

Capturing Locality of Reference and Branch Predictability of Programs in Synthetic Workloads

Ajay Joshi [†], Rob Bell ^{†‡}, and Lizy K. John [†]

{ajoshi, ljohn}@ece.utexas.edu, robbell@ibm.com

[†]University of Texas, Austin [‡]IBM Systems and Technology Division

IBM Technical Contact: Alex Mericas, Processor Performance, Systems Group

Abstract

A synthetic workload whose performance correlates well with long-running application programs is of great benefit to the computer architecture community because it reduces simulation time, fosters benchmark sharing by abstracting proprietary codes, and enables analysis of futuristic workloads by altering program characteristics. Recent research [2] [14] has demonstrated that it is possible to automatically construct such synthetic workloads by measuring performance statistics that uniquely characterize an application and modeling them into synthetic workloads. However, in these approaches the synthetic memory access pattern and branch behavior is created to match a target metric e.g. miss-rate, and hence they also reflect machine properties rather than pure program characteristics. Consequently, the synthetic workloads generated from these models may yield large errors when the cache and branch configurations are changed [2].

In this paper we build upon previous research and address its limitations by developing models to incorporate inherent program locality and control flow predictability of an application into synthetic workloads. The primary advantage of our models is that they are based on hardware independent characterization of programs and therefore reflect pure workload or code characteristics. We then use these hardware independent parameters to mimic the memory behavior and control flow predictability of the original program in the synthetic workload. By using a set of SPEC CPU 2000 benchmarks we demonstrate that these models improve representativeness of synthetic workloads by providing better correlation with the original code across a wide range of cache configurations and branch predictors.

1. Introduction

Estimating and comparing the performance of computer systems has always been one of the most challenging tasks faced by computer architects and

researchers. One of the classic and most popular techniques to measure the performance of a computer system is to simulate its behavior when executing a representative workload. The simulator is typically a performance model of the computer system and the workload used in the simulation is some form of a benchmark program that is believed to be representative of some typical or possible applications that could be executed on the computer system. The nature of these benchmarks has evolved from small programs such as microbenchmarks, kernels, hand-coded synthetics, to full-blown applications. Early synthetic benchmarks fell out of favor to applications in the nineteen-eighties because they were easily subjected to unfair optimizations and did not correlate well with real-world applications.

Recently, there has been an interest in the computer architecture community to develop small parametric synthetic workloads that can mimic the behavior of long-running real world applications [5]. Such synthetic workloads have several advantages over application benchmarks:

- 1) Synthetic workloads execute in a fraction of the time as the original application and significantly reduce simulation time.
- 2) The process of creating synthetic benchmarks enables one to isolate and study individual program characteristics that affect a processor's performance.
- 3) Synthetic workloads provide the flexibility to alter workload behavior in anticipation of future workloads that may have characteristics that are different from currently available benchmarks. This aids in design of computers of tomorrow for which benchmarks may not have been developed.
- 4) Synthetic workloads provide an efficient way to abstract proprietary codes and can encourage sharing of application codes between industry and academia.

Consequently, researchers have expended some effort in developing frameworks for automatically constructing synthetic workloads [2] [14] which can mimic the performance of longer-running real-world application programs. The idea behind these proposed techniques is to identify key attributes that affect a program's performance

and then create a synthetic trace or a benchmark with similar performance characteristics. Since the resulting workload is small and the workload characteristics have been statistically modeled, the synthetic workload rapidly converge to a result. Moreover, since the synthetic workload is created from a parameterized model of workload attributes, it is easily possible to generate workloads with a desired set of characteristics.

The works that propose such workload synthesis systems show that the challenges to advance the state-of-the-art in the development of synthetics are to develop models to capture the locality of reference and control flow predictability of programs into synthetic workloads [1][2]. This is because the synthetic workloads generated from models that use microarchitecture-dependent properties to model program properties may yield large errors when the cache and branch configurations are changed from the targeted configuration [2]. The objective of this paper is to overcome this limitation by proposing models and techniques to capture the program properties of locality of reference and control flow predictability into synthetic workloads. The essence of these models is to develop abstractions that capture the inherent locality and control flow predictability property of an application. We show that incorporating these program properties into the synthetic workloads results in the synthetic workload having similar locality and control flow predictability as the original workload.

The contributions of this paper are as follows:

- 1) We develop a model for capturing the instruction temporal and spatial locality of a program into synthetic workloads.
- 2) We study the data memory access pattern of programs and develop an abstraction to model the access patterns of static load and store instructions and how they intermingle with each other. We then use this model to mimic the access pattern of the application into the synthetic workload.
- 3) We develop a model to measure the control flow predictability of a program and use it to synthesize a workload that exhibits similar branch predictability.
- 4) We show that unlike prior works that match synthetic workload behavior to a target prediction, our models retain the inherent memory access patterns and branch predictability of the program. Therefore these models can be used to synthesize workloads that will yield good correlations across a wide range of cache and branch predictor configurations.

The remainder of this paper is organized as follows: In the next section we summarize the related work in the area of constructing synthetic workloads. In section 3 we describe the synthetic workload

construction framework that we used to develop and evaluate the models proposed in this paper. In section 4 we propose models for mimicking instruction locality, data locality, and branch predictability into synthetic workloads. Subsequently, in section 5 we present our results from evaluating the proposed models. Finally, in section 6 we summarize the contributions of our work and outline directions for future work.

2. Related Work

Workload characterization has long been recognized as a critical requirement for designing systems and there have been several investigations into constructing synthetic workloads. The focus of our paper is on developing synthetic workloads for evaluation of the processor and memory subsystem. We therefore describe previous work related to synthesizing of workloads for evaluating processor and memory performance.

Wong and Morris [6] use the hit-ratio in fully associative caches as the main criteria for the design of synthetic workloads. They also use a process of *replication* and *repetition* for constructing programs to simulate a desired level of locality of a target application. There is not clear way to use this technique to simultaneously also model program characteristics, other than its locality, into the synthetic executables.

Hsieh and Pedram [15] developed a technique to construct assembly programs that when executed exhibit the same power consumption signature as the original application. However, the branch and cache access models developed in this study are microarchitecture dependent i.e. capture interaction of program characteristics with machine behavior, and cannot be used to purely model the inherent locality and branch predictability of the program.

Eeckhout et al. [14] proposed a technique for evaluating performance by generating a synthetic trace from the workload. The synthetic trace generator takes as an input a statistical profile of the program features such as basic block size distribution, branch prediction rate, data/instruction cache miss-rate, instruction mix, dependency distances etc., and generates a smaller synthetic trace that has similar program characteristics as the workload statistics described above. The branch prediction rate and cache miss-rates are probabilistically modeled. Their work provides a framework for modeling program behavior in synthetic traces.

Bell and John [1] [2] present a framework for automatic synthesis of miniature benchmarks from actual application executables. The core of this technique is to capture the essential structure of the programs and generate C-code with assembly instructions that accurately model the

characteristics mentioned above. After a workload characterization of the application a sequence of basic blocks is produced that gives good simulation correlation as in [13]. A code generator takes the sequence of instructions and generates C-code with low-level assembly-language instructions. They conclude that the branch prediction models and cache access models are simplistic and further research is needed to develop advanced models. This benchmark synthesis approach provides a framework for investigation of such advanced models.

In this paper our approach is to improve the frameworks proposed in [1] [2] [14] by proposing better memory access and branch predictability models that use attributes that are hardware independent and hence provide better correlations across a wide range of configurations.

3. Synthesis Framework

A synthetic workload is an instruction trace or an executable program whose performance behavior is similar

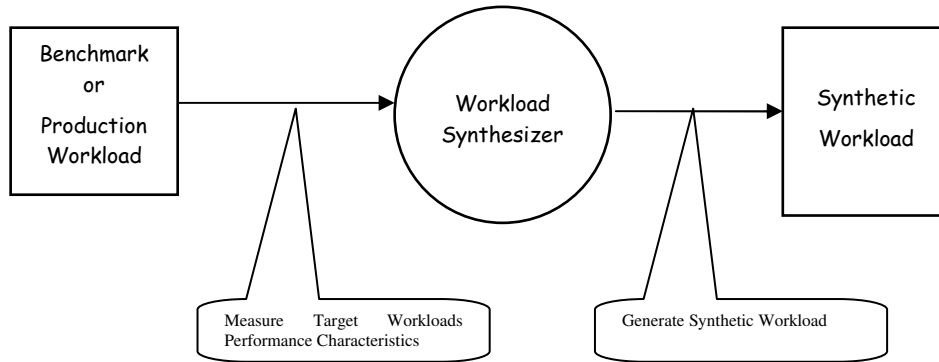


Figure 1. Framework for constructing synthetic workloads

3.1 Synthetic Workload Construction

Our general framework is similar to the one used in statistical simulation systems [12] [13]. In this paper, we developed an enhanced version of HLS++ [13] statistical simulation framework, called SS-HLS++, as our synthetic trace generation environment. As shown in Figure 1, synthetic trace generation using SS-HLS++ comprises of two steps: (1) A statistical profiling of the benchmark program is performed using a workload characterizer to obtain its important performance characteristics, and (2) A synthetic trace is then constructed that has similar statistical workload properties as the original application.

3.1.1 Workload Characterizer

In the first step, we measure the workload characteristics of the application program by profiling the dynamic execution of the program. The workload

to that of the application program from which it was constructed. Trace-driven simulation has the advantage of being fast, whereas using executables overcomes the portability problem of traces and enables simulation of synthetic workloads on real hardware, emulators, and execution-driven simulators. In order to investigate cache access and branch predictability models we use a framework that characterizes an application to extract its key workload characteristics and then automatically generates a synthetic trace with similar workload statistics as that of the application program. Although our evaluation of the models is based on how well they incorporate synthetic behavior memory and branch behavior into traces, the models have been designed in such a way that they are amenable to be used with frameworks, such as [1], that use synthetic traces to construct benchmark executables.

characteristics in this profile can be classified into two types, microarchitecture-independent, e.g. instruction mix, the distribution of read-after-write dependencies, basic block size, and microarchitecture-dependent metrics such as branch predictability, cache miss-rate etc. However, in order to ensure that that we measure pure workload properties it is desirable that all the characteristics are microarchitecture dependent. We used a modified version of SimpleScalar [3] profiler to gather these workload characteristics.

In each of the new model that we propose in this paper we use a microarchitecture-independent characteristic that captures the program property to be incorporated into the synthetic trace. Using this framework such a new characteristic can be easily profiled by augmenting the data structures and information that already exists.

3.1.2 Synthetic Trace Generator

Once a statistical profile has been created, the second step is to generate a synthetic trace. The synthetic trace consists of a number of instructions contained in basic blocks that are linked together into a program flow-control graph, similar to conventional code. However, instead of actual arguments and opcodes, each instruction in the synthetic trace is composed of a set of statistical parameters, such as: instruction type (integer add, floating point divide, load etc.), ITLB/L1/L2 I-cache hit probability, DTLB/L1/L2 D-cache hit probability (for load and store instructions), probability of branch misprediction (for branch instructions), and dynamic dependency distance (to determine how far a consumer instruction is away from its producer). These instructions are probabilistically generated by applying a random number generator to the workload characteristics in the statistical profile. This framework provides us with the flexibility to replace probabilistic cache hits and branch mispredictions with the actual instruction addresses, data addresses, and branch targets. This synthetic trace can then be simulated using a trace driven simulator.

4. Proposed Models

In this section we propose three models for incorporating synthetic instruction locality, data locality, and control flow predictability in traces and benchmark programs. The approach used in these models is to use an attribute to quantify and abstract code properties related to spatial locality, temporal locality, and branch predictability. These attributes are then used to generate a trace or a benchmark with similar properties. If the feature captured the program property very well the resulting performance metrics e.g. cache miss-rate and branch prediction rate will be similar to that of the original application program.

Prior work [2] shows that if a program property is modeled into synthetic workloads using a microarchitecture-dependent feature (e.g. generating a cache access pattern to match a target miss-rate) yields high errors on configurations that are very different from the ones for which they were synthesized. In order to ensure that the generated synthetic workload is valid across a wide range of configurations we choose a microarchitecture-independent attribute to capture the program property to be modeled.

4.1 Instruction Locality of Reference

It has been well observed that the instructions in a program exhibit a property termed *locality of reference*.

The locality of reference is widely observed in the rule of thumb often called the 90/10 rule, which states that a program spends 90% of the execution time only in 10% of the static program code. In order to model this program property in a synthetic trace it is essential to capture the program structure i.e. a map of how basic blocks are traversed and how branch instructions alter the direction of control flow in the instruction stream. During the statistical profiling phase, we propose to capture this information using the statistical flow graphs described in [EECK04-b]. The statistical flow graphs are a profile of the dynamic execution frequencies of each unique basic block in the program along with their transition probabilities to their successor basic blocks. In addition, during the profiling phase, we also annotate each node (representing a unique basic block) in the statistical flow graph with the number of instructions in the basic block and the static instruction address (program counter), of the first instruction in the basic block. Figure 3 shows an example statistical flow graph that is extracted from a program to generate synthetic instruction address traces.

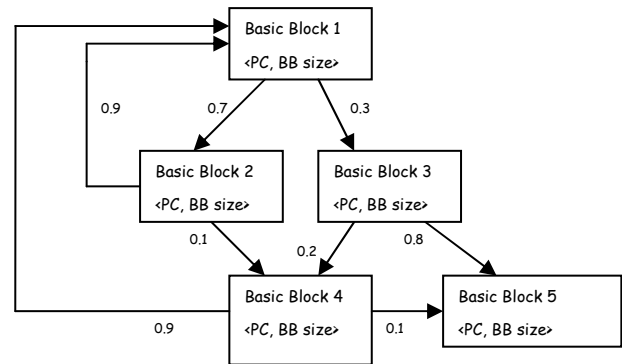


Figure 2. Statistical Flow Graph for capturing control flow structure of the application.

Once the statistical flow graph has been constructed by the statistical profiler, the synthetic trace generation unit probabilistically navigates the graph to create a synthetic instruction address trace. The algorithm used to generate the synthetic instruction address stream is as follows:

The algorithm used to generate the instruction address stream is as follows:

- (1) Generate a random number in the interval $[0, 1]$ and use this value to select a node in the statistical flow graph based on the cumulative distribution function based on the occurrence frequency of each node.
- (2) Using the information of the number of instructions in each basic block and the

instruction address of the first instruction in the basic block, output the instruction addresses for all instructions in that basic block.

- (3) The occurrence count of that node is then decremented.
- (4) Increment count of the total number of instruction addresses generated.
- (5) A cumulative distribution function based on the probabilities of the outgoing edges of the nodes is then used to determine the next node. If the node does not have any outgoing edges, go to step 1.
- (6) If the target number of instructions has not been generated, go to step 2. If the target number of addresses has been generated, the algorithm terminates.

The number of instruction addresses generated should be sufficient enough to warm the cache before steady state measurements can be made. Consequently, to overcome the warm-up issues, the generated address stream should at least be an order of magnitude higher than the size of the cache. Once the statistical flow graph has been traversed sufficient number of time, irrespective of the number of synthetic instruction address traces that are generated, the stream has a locality of reference property that is similar to the original application.

This synthetic instruction address trace can then be simulated using a trace driven cache simulator to obtain the performance estimate of the instruction cache miss-rate. Although cache simulation is inexpensive as compared to cycle-accurate simulation, now-a-days application benchmarks execute hundreds of billions of instructions; the time required for function simulation of the entire program to obtain cache miss-rates for different cache configurations can be very high. Also, the storage space required to store instruction address traces for such application programs can be very high. The synthetic trace generated using this model is very short compared to that of the original program and also captures the essential locality of the program. Therefore, the synthetic trace and can be used as a proxy for the entire program in order to study the instruction locality properties. Also, in order to perform cycle-accurate simulation, the generated synthetic instruction address trace can be annotated to the synthetic instruction stream [14] instead of probabilistically modeling instruction cache misses.

4.2 Data Locality of Reference

The principle of data locality is well known and recognized for its importance in determining an applications performance. Traditionally, data locality is considered to have two important components, temporal locality and spatial locality. Temporal locality is the locality in time and is due to the fact that data items that are referenced now will tend to be referenced soon.

Spatial locality is the locality in space and is due to the program property that when a data item is referenced, nearby items will tend to be referenced soon. Previous work [SORE02] shows that these abstractions of data locality and their measures are insufficient to replicate the memory access pattern of a program. Therefore, instead of quantifying temporal and spatial locality by a single number or a simple distribution, our approach for mimicking the 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 data address trace, the resulting synthetic trace should show similar inherent temporal and spatial locality characteristics.

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 hypothesis (which we validate) that the memory access pattern would appear more regular when viewed at a finer granularity of static memory access instructions (load / store) rather than at the global access stream. Figure 4 shows the percentage of dynamic memory references, for SPEC CPU benchmarks, which will be accounted for if one were to approximate every static memory access instruction in the program with different number of streams. From this chart we observe that for floating point benchmarks *art*, *mesa*, *lucas*, and *swim*, approximating each static memory access instruction as one stream accounts for almost all the dynamic memory references of the program. However, for the other floating point benchmarks, *equake*, *facerec*, *ammp*, *apsi*, *wupwise*, *mgrid*, and *applu* using two streams for representing every static memory reference instruction is a good approximation. For some integer benchmarks, *gcc-166*, *bzip2-program*, *gzip-graphic*, *eon-rushmeier*, and *crafty* two or three streams are required to represent the access pattern of static memory instruction. However, *mcf*, *vpr-route*, and *twolf* benchmark programs either have a very irregular access patterns or a large number of interleaved streams and even 4 streams are insufficient to model the access pattern of all static memory access instructions in the program.

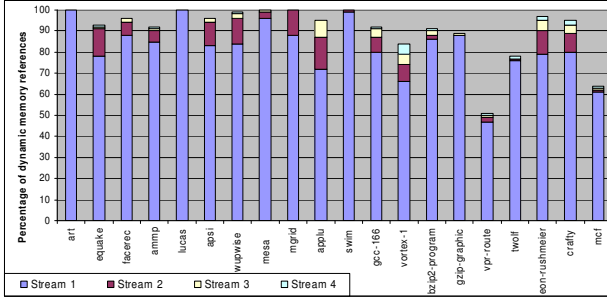


Figure 3. Modeling each static memory instruction using 1 to 4 streams.

Based on the results of these characterization experiments we propose a first-order model to generate a synthetic trace of data address references in the program. In order to correctly model the locality of the data address stream it is important to ensure that the reference streams are correctly interleaved with each other. In order to capture this behavior we build a statistical flow graph of the program as described in section 4.1 and, for each static memory access instruction in every node record the following information during the statistical profiling phase: `<Access type (load/store), Most frequently used stride value, Stride length, Seed address (first data address issued by load/store instruction)>`. The algorithm used to generate a stream of synthetic data address traces is outlined below. Although the basic structure of the algorithm is similar to the one described in section 4.1, it is stated here for the sake of completeness.

- (1) Generate a random number in the interval $[0, 1]$ and use this value to select a node in the statistical flow graph based on the cumulative distribution function based on the occurrence frequency of each node.
- (2) For each load/store instruction in the basic block represented by the selected node:
 - (a) Generate a new reference address (last generated address by this instruction + stride) and output a synthetic entry in the form `<Load/Store, New Address>`. For the first reference, use the seed address as the last address generated for this instruction.
 - (b) Increment counter for the current stream length.
 - (c) Update last generated address counter with the new address. If the stream length has reached the maximum stream length, reset the last generated address counter to the seed address.
- (3) The occurrence count of that node is then decremented.

- (4) Increment count of the total number of data instruction addresses generated.
- (5) A cumulative distribution function based on the probabilities of the outgoing edges of the nodes is then used to determine the next node. If the node does not have any outgoing edges, go to step 1.
- (6) If the target number of instructions has not been generated, go to step 2. If the target number of addresses has been generated, the algorithm terminates.

The resulting synthetic trace can then be simulated through a trace driven cache simulator to obtain the miss-rates. Also, the generated synthetic instruction address trace can be annotated to the synthetic instruction stream [14] instead of probabilistically modeling instruction cache misses. In summary, we have essentially created a congruence class for each static load or store instruction depending on its most frequently used stride and stride length. Each static load/store instruction will iterate through its congruence class. In order to demonstrate that this algorithm can generate an interleaved sequence of streams that result in similar cache miss-rates as the original program, we generate the same working set size and the total number of references as original program (i.e. to produce similar cold start, conflict, and capacity misses). However, in practice, the stream length for each static load/store instruction can be scaled to alter the problem size without altering the access pattern and hence the locality characteristics. In this case the capacity and cold start misses may not be the same as that of the original program. The first-order model outlined above can be extended by modifying step 2 to model larger number of streams to represent one static memory access instruction based on the probability distribution of the streams. This will model will be extended as a part of future work.

4.3 Branch Predictability

In order to incorporate synthetic branch predictability it is essential to understand the property of branches that make them predictable. The predictability of branches stems from two sources: most branches are highly biased towards one direction i.e. the branch is taken or not-taken for 80-90% of the time, and the outcome of some branches that are close together in the source code are dependent or related. Examples of highly biased conditional branches include loops, function calls and returns, exceptional conditions used in user-input validation, system call return values, and data structure initialization.

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. If a static branch had a taken-rate of 50% one can create a synthetic branch behavior such that a branch is taken half the time and not-taken for the other half. But the predictability of the branch depends more on the sequence of taken and not-taken directions than just the taken-rate i.e. a long sequence of taken followed by an equally long sequence of not-taken is easier to predict than a sequence where the taken and not-taken are randomly distributed and the taken-rate is 50%.

Therefore, in our control flow predictability model we also measure an attribute called transition rate, due to Haungs et al. [HAUN00], for capturing the branch behavior in programs. Transition rate is a measure of how often a branch switches between taken and not-taken directions as it is executed. By definition, branches with low transition rates are always biased towards either taken or not-taken and are easy to predict. Also, branches with a very high transition rate always toggle between taken and not-taken directions and are also highly predictable. However, branches that do not have a very high or low transition rates are the branches that transition between taken and not-taken sequences at a moderate rate and may be relatively difficult to predict.

In order to incorporate synthetic branch predictability we characterize the entire application and build a statistical flow graph structure of the entire program as described in section 4.1.1. In addition, every node in the statistical flow graph of the program is annotated with the following: <taken-rate, transition rate, static branch address, branch instruction opcode, instruction address of first instruction in the basic block>. Once the statistical profile has been generated, the workload generator uses the following algorithm used to generate a trace that synthetically models branch behavior:

- (1) Generate a random number in the interval $[0, 1]$ and use this value to select a node in the statistical flow graph based on the cumulative distribution function of the occurrence frequency of each node.
- (2) Use the *transition-rate* of the branch instruction in the basic block represented by the node to determine whether the branch should be *taken* or *not-taken*. If the branch is determined as *taken*, the next node to be selected is determined using the cumulative distribution function based on the

probabilities of the outgoing edges to the *taken* target nodes. Otherwise, the *not-taken* target node is chosen as the next node.

- (3) Using the profile information of the current and next node output a synthetic trace of the format: <PC of first instruction in basic block, PC of static branch instruction, Branch instruction opcode, Target Address of Branch>
- (4) The occurrence count of that node is then decremented.
- (5) Increment count of the total number of synthetic entries generated.
- (6) If the target number of instructions has not been generated, go to step 2. If the target number of addresses has been generated, the algorithm terminates.

5. Results

In this section we evaluate the synthetic trace generation models proposed in the previous section.

5.1 Experiment Setup and Benchmarks

We used benchmark programs and their *reference* input sets from the SPEC CPU 2000 benchmark suite to evaluate the models for generating synthetic locality of reference and control flow predictability. All benchmark programs were compiled on an *Alpha* machine using a native compiler with `-O3` compiler optimization. It is well-known that Floating Point benchmarks stress the data cache and have easy to predict branches (due to larger number of loops). Where as Integer codes have more control flow logic and challenge the instruction cache and branch predictor of a microprocessor. Therefore, we use the Integer benchmarks to evaluate the models for generating

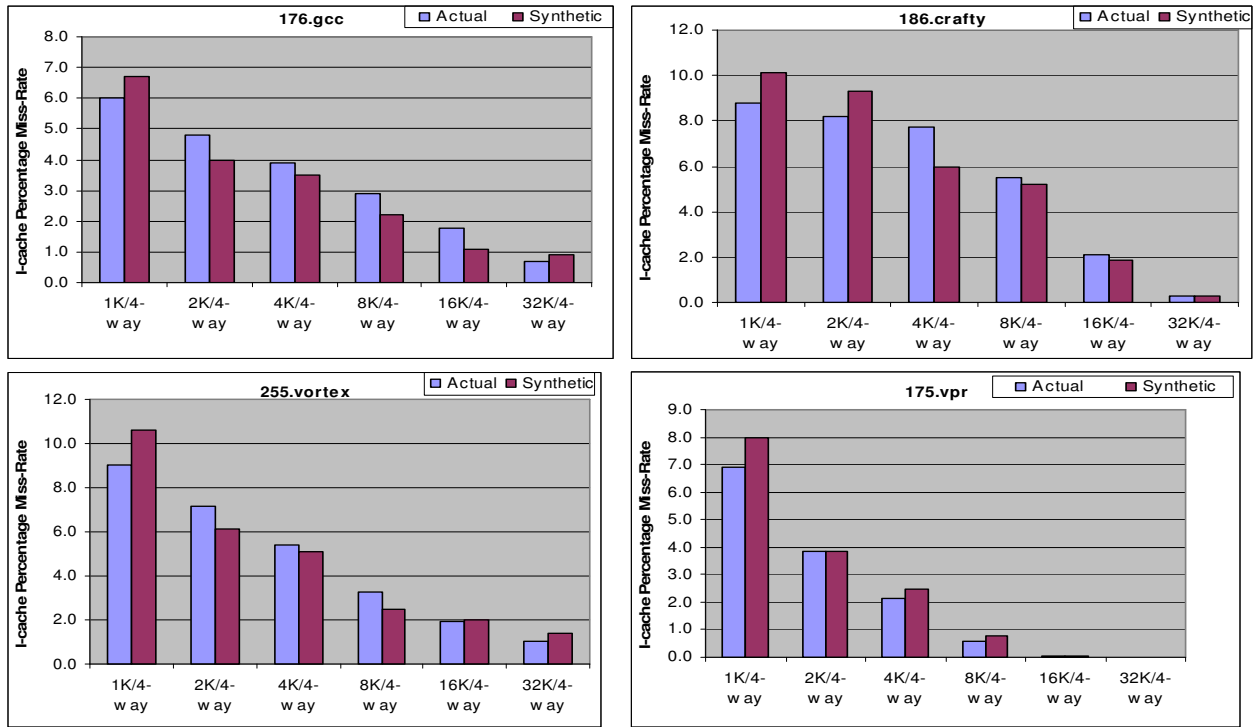


Figure 3. Instruction cache miss-rates from synthetic workload vs. original program

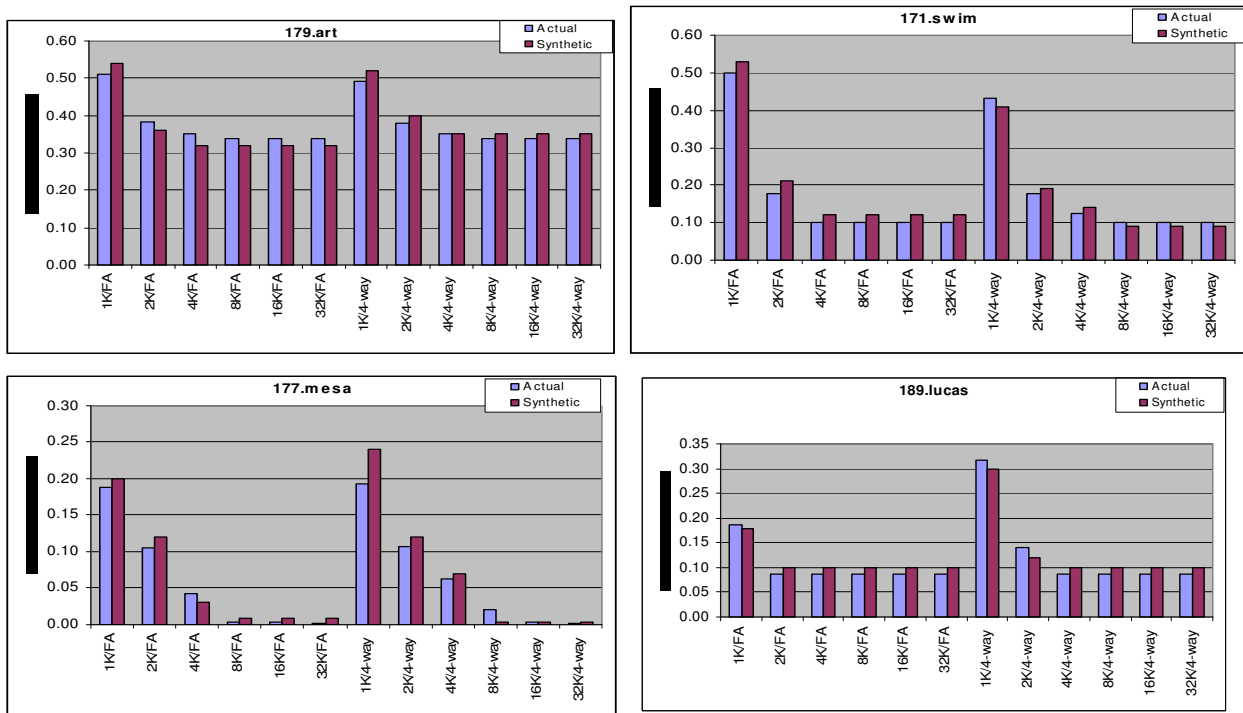


Figure 4. . Data cache miss-rates from synthetic workload vs. original program

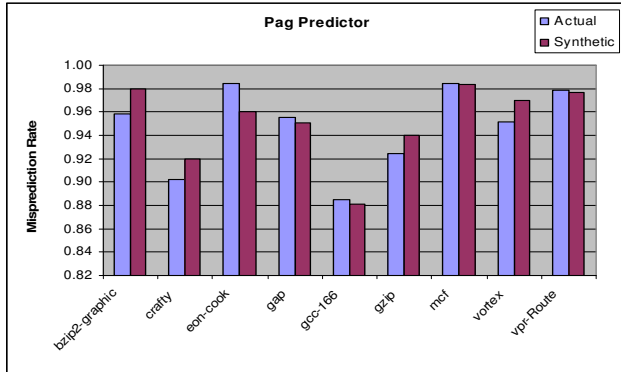
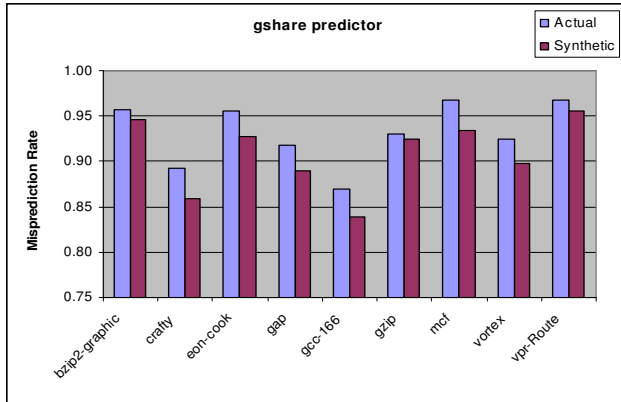
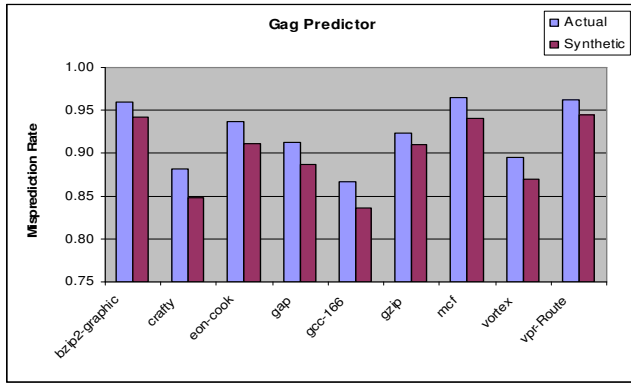


Figure 5. Branch prediction rates from synthetic workload vs. original program

synthetic instruction locality of reference and control flow predictability, and use Floating Point benchmarks to evaluate the model for mimicking the data memory access pattern in programs. In order to restrict the simulation time and study the most representative phase in every program, we used a single representative *SimPoint* [24] comprising of one hundred million instructions. Table 1 shows the benchmarks, their inputs sets, type, and the representative simulation point used in this study.

We used the models described in the previous section to generate synthetic instruction addresses, data address, and branch traces, and simulated these traces using a modified version of *sim-cache* and *sim-bpred* [3] simulators.

Table1. SPEC CPU 2000 benchmark programs used for evaluating proposed models

Program	Input	Type	SimPoint
188.ammmp	ref	FP	108
173.applu	ref	FP	2179
301.apsi	ref	FP	3408
179.art	110	FP	340
183.equake	ref	FP	194
187.facerec	ref	FP	375
189.lucas	ref	FP	35
177.mesa	ref	FP	89
177.mgrid	ref	FP	6
171.swim	ref	FP	2079
168.wupwise	ref	FP	584
256.bzip2	graphic	INT	718
186.crafty	ref	INT	774
252.eon	rushmeier	INT	403
176.gcc	166.i	INT	389
164.gzip	graphic	INT	653
181.mcf	ref	INT	553
255.vortex	lendian1	INT	271
175.vpr	route	INT	476

5.2 Evaluation Results

In order to evaluate the models for incorporating synthetic instruction locality we used 12 L1 data cache configurations with sizes ranging from and data locality we used 12 L1 cache configurations with sizes ranging from 1KB to 32 KB and two difference associativities – fully associative and 4-way set associative. For evaluating the instruction cache access models, we only used the 4-way set associative cache configurations. Although the synthetic instruction and address traces may exhibit the locality properties of the original application (if at least one million instructions are synthesized to ensure convergence of results), they may not yield the same cache miss-rates if their working set size is not as large as that of the original application (due to cold-start and capacity component of the cache misses). Therefore, if the objective of generating the synthetic workload is to create a similar cache miss-rate we it is essential to create a similar working set size as that of the application. In our evaluation we create a working set size for the synthetic workloads to match that of the original workload. This way we demonstrate the locality of the synthetic workload matches that of the application, and it is possible to create similar miss-rates. However, if the objective is to study the access pattern locality characteristics, it is not necessary to create such a large working set using the synthetic workloads and the stream lengths can be scaled proportionately. Figure 3 shows a comparison of the L1 instruction cache miss-rates

obtained from simulation of the original application with those obtained from trace driven simulation of the synthetic instruction address traces. We observe from these graphs that the synthetic instruction trace has small absolute errors and tracks the changes in instruction cache miss-rate.

Figure 4 shows the L1 data cache miss-rates obtained from simulation of original application with those obtained from trace driven simulation of the synthetic address traces. The results show that across a wide range of configurations the miss-rates from the synthetic workload correlates well with the miss-rates obtained from simulation of the original application and show small absolute errors. As shown in Figure 2, the programs used in this study have a highly regular access pattern, and as indicated by the simulation results, the stride based cache access model that we propose captures the data locality with good accuracy.

To evaluate the control flow predictability models we used 5 different branch predictors – gshare, GaG, and PaG, with the default configurations provided by *sim-bpred* [3] for these predictors. Figure 5 shows the comparison of the misprediction rates of the synthetic workloads and the corresponding application from it was synthesized.

From these evaluations we conclude that the models that we propose in this paper create accurate representations of instruction locality, data locality, and branch predictability in synthetic workloads. Also, since these models use hardware independent attributes to characterize the program property, the results show good correlation across a wide range of cache and branch configurations that have been used in this study.

6. Conclusions

Constructing synthetic workloads from application programs for evaluating computer systems is an important and challenging area of research. The cache access and branch prediction models that have been used in synthesis frameworks generate a sequence of addresses or branch directions to match a target cache miss or branch misprediction rate. Consequently, they do not accurately capture pure workload characteristics and can lead to high errors on configurations that are significantly different from the ones for which the workload was synthesized [2]. In this paper we develop models that capture the synthetic instruction locality of reference, data locality of reference, and branch predictability properties in synthetic workloads. Our models are an improvement over what has been previously proposed because they use hardware independent characteristics to measure code properties. This ensures that the memory access patterns and control flow patterns of the original application are retained in the synthetic workload. The synthetic workloads constructed using our proposed models yield good correlations with the original application on a wide range of cache configurations and branch predictors.

Overall, this paper contributes to developing the state-of-the-art in developing techniques for automatically constructing synthetic workloads. The memory access modeling approach used in this paper are promising for capturing behavior of scientific and technical applications that have a regular access pattern, and we feel that future research should investigate extending these models to model synthetic behavior of more

random memory access patterns observed in commercial workloads.

7. References

- [1] Bell, R., Jr. and John, L. The Case for Automatic Synthesis of Miniature Benchmarks. In *Proceedings of the Workshop on Modeling, Benchmarking, and Simulation (MoBS '05)* (Madison, WI, USA, June 4-8, 2005), 88-97.
- [2] Bell, R., Jr. and John, L. Improved Automatic Testcase Synthesis for Performance Model Validation. In *Proceedings of the International Conference on Supercomputing (ICS '05)* (Cambridge, MA, USA, June 20-22, 2005), 111-120.
- [3] Burger, D. and Austin, T. "The SimpleScalar Tool Set, Version 2.0," *ACM Computer Architecture News*, (June 1997), 13-25.
- [4] Lilja, D., "Measuring Computer Performance," Cambridge University Press, New York, NY, 2000.
- [5] Skadron, K., Martonosi, M., August, D., Hill, M., Lilja, D., and Pai, V., "Challenges in Computer Architecture Evaluation," *IEEE Computer*, 36, 8, (Aug. 2003), 30-36.
- [6] W. Wong and R. Morris. Benchmark Synthesis using the LRU Cache Hit Function. *IEEE Transactions on Computers*. vol. 37, no. 6, pp. 637-645, June 1988.
- [7] R. Weicker. An Overview of Common Benchmarks. *IEEE Computer*, pp. 65-75, December 1995.
- [8] D. Thiebaut. On the Fractal Dimension of Computer Programs and its Application to Prediction of the Cache Miss Ratio. *IEEE Transactions on Computers*, vol. 38 no. 7, pp. 1012-1026. July 1989.
- [9] K. Sreenivasan and A. Kleinman. On the Construction of a Representative Synthetic Workload. *Communications of the ACM*, pp. 127-133, March 1974.
- [10] E. Sorenson and J. Flanagan. Evaluating Synthetic Trace Models Using Locality Surfaces. In *Proceedings of the IEEE International Workshop on Workload Characterization*, pp. 23-33. November 2002.
- [11] L. Eeckhout, S. Naussbaum, J.E. Smith, and K.De Bosschere. Statistical Simulation: Adding Efficiency to the Computer Designer's Toolbox. *IEEE Micro*, vol. 23 no.5, pp. 26-38, Sept/Oct 2003
- [12] L. Eeckhout, R. Bell Jr., B. Stougie, K. De Bosschere, and L. John. Improved Control Flow in Statistical Simulation for Accurate and Efficient Processor Design Studies. *International Symposium on Computer Architecture*, June 2004.
- [13] R. Bell, Jr., L. Eeckhout, L. John and Koen De Bosschere. Deconstructing and Improving Statistical Simulation in HLS, *Workshop on Duplicating, Deconstructing and Debunking*, in conjunction with ISCA '04, June 2004.

- [14] L. Eeckhout, K. De Bosschere, and H. Neefs. Performance Analysis through Synthetic Trace Generation. International Symposium on Performance Analysis of Systems and Software. April 2000.
- [15] 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, no. 11, pp. 1080-1089, November 1998.
- [16] H. Curnow and B. Wichman. A Synthetic Benchmark. *Computer Journal*, vol. 19, no. 1, pp. 43-49, February 1976
- [17] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, vol. 24, no.4, pp. 703-746, 1999.
- [18] T. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. *In SIGPLAN Conference on Programming Languages Design and Implementation*, pp. 191-202, 2001.
- [19] E. Berg and E. Hagersten. Fast Data-Locality Profiling of Native Execution. *In ACM SIGMETRICS*, June 2005.
- [20] V. Iyengar and L. Trevillyan, and P. Bose. Evaluation and generation of reduced traces for benchmarks. Technical Report RC20610, IBM Research Division, T.J. Watson Research Center, October 1996.
- [21] T. Conte and W-M.Hwu. Benchmark Characterization for Experimental System Evaluation. *Proceedings of the 1990 Hawaii International Conference on System Sciences (HICSS)*, vol I, Architecture Track, pp. 6-16, 1990.
- [22] A. Agarwal et al. An Analytical Cache Model. *ACM Transactions on Computers Systems*. 1989.
- [23] 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*, 2000.
- [24] <http://www.cse.ucsd.edu/~calder/simpoint/single-sim-points.htm>