Genesys: Automatically Generating Representative Training Sets for Predictive Benchmarking

Reena Panda, Xinnian Zheng, Shuang Song, Jee Ho Ryoo, Michael LeBeane, Andreas Gerstlauer and Lizy K John
The University of Texas at Austin
{*reena.panda, xzheng1, songshuang1990, jr45842, mlebeane*}*@utexas.edu,* {*gerstl, ljohn*}*@ece.utexas.edu*

***Abstract*—Fast and efficient design space exploration is a critical requirement for designing computer systems, however, the growing complexity of hardware/software systems and significantly long run-times of detailed simulators often makes it challenging. Machine learning (ML) models have been proposed as popular alternatives that enable fast exploratory studies. The accuracy of any ML model depends heavily on the representativeness of applications used for training the predictive models. While prior studies have used standard benchmarks or hand-tuned micro-benchmarks to train their predictive models, in this paper, we argue that it is often sub-optimal because of their limited coverage of the program state-space and their inability to be representative of the larger suite of real-world applications. In order to overcome challenges in creating representative training sets, we propose Genesys, an automatic workload generation methodology and framework, which builds upon key low-level application characteristics and enables systematic generation of applications covering a broad range of program behavior state-space without increasing the training time. We demonstrate that the automatically generated training sets improve upon the state-space coverage provided by applications from popular benchmarking suites like SPEC-CPU2006, MiBench, MediaBench, TPC-H by over 11x and improve the accuracy of two machine learning based power and performance prediction systems by over 2.5x and 3.6x respectively.**

I. Introduction

Fast and efficient design space exploration is a critical requirement for designing and optimizing computer systems. But the growing complexity of hardware systems coupled with the rapid pace of software development makes efficient design space exploration challenging. Generally, detailed simulation-based techniques are employed for performing accurate performance/power analysis of programs. However, the high computational cost/run-time of detailed simulators [1] and large workload sizes affect the efficiency of elaborate exploratory studies. Machine learning (ML) based predictive modeling has been explored in several prior research studies [2, 3, 4, 5, 6, 7, 8, 9, 10] as a popular alternative that enables fast design space exploration. ML models have also been adopted in the industry, finding applications in areas such as server-scheduling, cluster maintenance models, operating system scheduling [11, 12] etc. For example, Google uses ML models for making scheduling decisions on their clusters.

A typical supervised learning-based framework employed for computer system modeling is shown in Figure 1. It consists of two phases: training and prediction/testing. The training phase involves learning the inherent behavioral patterns of a set of training applications and creating a predictive model based on the learned patterns. The prediction phase involves using the created ML model for predicting desired metrics for test applications. However, as the model learns patterns

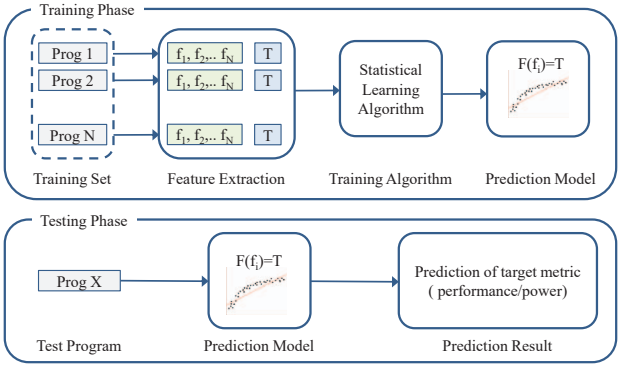

Fig. 1: Machine learning based prediction framework

based on the behavior of training applications, its accuracy depends significantly on the ability of the training set to represent the wide-range of real-world applications that are run on modern computer systems. If the target applications have similar performance behavior as the training applications, the accuracy of the model will be higher and vice versa.

Most prior studies [2, 5, 6, 8, 11, 13] leveraging machine learning for computer design exploration select their training sets from standard benchmarking suites (e.g., SPEC CPU [14, 15], SPECjbb [16], MiBench [17], Implantbench [18], TPC-H [19], etc.), which consist of a *"few"* selected benchmarks representing certain classes of applications. But the number of applications included in such suites is often far from being sufficient for capturing the workload space or training any statistical model effectively. Also, as standard benchmarking suites evolve at a much slower pace as compared to real-world software development, they are not always representative of the entire suite of applications that are run on modern-day computer systems. Furthermore, the significantly long run-times of many standard benchmarks limits several past studies to train their models by running either short subsets of the benchmarks or using reduced input sets (e.g. - Minnespec [20]), which further reduces the representativeness of the training application sets. Few other studies [4, 9, 12] use hand-tuned micro-benchmarks for training their predictive models, however they are harder to write and increase the training time, which limits the number of micro-benchmarks that can be used to have a representative training set.

To overcome these challenges in creating representative training programs for machine learning models, in this paper, we propose ***"Genesys"***, an automatic workload generation methodology and framework that enables systematic generation of applications covering a broad range of the program behavior state-space. Genesys builds on core low-level application characteristics and systematically varies them to

978-1-5090-3076-7/16/$31.00 ©2016 IEEE

generate sets of synthetic benchmarks that can provide targeted coverage of the workload space effectively. By enabling fast and effective coverage of the program state-space, Genesys aims to improve the accuracy of ML models used for design-space exploration of computer systems. In this paper, we demonstrate that using training sets automatically generated by Genesys improves upon the state-space coverage provided by applications from popular benchmarking suites (e.g., SPEC-CPU2006, MiBench, MediaBench, TPC-H) by over 11x. We also show that Genesys improves the accuracy of two machine learning based power and performance prediction systems by over 2.5x and 3.6x respectively. The key contributions of this paper are as follows:

- We propose ***Genesys***, an automatic workload synthesis framework to systematically improve the representativeness and coverage of training application sets.
- We propose a novel metric, ***SpreadRatio***, to quantify and compare the state-space coverage provided by different program sets.
- We show that by controlling 12 key application-specific metrics in a systematic manner, Genesys can improve upon the state-space coverage provided by popular, standard benchmarks like SPEC-CPU2006, MiBench, MediaBench, TPC-H by over 11x.
- We also demonstrate that using targeted synthetic training sets, the accuracy of two machine learning based power and performance prediction models can be increased by over 2.5x and 3.6x respectively.

The rest of this paper is organized as follows: In Section 2, we provide a brief background about ML modeling followed by discussing Genesys's methodology and evaluation in sections 3 and 4 respectively. Section 5 discusses prior work and section 6 concludes the paper.

II. Background

A. *Machine learning based modeling*

ML techniques have been widely explored for architectural performance/power prediction and micro-architectural design-space studies. In this paper, we use the supervised learning-based models, which typically consist of a training and a prediction phase (Figure 1). During the training phase, a set of programs, which constitute the training set, are executed to capture their performance features as well as the target reference metric(s) e.g., performance or power. In [6, 7, 9, 13], these features are obtained from hardware performance events, whereas in [2, 5, 8, 10], the underlying machine configurations are also used. The learning algorithm constructs a predictive model which associates applications' characteristics with the reference metric(s). During the prediction phase, test program features are used as an input to the predictive model, which produces estimates of the target characteristics.

The accuracy and usefulness of any such learning-based approach depends heavily on the training sets that are used for creating the predictive models. A well-formed training set can improve the accuracy and applicability of the predictive models to a wide range of test applications, while an ill-formed training set can affect the statistical model (e.g, cause over-fitting) and lead to inaccuracies during the prediction phase. Ideally, a good training set should consist of:

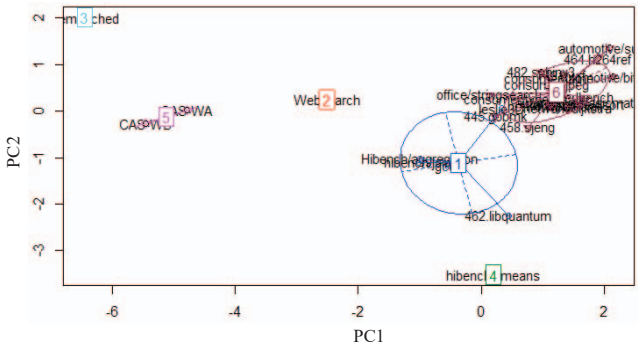

Fig. 2: KMEANS cluster plot

- programs that are representative of the target applications. As a wide-range of applications are run on modern computer systems, it is important that the training programs cover the application design-space sufficiently well. This increases the likelihood of finding programs with similar characteristics during the prediction phase.
- a large number of program instances that provide broader and more denser coverage of the application state space. A training set based on few program instances increases the risk of over-fitting the model with very similar programs.

However, it is challenging to find the right set of applications to create a balanced and broader-coverage training set.

Most prior studies [2, 5, 6, 8, 11, 12, 13] either use standard benchmarks or special, hand-tuned micro-benchmarks [4, 9] to train the predictive models. While standard benchmarking suites usually consist of applications representing particular application domains, they are not always representative of the wide-variety of applications that are run on modern-day computer systems. For example, Figure 2 compares a subset of the SPEC CPU2006, MiBench benchmarks with popular big-data applications (Memcached [21], HiBench [22] and Cassandra [23] running the Yahoo! cloud serving benchmark [24]) using an extensive set of performance metrics (e.g., instructions per cycle (IPC), control flow performance, cache misses, etc.). We can observe from the distinct cluster formations (KMeans clustering-based) that there are significant differences in performance characteristics between the two sets of applications. Also, it is usually difficult to assess any application's low-level behavior by looking at the high-level application, unless you are a domain expert. For example, Phansalkar et al [25] showed that many benchmarks in SPEC CPU2006 suite from different application domains exhibit similar behavior and vice versa. Special, hand-tuned micro-benchmarks are definitely useful in stressing particular program properties, but the process is very tedious and time-consuming which can significantly increase the training time. Finally, the number of program instances that can be created in this manner is usually far from being sufficient for training any statistical model.

B. *Overcoming the limitations*

In order to overcome the challenges in creating a balanced set of programs for training ML models, we propose Genesys, an automatic workload generation methodology and framework that builds upon core low-level program characteristics and enables systematic generation of applications covering a broad range of the program behavior state-space. In the next section, we will describe Genesys's methodology in detail.

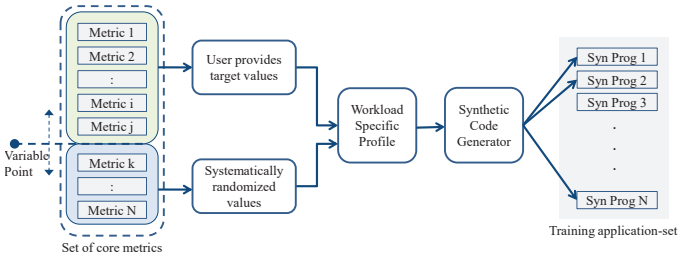

Fig. 3: Genesys's overall methodology and framework

III. METHODOLOGY

Genesys is a workload generation framework (shown in Figure 3) that enables systematic generation of synthetic applications covering a broad range of the application state-space. It builds upon a set of key workload-specific metrics that can be controlled systematically to generate workloads with desired properties. Each workload-specific metric corresponds to a low-level program feature, which defines particular application characteristics, and is available as a user-controllable knob. The user can choose to fix the values of some (or all) core metrics to generate targeted program behavior. For the remaining set of core metrics (if any) whose values are not fixed by the user, Genesys randomizes their values within reasonable bounds in order to achieve well-rounded program state-space coverage around the target behavior. By allowing each workload-specific metric to be controlled using easy-to-use programmable knobs, Genesys allows to create targeted benchmarks with desired program features. Together, the values chosen for the core metrics act as unique profiles for the synthetic workloads. These workload profiles are fed into a code generator algorithm that uses the target metric values to generate a suite of synthetic applications. Together, these applications form a set of unique training programs, which target particular aspects of the program behavior depending upon the choice of the core metrics and can be used to train any supervised ML model.

In this section, we will first present the core workload-specific metrics used by Genesys followed by the workload generation methodology. The core feature set is divided at a high-level into three categories, depending upon the aspects of program behavior that the individual metrics control. The three categories, together with their associated sub-categories and component metrics are shown in Table I and are described below. It should be noted that the set of core metrics used in this paper are not meant to be conclusive, rather they are key metrics that affect different aspects of program behavior. Nevertheless, Genesys's framework is flexible enough to add new metrics to control other aspects of program behavior.

A. *Instruction-level characteristics*

These metrics correspond to the instruction-level behavior of the applications.

1) ***Instruction mix****:* The first metric is the application's instruction mix (IMIX). Genesys categorizes instructions into fractions of loads and stores (memory), control-flow, integer and floating-point instructions. It should be noted that the framework is very flexible and can be easily extended to support specific instruction categories. The target IMIX can be provided as an input to Genesys directly, in which case it generates programs having the desired overall IMIX. Otherwise, IMIX fractions, randomized within bounded ranges, are used to generate the suite of training applications.

TABLE I: Core metrics forming the workload-specific profile

Category	Metrics	Count
Instruction-level Characteristics	1. Instruction mix	5 categories
	2. Instruction count	1
	3. Instruction cache miss rate (ICMR)	1
	4. Instruction level parallelism (ILP)	32 bins
Control-flow Characteristics	5. Average basic block size	1
	6. Branch transition rate (BTR)	1
	7. Branch misprediction rate	1
Memory-access Characteristics	8. Data footprint	1
	9. Regular/irregular behavior	1
	10. Spatial locality stride bins	32 bins
	11. Temporal locality bins	8 bins
	12. L1/L2 Data cache miss rates	2

2) ***Instruction count****:* The second metric that we consider is instruction count (IC), which controls the static instruction footprint of the application. IC can be provided by the user directly or estimated automatically based on the target instruction cache miss rate (ICMR, metric 3). If the ICMR metric is provided, Genesys determines the number of static instructions to instantiate in the synthetic benchmark to achieve the target ICMR. An initial estimate of the number of static instructions is made based on the assumption of a default instruction cache (Icache) size/configuration (32KB, 64 byte blocks, 2-way set-associativity). This serves as an initial estimate only, the final static code size is further tuned to achieve the target ICMR.

3) ***Instruction-level parallelism****:* Instruction-level parallelism (ILP) is an important determinant of an application's performance. Tight producer-consumer chains in program sequences limit ILP and performance because of dependency-induced serialization effects. Genesys models ILP by controlling the dependencies between instructions in the application sequence using the dependency distance metric. Dependency distance is defined as the total number of instructions in the dynamic instruction stream between the production (write) and consumption (read) of a register/memory location. We classify dependency distance into 32 bins (values varying between 1 to 32 and higher), where each bin represents the fraction of instructions having that particular dependency distance. The desired dependency distance can be provided as an input to Genesys or automatically randomized (within bounds) to generate training programs with varying degrees of ILP.

B. *Control-flow characteristics*

These metrics affect the program's control-flow behavior.

1) ***Average basic block size****:* Average basic block size is an important metric because it determines the average number of instructions that can be executed in the program sequence without executing any control instructions. This can affect performance significantly depending on the branch predictor performance. Again, this metric could be provided directly as an input or inferred from the ICMR metric (described before) and the fraction of control instructions.

2) ***Branch predictability model****:* We consider two other control-flow metrics: branch transition rate (BTR) and branch misprediction rate. Prior research studies [26] have shown that an application's branch misprediction rate is highly correlated with the transition characteristics of the branch instructions.

The key idea behind this correlation is that higher the transition probability of a branch instruction, the more difficult it is to predict its next direction and vice versa. To model a certain BTR, we choose a set of control instructions to be modeled with high transition probabilities (frequent switching) and the remaining branch instructions to have very low transition probabilities (infrequent switching activity). Similarly, Genesys can also model the transition probability of individual branch instructions in a directly correlated fashion to achieve a target branch misprediction rate (metric 7).

C. *Memory-level characteristics*

This section describes metrics that affect the memory performance (data-side) of applications.

1) ***Data footprint****:* Data footprint metric determines the range of data addresses accessed by the synthetic application during its execution time. This is important because it can determine performance of different levels of caches and memory based on how large the footprint is with respect to the available cache size and memory structure. It controls the size of the memory regions, which are accessed by the synthetic application.

2) ***Memory access regularity****:* This metric determines if the memory accesses made by load/store instructions of an application should have regular or irregular behavior. For irregular memory behavior, Genesys generates load/store instructions that access allocated and initialized memory regions based on a randomly generated sequence. Regular memory behavior is achieved using additional metrics (spatial-temporal locality or L1/L2 cache miss rate metrics) as described below.

3) ***Spatial and temporal locality****:* The principle of data locality and its impact on applications performance is widely recognized. Genesys models regular data memory accesses using simple strided stream classes over fixed-size data arrays, where strides are defined to be the difference between consecutive effective addresses. Strides can be provided directly as an input to Genesys to control spatial locality characteristics (bins representing strides from -1K to 1K in multiples of 64B). Genesys also provides knobs to control the temporal locality (8 bins expressed as powers-of-two from 0 to 128) in the memory accesses. Temporal locality metric controls the number of unique memory accesses between access to the same memory location and affects the achieved cache miss rates as well. Together, the stride and temporal locality bin values are used to generate the sequence of memory addresses.

Genesys can also automatically estimate the strides (offsets) of the load/store instructions based on the target data cache miss rate statistics. This approach is similar to that adopted by Bell et al [27]. The strides for a particular memory access is determined first by matching the L1 hit rate of a load/store, followed by the L2 hit rate. We generate a table that holds the correlation between L1/L2 cache hit rates and the corresponding stride values to be used. We use the target L1/L2 hit rate information along with this table to generate stride values of load and store instructions. By treating all memory accesses as streams and working from a base cache configuration, the memory access model is kept simple.

D. *Workload generation methodology*

The workload synthesis algorithm, based on the metrics discussed before, is as follows:

1) Generate a random number in the interval [0, 1] and select a basic block based on this number and the block's access frequency.
2) Basic block's size is determined to satisfy the mean and standard deviation of the target basic block size.
3) The basic block is populated with instructions based on the IMIX metrics, while ensuring that the last instruction of the basic block is a branch instruction.
4) Every instruction is assigned a dependency distance (i.e., a previous instruction that it is dependent upon) in order to satisfy the dependency distance criterion.
5) Load and store instructions are assigned a stride-offset based on the memory access model described in the previous section (regular or irregular).
6) An X86 test operation is used to set the condition codes that affects the outcome of the conditional branch instruction at the end of each basic block. The "test" operand is controlled to achieve the target BTR metric.
7) Increment the number of basic blocks generated.
8) If target number of basic blocks have been generated, go to step 9, else update the individual metric distributions and go back to step 1.
9) Available architected registers are assigned to each instruction while satisfying the data dependencies established in step 4.
10) The above instruction sequence is generated as a part of two-level nested loops where the inner loop controls the application's data footprint and the outer loop controls the number of dynamic instructions (overall runtime). Every static load or store instruction resets to the first element of the strided memory streams and re-walks the entire stream in the outer loop iterations.

The code generator generates the instruction sequence using C-language with embedded X86-based assembly instructions. An example code snippet is shown in Figure 4. The code generator can be modified to generate instructions for a different ISA. The code is encompassed inside a main header and malloc library calls are used to allocate memory for the data streams. Volatile directives are used for each asm statement and the program is compiled using the lowest compiler optimization level (-O0 with gcc) in order to prevent the compiler from optimizing out the machine instructions.

IV. Evaluation

This section describes our experimental setup and evaluation in detail.

A. *Experimental setup*

All our experiments are conducted on Intel Xeon E5-2430 v2 server class machines with Ivy-bridge micro-architecture based processing cores, three levels of caches (1.5MB L2 and 15MB L3 cache) and 64 GB of main memory. For measuring

```
{
  _asm_ _volatile_("BBL1INS0:add %%ebx,%%ebx")
  _asm_ _volatile_("BBL1INS1:mov 0(%%r8),%%ecx")
  _asm_ _volatile_("BBL1INS2:add %%ebx,%%ebx")
  _asm_ _volatile_("BBL1INS3:mov 0(%%r14),%%ecx")
  _asm_ _volatile_("BBL1INS4:add %%ebx,%%ebx")
  _asm_ _volatile_("BBL1INS5:test $0,%%eax")
  _asm_ _volatile_("BBL1INS6:jz BBL2INS0")
}
```

Fig. 4: Example synthetic code snippet

hardware performance of different applications, we use Linux perf tool [28] that provides an interface to the processor performance counters. Power consumption is monitored using Intel's RAPL counters.

To show the efficacy of Genesys, we compare the synthetic programs generated using Genesys with a training set comprising of benchmarks drawn from several popular benchmarking suites (hereafter referred to as the REAL training set). The REAL training set includes 70 standard benchmarks: 29 benchmarks from the SPEC CPU2006 suite (using ref inputs), 20 benchmarks from MiBench, 10 benchmarks from MediaBench and 11 TPC-H queries.

Details about the synthetic training programs created using Genesys (hereafter referred to as the GEN training sets) is provided in the following sections. Each GEN training program's size is restricted to complete within 1 to 15 seconds on the target machine.

B. *State-space coverage*

In this section, we show how Genesys can be leveraged to automatically create programs with different features leading to a wider coverage of the program state-space. To do so, we compare the program state-space coverage provided by the REAL training set against the GEN training set. For this study, the GEN set consists of 500 synthetic programs created using Genesys. The GEN programs are uniquely generated by using random combinations of individual metric values (chosen systematically within respective metric bounds). It takes roughly a few (∼5-20) seconds to generate each GEN training program and every program completes execution within 1 to 15 seconds on the target machine. Thus, the training time to generate and collect feature sets for the GEN training set is roughly equal to running the 70 programs from the REAL set due to the significantly longer run-times of several REAL benchmarks.

In order to compare the program state-space coverage achieved by either training sets, we define a novel metric called ***SpreadRatio***, which is defined as the ratio of the area of the convex hull envelope of the REAL versus GEN program features. The convex hull [29] of a set S of points in the Euclidean space is defined as the smallest convex set that contains S. The convex hull of a set of points S in n dimensions is the intersection of all convex sets containing S. For N points $p_1, ..., p_N$ in n-dimensions, the convex hull C is given by the expression:

$$C \equiv \sum_{j=1}^{N} \lambda_j p_j : \lambda_j \geq 0 \ \forall j \ and \sum_{j=1}^{N} \lambda_j = 1$$

Based on this definition of a convex hull, let C_{REAL} represent the convex hull of the points covered by the REAL training set and C_{GEN} represent the convex hull of the points covered by the GEN training set. Then, $SpreadRatio$ can be defined using the following expression:

$$SpreadRatio = \frac{Area(C_{GEN})}{Area(C_{REAL})}$$

Next, we compare the state-space coverage provided by the GEN and REAL programs using the *SpreadRatio* metric. To better demonstrate the degree of controllability provided by Genesys, we first compare the GEN and REAL programs using subsets of performance characteristics, followed by using the entire set. Since the number of metrics is large, it is difficult to visualize all the variables simultaneously to draw any meaningful conclusions. Thus, we use statistical data analysis techniques to simplify the comparison. Using a large number of correlated variables tends to unduly overemphasize the importance of a particular property. Therefore, we first normalize the raw data to a unit normal distribution (mean = 0, standard deviation = 1) and then, pre-process them using Principal Component Analysis (PCA) [30]. PCA is an effective statistical data analysis technique to reduce the dimensionality of a data-set, while maintaining most of its original information. PCA transforms the original variables into a set of uncorrelated principal components (PCs). If significant correlation exists between the original variables, then most of the original information will be captured using just the top few PCs.

First, we compare the memory subsystem performance behavior of the GEN and REAL program sets based on the L1 Dcache, Icache, L2 and LLC misses per kilo instruction (MPKI) metrics. Figure 5a and 5b shows the scatterplot of the top 4 PCs respectively. We can observe that applications from the REAL set do not stress the instruction side performance much, because of which the Icache MPKI for the REAL programs is mostly very low. Such behavior is different from the emerging big-data and cloud applications which have been shown to stress the instruction side performance more heavily [31, 32]. Nevertheless, by controlling Genesys's I/D memory-side metrics, it is possible to create programs that stress the instruction and data-side performance to varying degrees. Overall, we can see that GEN programs provide 25.4 times (SpreadRatio = 25.4) higher coverage area than the REAL programs for the first two principal components and 12.9 times (SpreadRatio = 12.9) higher coverage than REAL programs in the PC3 versus PC4 space.

Figure 5c compares the I/D TLB performance of the REAL and GEN programs. The x-axis corresponds to the ITLB MPKI whereas the y-axis corresponds to the DTLB MPKI of the programs. We can see that although the standard benchmarks provide good coverage in terms of DTLB performance, but none of the REAL programs stress the ITLB much whereas the GEN programs provide extensive state-space coverage in terms of both the instruction and data TLB performance. Overall, GEN training set provides 12.8 times higher coverage than the REAL training set (SpreadRatio = 12.8).

Next, we consider the instruction-level performance characteristics (shown in Figure 5d), based on the overall IPC, μOps/instruction, IMIX and ILP (given by dependency-driven pipeline stalls) metrics. We can see that GEN programs provide 8.1 times broader state-space coverage as compared to the REAL programs for the instruction-level metrics as well.

Next, we compare the control-flow performance coverage of the REAL and GEN programs as shown in Figure 5e. We specifically consider the branch misprediction rate, average basic block size, percentage of branch instructions. We can see that the REAL programs have much better branch performance coverage as compared to their cache and TLB performance. But GEN programs still outperform the REAL set by providing 2.4x higher coverage (SpreadRatio = 2.4).

Figure 5f shows the state-space coverage provided by the REAL and GEN training programs in the PC1 versus

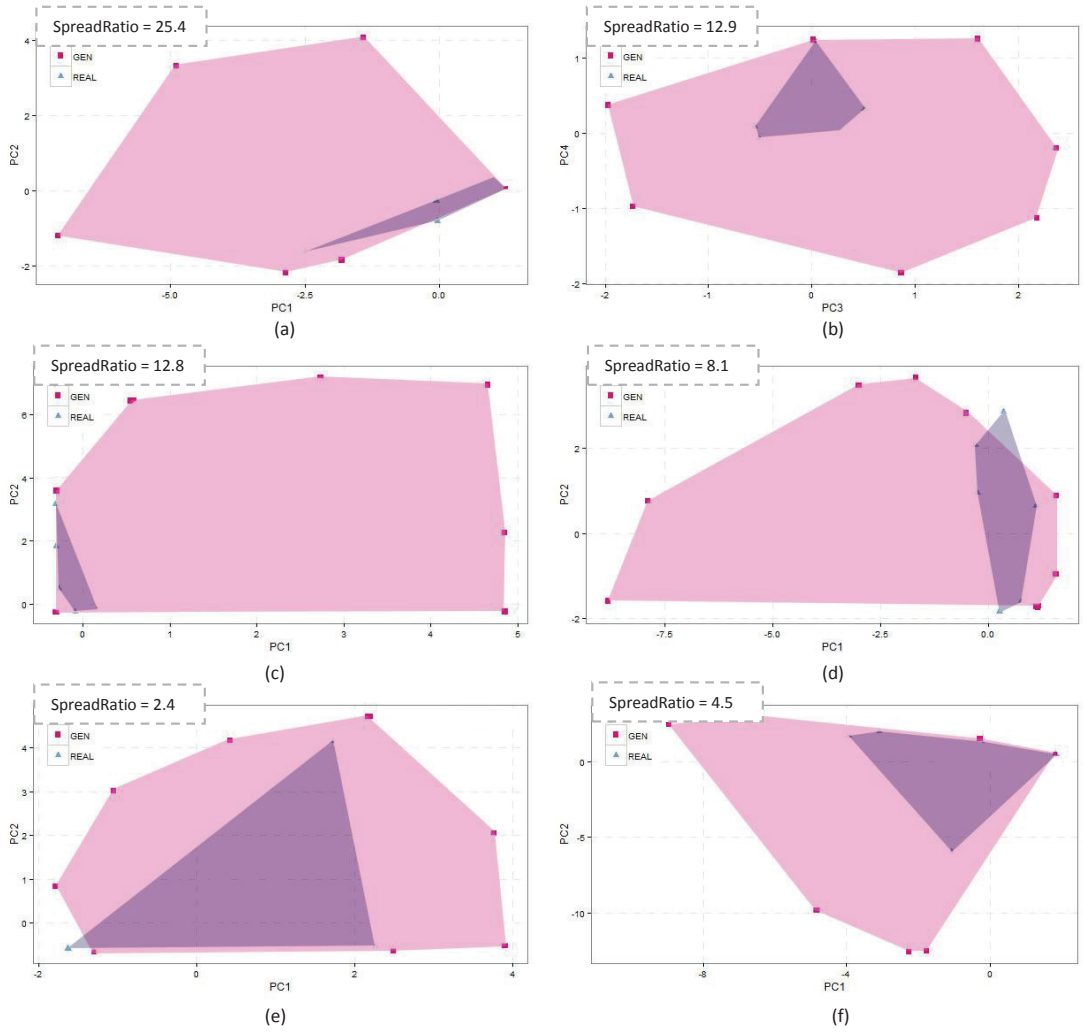

Fig. 5: State-space coverage of REAL and GEN programs using (a) Cache/memory behavior - PC1 vs PC2 (b) Cache/memory behavior - PC3 vs PC4 (c) TLB behavior (d) Instruction-level behavior (e) Control-flow behavior (f) Overall characteristics

PC2 space using all performance features shown in Table II including IPC. Overall, GEN provides 4.5x higher state-space coverage as compared to the REAL set using all the program features. We can thus, conclude that Genesys's methodology of controlling key low-level application metrics allows to easily generate programs with varied performance characteristics.

C. Case-study: performance and power modeling using machine learning

In this section, we demonstrate that Genesys framework can be leveraged to create targeted training sets by increasing the coverage density around regions of interest in the performance state-space. Using such targeted programs sets for training ML models, we demonstrate that significant improvements in prediction accuracy for two power/performance prediction frameworks can be achieved as compared to using standard benchmarks. For this case study, we consider the case of performance and power modeling of a given hardware platform. The performance metric that we predict is the standard IPC of the test program and the target power metric that we predict is the average power consumption of the program in watts. It is important to note that Genesys's framework and methodology is applicable to any supervised learning based ML model and this case study merely serves as an example to demonstrate the utility of Genesys's framework in a particular context.

For this case study, Genesys is used to create targeted GEN training sets (consisting of 60 synthetic benchmarks) having desired performance characteristics. As such, the training time using the GEN training applications is significantly lower as compared to the REAL training set. For building the predictive models, we collect a set of performance features, given in Table II, by running the REAL and GEN training sets together with the reference IPC or power metric. The choice of these metrics is intended to characterize the application's behavior from a micro-architecture perspective. These characteristics are significant predictors of application's overall performance and power. For example, the performance effect of increasing the L1 Dcache size will have a larger impact on applications with a high L1 Dcache miss rate.

*1) **Learning models**:* In this section, we introduce the learning framework that we use for constructing the perfor-

TABLE II: Measured hardware performance features

Performance Features	
μOps/instruction	FP Ops/instruction
branch/instruction	branch miss/instruction
Icache MPKI	Dcache MPKI
ITLB MPKI	DTLB MPKI
L2 MPKI	LLC MPKI

mance and power prediction models. Formally, for a given program i, we denote $x_i \in \mathbb{R}^d$ as its d-dimensional performance feature vector, and $y_i \in \mathbb{R}$ as the corresponding reference IPC or power. The goal of the learning algorithm is to find a function $F : \mathbb{R}^d \rightarrow \mathbb{R}$ such that,

$$F(x_i) \approx y_i \ \forall i \text{ in the training set.}$$

Similar to prior research work [2, 7, 10], in this case-study, we consider only the family of linear functions for F. Under such assumptions, we formulate our problem of finding F as the following optimization problem:

$$\underset{w}{\text{minimize}} \quad \|X^T w - Y\|_2^2 + \lambda \|w\|_2^2 \tag{1}$$

where $X = \left(x_1^T \ldots x_n^T\right)$ is a matrix with each row corresponding to the performance feature vectors x_i of each training program, and $Y = (y_1 \ldots y_n)$ is a vector consisting of the corresponding reference performance or power values. In our case, λ is a tuning parameter chosen to be 1. The optimization problem in equation 1 is called Ridge Regression [33]. While minimizing the sum of square errors of the predictor, it penalizes the model parameter w if its magnitude is large. The $\lambda \|w\|_2^2$ term is often called the l_2-regularizer, which is widely used to prevent over-fitting of the model [33]. Ridge regression can be solved efficiently by using standard gradient descent method [33]. We use the CVXOPT v1.1.7 library in Python as our main computation tool for solving the ridge regression problem.

*2) **Performance prediction results**:* This section presents the results from predicting the performance (IPC) of a set of 18 test applications from SPEC-CPU2006, MiBench, Mediabench and TPC-H benchmarks. The learning model described before is trained using two different training sets: the REAL training set benchmarks excluding the test benchmark and the GEN training set benchmarks containing 60 synthetic programs targeting the test benchmark performance characteristics. Individual sets of GEN training programs are generated for each test application by providing the target performance metrics (e.g., IMIX, branch misprediction rate, cache miss rates, etc. collected using performance counters) as an input to Genesys. Figure 6 compares the IPC prediction accuracy using the two training sets. We can see that using the GEN training set, performance prediction accuracy improves for all test programs except tonto from SPEC-CPU2006 suite, where the accuracy reduces marginally. Overall, creating models using the GEN training set has significantly lower prediction error (3.32%) as compared to the REAL training set (12.26%), which is a 3.69 times improvement in prediction accuracy without increasing the average training time significantly (owing to the shorter runtime of the workload generation algorithm and the synthetic benchmarks). Additionally, generating synthetic benchmarks based on target performance metrics is a one-time effort, and they can be reused for training different prediction models (as shown in the next section), which results in significantly shorter training time due to their shorter run-times.

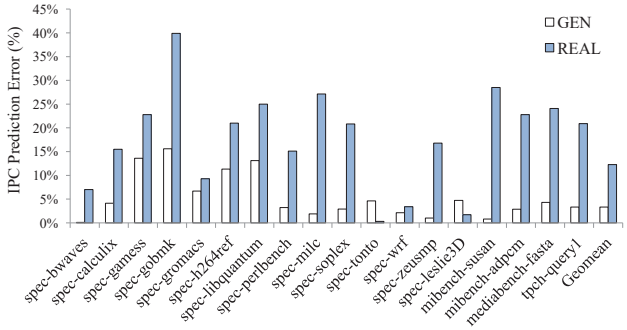

Fig. 6: Performance prediction accuracy modeled using GEN and REAL training sets

*3) **Power prediction results**:* This section presents the results from predicting power consumption of a set of 18 test applications from SPEC-CPU2006, MiBench, Mediabench and TPC-H benchmarks. Again, we use and compare two different training sets: the REAL training set benchmarks excluding the test benchmark and the GEN training set benchmarks generated for the performance prediction experiments before. Figure 7 compares the power prediction accuracy using the two training sets. We can see that power prediction accuracy using GEN set improves for most benchmarks except for five benchmarks libquantum, milc, soplex, leslie3D and zeusmp where the prediction accuracy reduces marginally. On an average though, creating models using the GEN training set has significantly lower prediction error (6.19%) as compared to the REAL training set (15.90%), which is a 2.57 times improvement in prediction accuracy.

V. Related Work

Analytical approaches are being used for performance prediction for several decades [34, 35]. Lee and Brooks [2] used regression-based modeling for micro-architecture design space exploration. Joseph et al. [8] and Ipek et al. [3] used regression-based and artificial neural networks based modeling for creating performance models. Zheng et al [6] used performance counter based measurements on a host machine to predict performance of another system. Genesys is a framework to systematically generate training sets providing both broader and denser coverage of the program state-space and can be applied to all the discussed ML proposals.

Synthetic benchmarking has been applied for benchmark cloning [27, 36, 37, 38] in prior studies. Bell et al. [27] profiled applications at runtime to extract execution related metrics

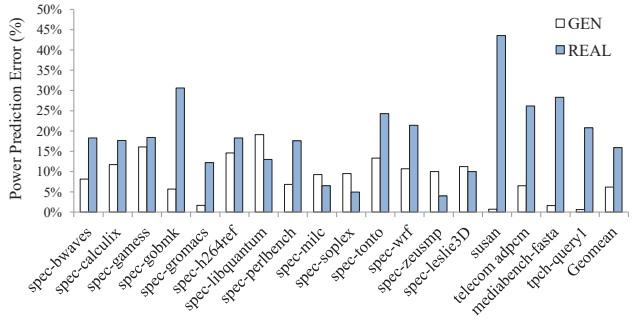

Fig. 7: Power prediction accuracy modeled using GEN and REAL training sets

and created proxy binaries that could be executed on simulators or real hardware. Other studies [37] cloned proprietary benchmarks into synthetic proxies using microarchitecture-independent attributes. Automatic program generation techniques [39, 40] have also been used for functional hardware verification. In contrast, this paper is the first proposal that uses automatic workload generation to systematically improve the state-space coverage of training sets for ML models.

VI. Conclusion

In this paper, we propose Genesys, a novel automatic workload generation framework that enables systematic generation of representative training set applications, providing a wider coverage of program behavior state-space, for effectively training machine learning models. Genesys allows to control a set of key workload-specific characteristics using easy-to-use, programmable knobs and thereby, provides the ability to generate applications targeting specific program properties as well. In order to compare the state-space coverage provided by different sets of applications, we define a novel metric called SpreadRatio, which is based on the area of the convex hull envelope surrounding the program points. We demonstrate that by using automatically generated training sets, it is possible to achieve over 11 times higher state-space coverage than that provided by popular, standard benchmarks such as SPEC-CPU2006, MiBench, MediaBench and TPC-H. We also show that modeling using targeted synthetic training sets improves the predictability/accuracy of two machine learning based power and performance prediction systems by over 2.5x and 3.6x respectively.

VII. Acknowledgement

This research is supported partially by SRC under Task ID 2504 and National Science Foundation (NSF) under grant number 1337393. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or other sponsors.

References

[1] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marss: A full system simulator for multicore x86 cpus," in *DAC*, 2011, pp. 1050–1055.

[2] B. C. Lee and D. M. Brooks, "Illustrative design space studies with microarchitectural regression models," in *HPCA*, 2007, pp. 340–351.

[3] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, "Efficiently exploring architectural design spaces via predictive modeling," *SIGPLAN Not.*, vol. 41, no. 11, pp. 195–206, Oct. 2006.

[4] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance." in *Micro*, 2015, pp. 725–737.

[5] C. Dubach, T. Jones, and M. O'Boyle, "Microarchitectural design space exploration using an architecture-centric approach," in *MICRO 2007*, Dec 2007, pp. 262–271.

[6] X. Zheng, P. Ravikumar, L. K. John, and A. Gerstlauer, "Learning-based analytical cross-platform performance prediction," in *SAMOS*, 2015, pp. 52–59.

[7] W. Lee, Y. Kim, J. H. Ryoo, D. Sunwoo, A. Gerstlauer, and L. K. John, "Powertrain: A learning-based calibration of mcpat power models." in *ISLPED*. IEEE, 2015, pp. 189–194.

[8] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "A predictive performance model for superscalar processors," in *MICRO*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 161–170.

[9] K. Singh, M. Bhadauria, and S. A. McKee, "Real time power estimation and thread scheduling via performance counters," *SIGARCH Comput. Archit. News*, vol. 37, no. 2, pp. 46–55, Jul. 2009.

[10] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "Gpgpu performance and power estimation using machine learning," in *HPCA*, 2015, pp. 564–576.

[11] S.-w. Liao, T.-H. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou, "Machine learning-based prefetch optimization for data center applications," in *SC*, 2009, pp. 56:1–56:10.

[12] A. Negi and P. K. Kumar, "Applying machine learning techniques to improve linux process scheduling," in *TENCON 2005 2005 IEEE Region 10*, Nov 2005, pp. 1–6.

[13] S. Sharkawi, D. DeSota, R. Panda, R. Indukuru, S. Stevens, V. E. Taylor, and X. Wu, "Performance projection of hpc applications using spec cfp2006 benchmarks." in *IPDPS*. IEEE, 2009, pp. 1–12.

[14] "SPEC CPU2006," https://www.spec.org/cpu2006.

[15] "SPEC CPU2000," https://www.spec.org/cpu2000.

[16] "SPECjbb 2005," https://www.spec.org/jbb2005/.

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *WWC-4*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14.

[18] Z. Jin and A. C. Cheng, "Implantbench: Characterizing and projecting representative benchmarks for emerging bioimplantable computing," *IEEE Micro*, vol. 28, no. 4, pp. 71–91, July 2008.

[19] "TPC-H Benchmark Suite," http://www.tpc.org/tpch.

[20] A. J. KleinOsowski and D. J. Lilja, "Minnespec: A new spec benchmark workload for simulation-based computer architecture research." *Computer Architecture Letters*, 2002.

[21] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, Aug. 2004.

[22] S. Huang, J. Huang, J. Dai, T. Xie, and B. H. 0002, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis." in *ICDE Workshops*. IEEE, 2010, pp. 41–51.

[23] "Cassandra," wiki.apache.org/cassandra/FrontPage.

[24] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010, pp. 143–154.

[25] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John, "Measuring program similarity: Experiments with spec cpu benchmark suites," in *IEEE ISPASS*, 2005, pp. 10–20.

[26] M. Haungs, P. Sallee, and M. K. Farrens, "Branch transition rate: A new metric for improved branch classification analysis." in *HPCA*. IEEE Computer Society, 2000, pp. 241–250.

[27] R. H. Bell, Jr. and L. K. John, "Improved automatic testcase synthesis for performance model validation," in *ICS*. New York, NY, USA: ACM, 2005, pp. 111–120.

[28] "Linux perf tool," https://perf.wiki.kernel.org/index.php/Main_Page.

[29] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls," *ACM TRANSACTIONS ON MATHEMATICAL SOFTWARE*, vol. 22, no. 4, pp. 469–483, 1996.

[30] G. Dunteman, *Principal Component Analysis*. Sage Publications, 1989.

[31] R. Panda, C. Erb, M. LeBeane, J. Ryoo, and L. K. John, "Performance characterization of modern databases on out-of-order cpus," in *IEEE SBAC-PAD*, 2015.

[32] R. Panda and L. K. John, "Data analytics workloads: Characterization and similarity analysis." in *IPCCC*. IEEE, 2014, pp. 1–9.

[33] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[34] D. B. Noonburg and J. P. Shen, "Theoretical modeling of superscalar processor performance," in *Micro*, 1994, pp. 52–62.

[35] P. G. Emma and E. S. Davidson, "Characterization of branch and data dependencies in programs for evaluating pipeline performance," *IEEE Transactions on Computers*, vol. 36, no. 7, pp. 859–875, 1987.

[36] L. Eeckhout, K. D. Bosschere, and H. Neefs, "Performance analysis through synthetic trace generation," *IEEE ISPASS*, vol. 0, p. i, 2000.

[37] A. Joshi, L. Eeckhout, R. H. B. Jr., and L. K. John, "Performance cloning: A technique for disseminating proprietary applications as benchmarks." in *IISWC*. IEEE Computer Society, 2006, pp. 105–115.

[38] E. Deniz, A. Sen, B. Kahne, and J. Holt, "Minime: Pattern-aware multicore benchmark synthesizer." *IEEE Trans. Computers*, vol. 64, pp. 2239–2252, 2015.

[39] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek, "Test program generation for functional verification of powerpc processors in ibm." in *DAC*, 1995, pp. 279–285.

[40] S. Ur and Y. Yadin, "Micro architecture coverage directed generation of test programs," in *DAC*, 1999, pp. 175–180.