint.** We characterize the instruction footprint as the total number of unique instructions referenced by the program. The instruction footprint of a program gives an idea of whether the data set fits into the level-1 or level-2 caches. The instruction footprints of all the programs that we studied are very small (`gcc` has the largest instruction footprint) and do not stress the instruction cache.

**Branch Transition Rate.** In order to capture the inherent branch behavior in a program, the most popular microarchitecture-independent metric is to measure the percentage of taken branches in the program or the taken rate for a static branch, i.e., fraction of the times that a static branch was taken during the complete run of the program. Branches that have a very high or low taken rate are biased towards 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. The predictability of the branch depends more on the sequence of taken and not-taken directions than just the taken rate.

Therefore, in our control flow predictability model we also measure an attribute called transition rate, due to [HAUN00], for capturing the branch behavior in programs. 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, the branches with low transition rates are always biased towards either taken or not-taken. It has been well observed that such branches are easy to predict.

Also, the 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 measure a distribution of the transition rate of all static branches in the program. When generating the synthetic benchmark clone we ensure that the distribution of the transition rates for static branches in the synthetic stream of instructions 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.

**Summary.** To summarize the above discussion, the abstract model characterizing a workload consists of 40 numbers in total, as shown in Table 1. Collecting only 40 workload statistics results in a much more compact representation of a workload; compared to the previous benchmark synthesis approaches [BELL05][JOSH06], where most of these statistics are separately measured for every basic block resulting in typically several thousands of numbers to characterize a workload. Consequently, the BenchMaker framework has 40 'knobs' that can be controlled to efficiently explore the application behavior space.

**Table 1.** Microarchitecture-independent characteristics that form an abstract workload model.

| Category | Num. | Characteristic |
|---|---|---|
| instruction mix | 8 | percentage of integer short latency<br>percentage of integer long latency<br>percentage of floating-point short latency<br>percentage of floating-point long latency<br>percentage of integer load<br>percentage of integer store<br>percentage of floating-point load<br>percentage of floating-point store |
| instruction level parallelism | 8 | register-dependency-distance – 8 distributions for register dependencies. Register dependency distance equal to 1 instruction, and the percentage of dependency dependencies that have a distance of up to 2, 4, 6, 8, 16, 32, and greater than 32 instructions. |
| data locality | 1<br>10 | data footprint<br>distribution of local stride values |
| instruction locality | 1 | instruction footprint |
| branch predictability | 10<br>2 | distribution of branch transition rate<br>average and std. dev in basic block size |

## 2.2 Synthetic Benchmark Construction

We now describe the algorithm that is used to generate a synthetic benchmark from the abstract workload model. The synthetic benchmark generator constructs a synthetic benchmark by modeling all the microarchitecture-independent workload characteristics described in 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 [Bell05]. However, the memory and branching model is replaced with a microarchitecture-independent model, as described later in this section. The clone generation process comprises of five sub steps – generating the synthetic program spine using instruction mix and basic block analysis, incorporating memory accessing pattern modeling, modeling branch predictability, register assignment, and code generation.

### 2.2.1 Generating Program Spine

A normal distribution function based on the average basic block size and its standard deviation is used to generate a linear chain of basic blocks. This linear chain of basic blocks forms the spine of the synthetic benchmark program. We use the maximum instruction footprint of the program as a guideline to decide the length of the spine for each program. The chain of basic blocks can be made arbitrarily long in order to generate a large footprint that will stress the instruction cache. After the spine has been instantiated, each basic block is populated using the instruction mix characteristics. Also, each operand in each instruction is assigned a value based on the dependency distance distribution. This is used in a later stage when register assignment is being performed.

### 2.2.2 Modeling Memory Access Pattern

For each memory access instruction in the synthetic benchmark we assign a stride value from the stride distribution function. The load or store instruction is modeled as a bounded stream of circular references, i.e., each memory access walks through an entire array using the stride value assigned to it and then restarts from the first element of the array. The length of each array is simply the ratio of the data footprint of the program and the total number of static load or store instructions in the program. This makes it possible to easily alter the data footprint of the program while maintaining the same stride distribution. Since the maximum number of unique stride values in the program is restricted to 10, we do not need a large number of registers to store the stride values.

### 2.2.3 Modeling Branch Predictability

For each static branch in the spine of the program we assign a transition rate based on the specified transition rate distribution. 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. A counter is incremented on each iteration count and a modular operation is used to decide whether the branch is taken or not-taken.

### 2.2.4 Register Assignment

In this step we use the dependency distances that were assigned to each instruction to assign registers. 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.

### 2.2.5 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 (for modeling the memory access patterns) and 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 towards 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. The entire program is executed in a loop whose value can be controlled to control the dynamic instruction count of the program. This value is tuned to ensure that the synthetic clone's performance, cycles per instruction (CPI), converges to a stable value.

## 3. Experiment Setup

In all of our experiments we use the `sim-alpha` simulator [DESI01] from the `SimpleScalar` Tool Set [BURG97]. The `sim-alpha` simulator is an execution driven performance model that has been validated against the superscalar out-of-order Alpha 21264 processor. In order to measure the abstract workload characteristics of a program we used a modified version of the `sim-safe` simulator.

**Table 2.** SPEC CPU programs, input sets, and simulation points used in study.

| Benchmark | Input | SimPoint(s) |
|---|---|---|
| | *SPEC CPU2000 Integer* | |
| `bzip2` | graphic | 553 |
| `crafty` | ref | 774 |
| `Eon` | rushmeier | 403 |
| `Gcc` | 166.i | 389 |
| `gzip` | graphic | 389 |
| `Mcf` | ref | 553 |
| `perlbmk` | perfect-ref | 5 |
| `twolf` | ref | 1066 |
| `vortex` | lendian1 | 271 |
| `vpr` | route | 476 |
| `gcc` | expr | 8, 24, 47, 51, 56, 73, 87, 99 |
| | *SPEC CPU95 Integer* | |
| `gcc` | expr | 0, 3,5,6,7,8,9,10,12 |

In our experiments we use the integer benchmarks from the SPEC CPU2000 benchmark suite. In most of our experiments we use one 100M-instruction simulation point selected using `SimPoint` [SHER02]. However, when comparing programs from two generations of SPEC CPU benchmark suites we use multiple simulation points. All the SPEC CPU2000 Integer benchmark programs were compiled on an `Alpha` machine using the native Compaq cc `v6.3-025` compiler with `-O3` compiler optimization. The SPEC CPU95 benchmark program, `gcc`, was compiled using a native circa 1995 compiler, `gcc 2.6.3`. Table 2 summarizes the benchmarks and the simulation points that were used in this study.

## 4. Evaluation of BenchMaker Framework

In this section we evaluate the accuracy of the BenchMaker framework by using it to generate synthetic benchmark programs that show similar characteristics as the SPEC CPU2000 benchmark programs. We measure the workload characteristics of the SPEC CPU2000 benchmarks and feed this abstract workload model to the BenchMaker framework.

Figure 3 evaluates the accuracy of BenchMaker for estimating the pipeline instruction throughput measured in instructions-per-cycle (IPC): this is done by comparing the IPC for the actual benchmark compared to the IPC for the synthetic benchmark. We observe that the synthetic benchmark performance numbers tracks the real benchmark performance numbers very well. The average IPC prediction error is 14% and the maximum error is observed for `mcf` (19.9%).

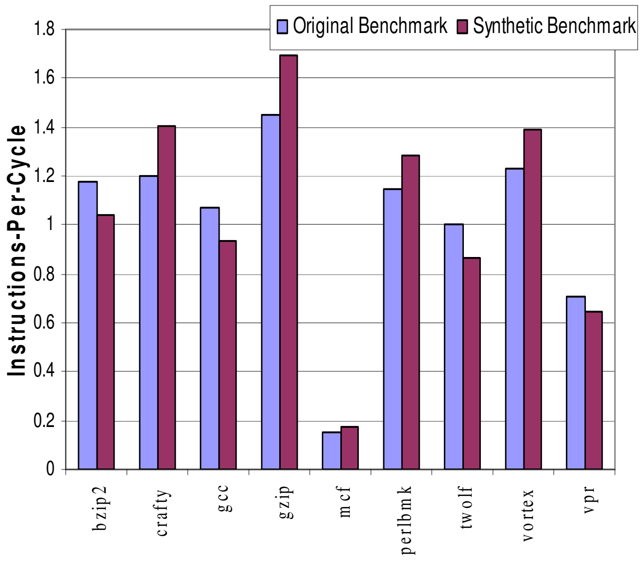


**Figure 3.** Comparison of Instructions-Per-Cycle (IPC) of the actual benchmark and its synthetic version.

Figure 4 shows similar results for the L1 D-cache performance: the number of L1 D-cache misses per one thousand instructions is shown on the vertical axis for the various benchmarks. Again, the synthetic benchmark numbers track the real benchmark numbers very well. The maximum error in predicting the number of L1 cache misses-per-1K instructions is observed for `mcf` for which the difference between the real and the synthetic benchmark is 9 misses-per-1K-instructions (or less than 4% in relative terms). We obtain similar results for the L2 cache performance. All of the benchmarks except for `mcf` and `vpr` have a negligibly small miss rate at the L2 cache level; `mcf` shows 120 L2 misses-per-1K-instructions, and `vpr` shows 8 L2 misses-per-

1K instructions. The synthetic benchmark accurately tracks this trend, and shows 114 and 5 L2 misses-per-1K instructions respectively for `mcf and vpr` benchmarks. Also, the L1 instruction cache miss rate is negligible for all programs, with `gcc` having the highest miss rate of 1.3%.

Figure 5 evaluates the accuracy of BenchMaker for replicating the branch behavior of a real benchmark into a synthetic benchmark. Here again, we observe that the synthetic versions of the benchmark track the real benchmark numbers very well. One particularity to note here is that the branch prediction rates are always higher for the synthetic benchmarks than for the real benchmarks. This suggests that some of the difficult-to-predict branch sequences in the program are not captured in the synthetic benchmark. The branches in the synthetic benchmark tend to be relatively easier to predict than is the case for the original benchmark.

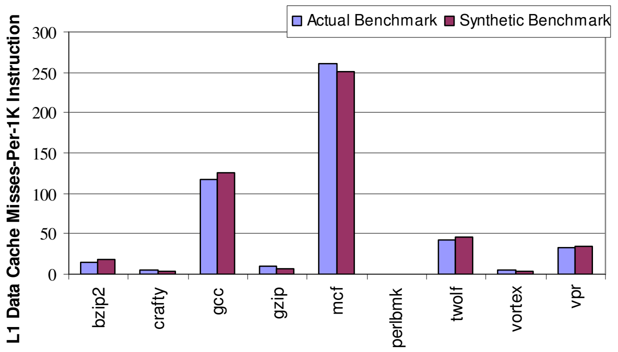


**Figure 4.** Comparison of the number of L1 D-cache misses-per-1K-instructions for the actual benchmark and its synthetic version.

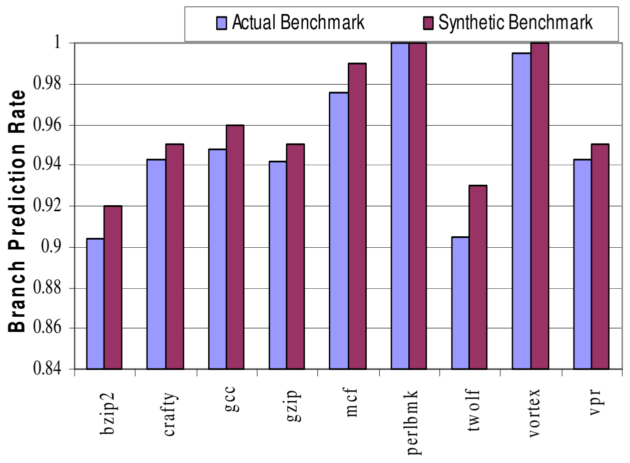


**Figure 5.** Comparison of the branch prediction rate for the actual benchmark and its synthetic version.

## 5. Applications of BenchMaker Framework

### 5.1 Program Behavior Studies

In order to demonstrate the usefulness of the BenchMaker framework we show how it can be applied for studying workload behavior and its interaction with the microarchitecture. It is extremely difficult to conduct comparable 'what-if' studies using a set of standardized benchmarks because their characteristics form an essential part of the benchmark application and cannot be easily altered. On the contrary, using BenchMaker, it is possible to easily generate a benchmark program from a limited list of characteristics.

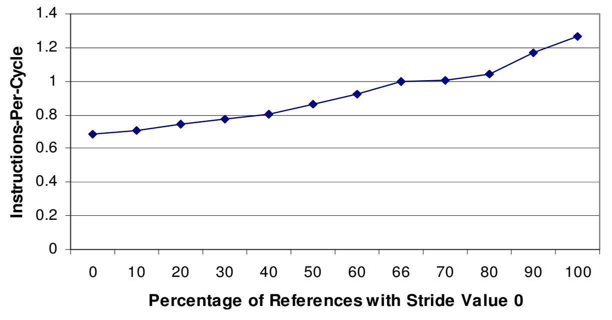
We generate a synthetic benchmark using the average of all the characteristics across the SPEC CPU Integer benchmark programs. The synthetic benchmark, *AvgSynBench*, modeling the average characteristics shows a pipeline throughput of 1.1 IPC on the Alpha 21264 processor. In our study we use the characteristics of this benchmark as our baseline characteristics and alter them to study the effect of each program characteristic on performance, their interaction with each other, and their interaction with the microarchitecture.

#### 5.1.1 Impact of Individual Program Characteristics on Performance

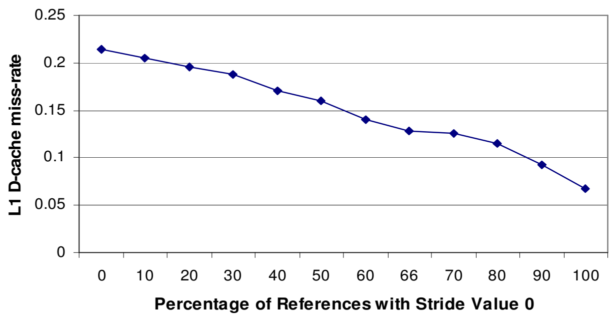
In this section we use BenchMaker to study the impact of data locality and control flow predictability by varying memory access patterns and branch transition rates, respectively.

Figure 6 shows how the change in percentage of references with zero strides (subsequent executions of the same static memory operations access memory within a 32-byte block size) affects IPC and L1 D-cache miss rate. We observe that as the percentage of references with zero stride varies from 0 (no accesses to the same cache line) to 100 (all executions of the same static memory operation access the same cache line), the IPC of the program linearly increases. Interestingly, the drop in L1 data cache miss rate is also linear with the increase in percentage of references with stride value 0. This suggests that if all other characteristics remain constant, the L1 data cache miss rate and IPC have an almost perfect negative linear correlation (-0.99).

Next we study how the branch transition rate affects performance. Recall, that the branch transition rate of a program is measured as a distribution. We experimented with a number of random combinations of distribution of transition rates. We observed that with these random combinations, the branch prediction rate varies between 0.99 and 0.82, and correspondingly the variation in IPC was a factor of 1.61 (61% dip in performance if branch prediction rate falls to 0.82).



**(a)** Impact on IPC of the percentage of references with zero stride value



**(b)** Impact on L1 D-cache miss rate of the percentage of references with zero stride value

**Figure 6.** Studying the impact of data spatial locality by varying the local stride pattern.

Based on these studies we can conclude that the BenchMaker framework is a useful tool for isolating and studying the behavior of individual program characteristics and their impact on performance.

### 5.1.2 Interaction of Program Characteristics

In our abstract workload model we characterize the data locality of a program by measuring its data footprint (which is an indicator for temporal locality) and the distribution of local stride pattern (which is an indicator for spatial locality).

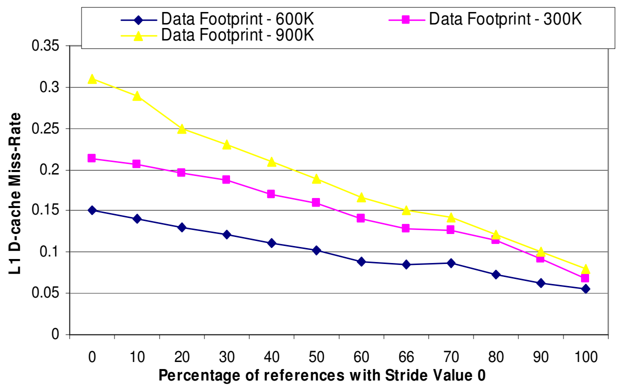


**Figure 7.** Interaction of local stride distribution and data footprint program characteristics.

In this section we analyze how the local stride distribution pattern and the data footprint of a program interact with each other. Figure 7 shows the effect of changes in percentage of references with zero strides for three different data footprints. From this graph we observe that for larger footprints, we see a steeper fall in L1 D-cache miss rate as the percentage of references with stride value 0 increases. For the case where 100% of the references access the same cache line, the footprint does not seem to have an impact on the L1 D-cache miss rate.

### 5.1.3 Interaction of Program Characteristics with Microarchitecture

A benchmark synthesis framework is not only useful for isolating and studying the impact of program characteristics on performance, but is also an invaluable tool to understand how program characteristics interact with microarchitectural structures. For example, BenchMaker can be used to find a combination of program characteristics that interact poorly with a given microarchitecture. More in particular, automatically generating benchmarks that 'stress' the microarchitecture can give insight into critical program-microarchitecture interactions. The 'stress' benchmarks can help in exposing performance anomalies and understanding the limitations of a given microarchitecture.

As an example, in order to find a benchmark that stresses the branch predictor, we generated a number of synthetic benchmarks that contain randomly generated distributions of transition rates. Interestingly, the transition rate distribution that resulted in the lowest prediction rate was the case where 100% of the branches have a transition rate between 90% and 100%. In this configuration, every branch in the synthetic benchmark continuously toggles between taken and not-taken directions. This sequence of branches heavily stresses the Alpha 21264 branch predictor (which is a tournament branch predictor that chooses between local and global history to predict the direction of a given branch): it achieves a branch prediction rate of only 0.82. Similarly, this approach can be extended to stress-test different microarchitectural structures for performance, power, energy and temperature studies, see [JOSH08].

## 5.2. Workload Drift Studies

Research work [YI06] has shown that it is important to account for the potential impact of workload drift when designing a microprocessor. This section demonstrates how BenchMaker can be used to study workload drift.

### 5.2.1 Analyzing the impact of benchmark drift

As a first case study, we use the `gcc` benchmark with the `expr` input set from the SPEC CPU95 and SPEC CPU00 benchmark suites. The `gcc-expr95` benchmark shows an IPC throughput of 1.54 on the Alpha 21264; `gcc-expr00` shows an IPC throughput of 1.11. This clearly shows that a new release of the same application program (with the same input) can result in significant performance degradation (36% degradation in the case of `gcc`). To understand this behavior, we now compare the abstract workload model for `gcc-expr95` and `gcc-expr00`. Most of the program characteristics are more or less the same across the two `gcc` versions. Even the local stride values (indicative of spatial locality) exhibit a similar distribution. However, the data footprint (indicative of temporal locality) appears to have increased by a factor of 3. Based on this observation, we constructed a synthetic benchmark with the same characteristics as `gcc-expr95` but with three times its data footprint. This benchmark shows an IPC throughput of 1.19 (an error of only 7.2% compared to IPC of `gcc-expr00)`.

This result demonstrates that BenchMaker can be a useful tool to generate futuristic workloads in the anticipation of changes in program characteristics, and can help in projecting the impact of workload drift on performance.

### 5.2.2 Analyzing the impact of code size increase

Previous characterization studies [PHAN05] have pointed out that although the dynamic instruction count has increased by a factor 100 over the four generations of SPEC CPU benchmark suites, the static instruction count of the programs has not significantly grown. However, in general, the static instruction count of any commercial software application tends to increase with every generation as the application evolves with the advent of new features and functionality. The absence of any benchmarks that stress the instruction cache makes it difficult to analyze the performance impact of an application that could result from code footprints that are substantially larger than available benchmarks. To illustrate the application of BenchMaker to study the impact of potential increase in code size on program performance, we use the *AvgSynBench* benchmark and vary its code footprint. Figure 8 shows different flavors of the *AvgSynBench* benchmark with varying instruction footprints to stress the instruction cache. The graph shows that increases in code size can have a significant impact on performance and must be taken into account if application code size is expected to increase.

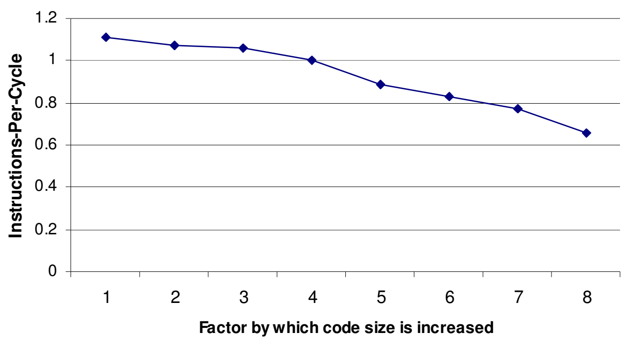


**Figure 8.** Effect of increasing instruction footprint on program performance.

As such, we can conclude that in the absence of any SPEC CPU benchmarks that stress the instruction cache, this is a plausible approach to project the impact of I-cache misses on the performance of an application.

## 6. Related Work

[OSKI00] [EECK00] [NUSSB00] introduced the idea of statistical simulation. 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. [EECK04] improved statistical simulation by profiling the workload attributes at a basic block granularity using statistical flow graphs. Recent improvements include more accurate and detailed memory data flow modeling for statistical simulation [GENB06]. In comparison, the objective of this paper is to keep the workload model simple and yet accurate enough to explore the application behavior space.

[BELL05] extended the concept of statistical simulation 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. [JOSH06] improved the usefulness of this workload synthesis technique by developing microarchitecture-independent models to capture locality and control flow predictability of a program into synthetic workloads. However, similar to statistical simulation, these techniques characterize a program at a fine granularity and make it impractical to easily change program characteristics.

[EECK01][OSKI00] showed that using a combination of analytical and statistical modeling, it is possible to efficiently explore the workload and microprocessor design space. However, this technique uses a combination of microarchitecture-independent and microarchitecture-dependent workload characteristics – limiting the application behavior space that can be explored. The approach proposed in this paper overcomes this shortcoming that it is possible to characterize a workload using only a few microarchitecture-independent workload characteristics – enabling exploration of a wider application behavior space. Also, the construction of synthetic benchmarks instead of synthetic traces makes it possible to run the synthetic benchmark on real hardware and execution-driven simulators.

Several approaches [FERR84] [CURN76] [SREE74] have been proposed to construct a synthetic drive 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. There has been a lot of research on developing 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, locality space, etc., see for example [CONT90] [DENN68] [SEZN00] [SPIR72][CHAN2005]. Generic measures of parallelism based on the dependency distance in a program have been used by [NOON94] and [DUBE94].

## 7. Conclusions

The objective of this paper was to develop a framework that can be used to construct parameterized synthetic benchmarks. One of the key results from this paper is that it is possible to fully characterize a workload by only using a limited number of microarchitecture-independent program characteristics, and still maintain good accuracy. Moreover, since these program characteristics are measured at a program level they can be measured more efficiently and are amenable to parameterization. We implement this approach in a framework called BenchMaker and demonstrate various applications that help in studying program characteristics that are typically difficult to vary in standardized benchmarks. The need for a scientific approach to construct parameterized synthetic benchmarks, to complement standardized benchmarks, has long been recognized by the computer architecture research community, and this work is a significant step towards achieving that goal.

## Acknowledgements

Ajay Joshi was supported by an IBM Fellowship. Lieven Eeckhout is supported by a Postdoctoral Fellowship with the Fund for Scientific Research in Flanders (Belgium) (FWO Vlaanderen). This work is also supported in part through the NSF award numbers 0429806 and 0702694, an IBM Faculty Partnership Award, the UGent-BOF project 01J14407, the FWO project G.0255.08, and HiPEAC.

## References

[BELL05] R. Bell Jr. and L. John. "Improved Automatic Test Case Synthesis for Performance Model Validation", in *Proceedings of International Conference on Supercomputing*, 2005, pp. 111-120.

[BURG97] D. Burger and T. Austin. The SimpleScalar Toolset, version 2.0. University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, 1997.

[CURN76] H. Curnow and B.Wichman. A Synthetic Benchmark. *Computer Journal*, vol. 19(1), pp. 43-49, 1976.

[CONT90] T. Conte and W. Hwu, "Benchmark Characterization for Experimental System Evaluation", *Proceedings of the Hawaii International Conference on Systems Sciences*, vol I, Architecture Track, 1990.

[DESI01] R. Desikan *et al.*, "Measuring Experimental Error in Microprocessor Simulation", *Proceedings of International Symposium on Computer Architecture,* 2001.

[EECK00] L. Eeckhout, K. De Bosschere, and H. Neefs, "Performance Analysis through Synthetic Trace Generation," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pp. 1-6, April 2000.

[EECK04] L. Eeckhout, R. Bell Jr., B. Stougie, K. De Bosschere, and L. John, "Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies," in *Proceedings of International Symposium on Computer Architecture*, pp. 350-361, June 2004.

[EECK01] L. Eeckhout and K. De Bosschere, "Hybrid Analytical-Statistical Modeling for Efficiently Exploring Architecture and Workload Design Spaces", *Parallel Architectures and Compilation Techniques,* pp. 25-34, Sept 2001.

[FERR84] D. Ferrari, "On the foundations of artificial workload design," in *Proceedings of AMC SIGMETRICS* Conference on Measurement and Modeling of Computer Systems, pp. 8-14, 1984.

[HAUN00] M. Haungs et al. "Branch Transition Rate: A New Metric for Improved Branch Classification Analysis," in *Proceedings of International Symposium on High Performance Computer Architecture*, pp. 241-250, Feb 2000.

[HSIE98] C. 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(11), pp. 1080-1089, November 1998.

[IYEN96] V. Iyengar, L. Trevillyan, and P. Bose, "Representative traces for processor models with infinite cache", in *Proceedings of International Symposium on High Performance Computer Architecture*, pp. 62-73, Feb 1996.

[JOSH06] A. Joshi, L. Eeckhout, R. H. Bell Jr., L. K. John, "Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks," *IEEE International Symposium on Workload Characterization*, pp. 105-115, Oct 2006.

[JOSH08] A. Josh, L. Eeckhout, L. John, and C. Isen. Automated Microprocessor Stressmark Generation, in *Proceedings of International Symposium on High Performance Computer Architecture,* Feb 2008.

[KEAT99] K. Keaton and D. Patterson, "Towards a Simplified Database Workload for Computer Architecture Evaluations," in *Proceedings of IEEE*

*Workshop on Workload Characterization*, pp.115-124, 1999.

[KURM03] Z. Kurmas *et al.*, "Synthesizing Representative I/O Workloads Using Iterative Distillation," in *Proceedings of International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp. 6-15, 2003.

[NUSS01] Nussbaum and J.E. Smith, "Modeling Superscalar Processors via Statistical Simulation," in *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pp 15-24, Sept 2001.

[NOON97] D. Noonburg and J. Shen, "A Framework for Statistical Modeling of Superscalar Processor Performance", *Proc. of International Symposium on High Performance Computer Architecture*, 1997, pp. 298-309.

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

[PHAN05] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. "Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites", *Proceedings of the International Symposium on Performance Analysis of Systems and Software,* 2005.

[SAAV96] R. Saveedra and A. Smith, "Analysis of benchmark characteristics and benchmark performance prediction", *Proc. of ACM Transactions on Computer Systems*, vol. 14, no.4, pp. 344-384, 1996.

[SHER02] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior", in *Proceedings of ASPLOS*, pp. 45-57, Oct 2002.

[SKAD03] K. Skadron, M. Martonosi, D. August, M. Hill, D. Lilja, and V. Pai, "Challenges in Computer Architecture Evaluation", IEEE *Computer*, vol. 36(8), pp. 30-36, August 2003.

[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*, pp. 23-33, Nov. 2002.

[SPIR72] J. Spirn and P. Denning, "Experiments with Program Locality", *The Fall Joint Conference*, pp. 611-621, 1972

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

[WEIC84] R. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Communications of the ACM*, pp. 1013-1030, Oct 1984.

[WEIC97] R. Weicker, "One the use of SPEC benchmarks in computer architecture research", *Computer Architecture News,* Mar 1997.

[YI06] J. Yi, H. Vandierendonck, L. Eeckhout, and D. J. Lilja, "The Exigency of Benchmark and Compiler Drift: Designing Tomorrow's Processors with Yesterdays Tools", *International Conference on Supercomputing*, pp. 75-86, June 2006.