Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks

Ajay Joshi[1], Lieven Eeckhout [2], Robert H. Bell Jr.[3], and Lizy John[1]

[1] - Department of Electrical and Computer Engineering
The University of Texas at Austin, Texas
{ajoshi,ljohn}@ece.utexas.edu

[2] - ELIS Department
Ghent University, Belgium
leeckhou@elis.ugent.be

[3] - IBM Systems and Technology Group
Austin, Texas
robbell@us.ibm.com

Abstract

Many embedded real world applications are intellectual property, and vendors hesitate to share these proprietary applications with computer architects and designers. This poses a serious problem for embedded microprocessor designers – how do they customize the design of their microprocessor to provide optimal performance for a class of target customer applications? In this paper, we explore a technique that can automatically extract key performance attributes of a real world application and clone them into a synthetic benchmark. The advantage of the synthetic benchmark clone is that it hides functional meaning of the code but exhibits similar performance characteristics as the target application. Unlike previously proposed workload synthesis techniques, we only model microarchitecture-independent performance attributes into the synthetic clone. By using a set of embedded benchmarks from the MediaBench and MiBench suites, we demonstrate that the performance and power consumption of the synthetic clone correlates well with that of the original application across a wide range of microarchitecture configurations.

1. Introduction

Embedded microprocessors, unlike the microprocessors that are targeted for general purpose, workstation, and server class of applications, are typically designed to provide optimal performance for a very narrow spectrum of applications. Also, the objective in the embedded space is to design a microprocessor which provides the necessary performance for a given class of applications at the lowest cost; in contrast to the objective of maximizing performance for a given cost, in the general purpose and workstation class of microprocessors. In order to achieve this objective, it is extremely important for architects and designers to understand the workload characteristics of the specific applications, used by targeted customers, for which the embedded microprocessor is being designed. If it would be possible to make a real world customer workload available to architects and designers, design tradeoffs could be made with higher confidence. Moreover, if a real world application that a customer cares about was used to project the performance of an embedded microprocessor, it would also tremendously increase the customer's confidence when making purchasing decisions. However, many of the critical real world embedded applications are proprietary and customers hesitate to share them with third party computer architects and designers.

The absence of such real world embedded applications makes it extremely challenging to reliably benchmark microprocessors used in embedded systems. Unfortunately, there have not been good benchmarks for embedded systems. Traditionally, most of the embedded microprocessor designers have used workstation benchmarks such as Standard Performance Evaluation Corporation (SPEC) CPU benchmarks [36] and synthetic benchmarks such as Dhrystone [26] to make design tradeoffs. Although these benchmarks measure specific aspects of a microprocessor performance, it is difficult to extrapolate the performance of the microprocessor for a specific embedded customer application. The arrival of the EDN Embedded Microprocessor Benchmarking Consortium (EEMBC) benchmarks [35], comprising of core algorithms used in a wide range of embedded applications has, to some extent, alleviated the problem of embedded microprocessor performance evaluation and benchmarking. However, embedded microprocessor designers believe that it is still extremely important to evaluate the performance of an embedded microprocessor design in the context of a target customer application when making design tradeoffs in the middle and end phase of the design process [37].

The objective of this paper is to explore an approach which will make it possible to share real world embedded applications with architects and designers without compromising on the proprietary nature of such applications. In order to achieve this we propose a technique, *performance cloning*, which extracts key performance characteristics of a real world application and then models them into a synthetic benchmark – effectively creating a synthetic program clone with similar performance characteristics as the original application, but with entirely different source code. The synthetic benchmark clone comprises of C-code with

embedded assembly language statements using the *asm* construct. The advantage of the synthetic benchmark clone is that it provides code abstraction capability *i.e.,* it hides the functional meaning of the code used in the original application but exhibits similar performance characteristics as the real application. This ability to clone performance makes it possible to disseminate a proprietary real world application as a synthetic benchmark that can be used by architects and designers, in lieu of the original application.

Our technique of performance cloning is similar to the concept of automatic workload synthesis proposed by Bell *et al.* [24] to reduce simulation time of long running benchmark programs. However, the primary objective of performance cloning is to disseminate real world customer applications as benchmarks, and not to reduce simulation time. Also, the performance cloning approach that we propose overcomes an important shortcoming of the existing workload synthesis technique – the memory access patterns and control flow behavior in existing workload synthesis techniques are modeled using microarchitecture-dependent attributes *i.e.,* the synthetic workload is generated to match a target metric such as cache miss rate and branch misprediction rate. As a result, the workloads generated using microarchitecture-dependent attributes yield large errors when the cache and branch configurations are changed [24]. Instead, in the performance cloning technique, we model the memory access pattern and branch behavior of a real world application into the synthetic benchmark clone using microarchitecture-independent workload characteristics. Consequently, as we show in our evaluation, the synthetic benchmark clone shows good correlation with the original application across a wide range of cache, branch predictor, and other microarchitecture configurations.

Specifically, we make the following contributions in this paper:

1) We propose to apply a workload synthesis technique to address the problem of making real world proprietary embedded applications available to architects and designers.
2) We develop a model using only microarchitecture-independent attributes to mimic the data access pattern of a real application in a synthetic benchmark clone.
3) We develop a model using only microarchitecture-independent attributes to replicate the control flow predictability of an application into a synthetic benchmark clone.
4) We apply the performance cloning technique to generate synthetic benchmark clones for twenty three programs from the MiBench and MediaBench embedded benchmark suite, and show that they provide good correlation with the original benchmark programs across a wide range of microarchitecture configurations.

The remainder of this paper is organized as follows: In section 2 we summarize prior research work in workload synthesis. In section 3 we provide an overview of the performance cloning approach and the new microarchitecture-independent approaches that we have developed to model memory access patterns and branch predictability into the synthetic benchmark clone. In section 4 we outline the experimental setup and the benchmarks used in this study. In section 5 we evaluate the performance cloning approach and present results from our experiments. In section 6 we discuss opportunities for improving the usefulness of the performance cloning technique. Finally, in section 7 we summarize the key results from this paper and the contributions of our work.

2. Prior Work

Statistical Simulation: Oskin *et al.* [21], Eeckhout *et al.* [18], and Nussbaum *et al.* [28] introduced the idea of statistical simulation which forms the foundation of synthetic workload generation. The approach used in statistical simulation is to generate a short synthetic trace from a statistical profile of workload attributes such as basic block size distribution, branch misprediction rate, data/instruction cache miss rate, instruction mix, dependency distances, *etc.,* and then simulate the synthetic trace using a statistical simulator. Eeckhout *et al.* [18] improved statistical simulation by profiling the workload attributes at a basic block granularity using statistical flow graphs. Further improvements include more accurate memory data flow modeling for statistical simulation [9]. The important benefit of statistical simulation is that the synthetic trace is extremely short in comparison to real workload traces – 1 million synthetically generated instructions are typically sufficient. Moreover, various studies have demonstrated that statistical simulation is capable of identifying a region of interest in the early stages of the microprocessor design cycle while considering both performance and power consumption. As such, the important application for statistical simulation is to cull a large design space in limited time in search for a region of interest.

Constructing Synthetic Workloads: Several approaches [7] [13] [17] have been proposed to construct a synthetic drive workload that is representative of a real workload under a multiprogramming system. In these techniques, the characteristics of the real workload are obtained from the system accounting data, and a synthetic set of jobs are constructed that places similar demands on the system resources. Hsieh *et al.* [3] developed a technique to construct assembly programs that, when executed, exhibit the same power consumption signature as the original application. Sorenson *et al.* [11] evaluate various approaches to generating synthetic traces using locality surfaces. Wong and Morris [33] 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.

The work most closely related to our approach is the one proposed by Bell and John [24]. They present a

framework for the automatic synthesis of miniature benchmarks from actual application executables. The key idea of this technique is to capture the essential structure of a program using statistical simulation theory, and generate C-code with assembly instructions that accurately model the workload attributes. Our performance cloning technique significantly improves the usefulness of this workload synthesis technique by developing microarchitecture-independent models to capture locality and control flow predictability of a program into synthetic workloads.

Workload Synthesis in Other Computer Systems: Approaches to generate synthetic workloads have been investigated for performance evaluation of I/O subsystems, file system, networks, and servers [12] [15] [25] [34]. The central idea in these approaches is to model the workload attributes using a probability distribution such as Zipf's law, binomial distribution *etc.,* and use these distributions to generate a synthetic workload.

3. Performance Cloning Framework

Figure 1 illustrates the performance cloning framework that we explore in this paper for generating synthetic benchmark clones from a real world application. The process comprises of two steps: 1) Profiling the real world proprietary workload to measure a collection of its inherent microarchitecture-independent workload attributes, and 2) Modeling the measured workload attributes into a synthetic program.

In the first step we characterize the application by measuring its key microarchitecture-independent workload characteristics that can impact performance. Note that these characteristics are related only to the functional operation of the program's instructions and are independent of the microarchitecture on which the program executes. Automatic workload synthesis techniques that have been previously proposed [24] have typically used a combination of microarchitecture-independent and microarchitecture-dependent workload attributes to characterize an application. Typically, these techniques model the memory access pattern and branch behavior in the synthetic workload using microarchitecture-dependent attributes such as cache miss rates and branch misprediction rates, *i.e.,* the synthetic workload is generated to match a target cache miss rate or a branch misprediction rate.

Consequently, the synthetic workloads generated from these models yield large errors when the cache and branch configurations are changed [24]. As a result, if one were to construct a synthetic benchmark clone using microarchitecture-dependent attributes, it would be necessary to construct separate clones for all branch predictor and cache configurations of interest. This severely limits the usefulness of the synthetic benchmark clone. In addition, this also implies that workload profiles need to be computed for every cache hierarchy and branch predictor of interest. An important contribution of this paper is that we develop memory access and branching models that use microarchitecture-independent workload attributes to capture the inherent locality and control flow predictability of a real world application into the synthetic benchmark clone.

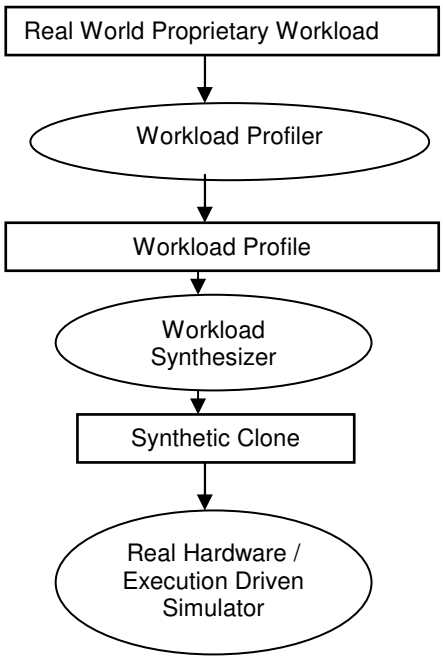

Figure 1. Performance Cloning framework for constructing synthetic benchmark clones.

After characterizing the real application, the second step is to construct a synthetic program with similar microarchitecture-independent attributes as the original application. Theoretically, if all the key microarchitecture-independent characteristics of the real application are successfully replicated into the synthetic benchmark clone, the synthetic benchmark should exhibit similar performance as the original application across a wide range of microarchitecture configurations. The characteristics that we model in this study are a subset of all the microarchitecture-independent characteristics that can be potentially modeled, but we believe that we model all the important inherent characteristics that impact a program's performance; the results from evaluation of the synthetic benchmark clones in this paper in fact show that this is the case, at least for the embedded application domain we target in this paper. The generated synthetic benchmark clone comprises of C-code with low-level assembly instructions instantiated as *asm* statements. The synthetic benchmark clone can be compiled and used in lieu of the original application for making design tradeoffs.

The details of the memory access model, branching model, and other microarchitecture-independent workload attributes that we use for profiling a real world applications and the procedure for modeling them into the synthetic benchmark clone are described in the following section.

3.1 Microarchitecture-Independent Workload Profiling

In this step we characterize the real application by measuring its inherent, or microarchitecture-independent, workload characteristics. In this paper we measure these characteristics using a functional simulator. However, instead of simulation, when using a customer production workload application, it is possible to efficiently measure these characteristics using a binary instrumentation tool such as ATOM [1] or PIN [4]. The microarchitecture-independent characteristics that we measure are fairly broad and cover a wide range of important program characteristics related to the instruction mix, control flow behavior, instruction and data locality, and instruction level parallelism (ILP).

3.1.1 Control Flow Behavior

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 benchmark clone it is essential to capture the program structure *i.e.,* a map of how the 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 [18]. A statistical flow graph is 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 basic block map with its size. Figure 2 shows an example statistical flow graph that is generated by profiling the execution of a program. The probabilities marked on the edges of each basic block indicate the transition probabilities, *e.g.,* the control flow transfer probability from Basic Block 1 to Basic Block 2 is 70%, if Basic Block 1 was executed. Note that this is analogous to a control flow graph of the program with the edges annotated with transition probabilities.

We measure the workload characteristics described below, instruction mix, data dependency distance distribution, and data locality characteristics for a unique pair of predecessor and successor basic blocks in the control flow graph *e.g.,* instead of measuring a single workload characteristics profile for Basic Block 4, we maintain separate workload characteristic profiles for the two instances where Basic Block 2 and Basic Block 3 are predecessors of Basic Block 4. Gathering the workload characteristics at this granularity improves the modeling accuracy because the performance of a basic block depends on the context in which it was executed.

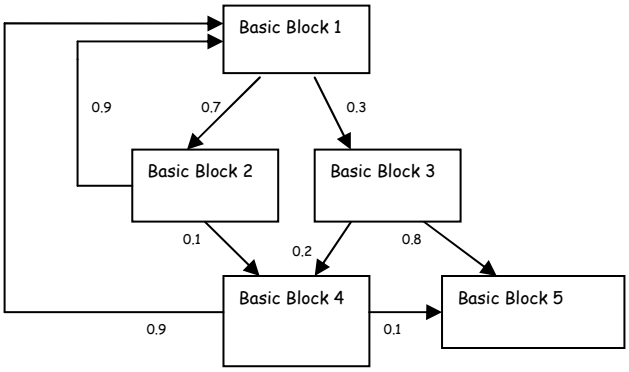

Figure 2. Example statistical flow graph used to capture the control flow structure of the program.

3.1.2 Instruction Mix

The instruction mix of a program measures the relative frequency of various operations performed in the program; namely the percentage of integer arithmetic, integer multiplication, integer division, floating-point arithmetic, floating-point multiplication, floating-point division operations, load, store, and branch instructions in the dynamic instruction stream of the program.

3.1.3 Data Dependency Distance Distribution

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 and memory location. The goal of measuring these data dependency distance distributions is very useful in capturing the inherent ILP of the program. We classify the dependency distance into six categories: percentage of total dependencies that have a distance of 1 instruction, and the percentage of total dependencies that have a distance of up to 2, 4, 6, 8, 16, 32, and greater than 32 instructions.

3.1.4 Data Locality

The principle of data locality is well known and recognized for its importance in determining an applications performance. 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 benchmark clone, the resulting program should show similar inherent temporal and spatial locality characteristics [29].

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 in typical embedded applications would appear more regular when viewed at a finer granularity of static memory access instructions (load/store), rather than at a coarser granularity of the global access stream. We profiled a set of embedded benchmark programs (described later) and measured the most frequently used stride value for every static load and store in the program. Then, based on the frequency of each static load or store instruction in the program, we computed the percentage of the dynamic references that will be accounted for if one were to approximate every static memory access instruction in the program with a single stride value. Figure 3 shows the percentage of dynamic memory references that exhibit a stride pattern with a single stride value. From this chart we observe that the embedded benchmark programs are fairly well behaved and modeling each static memory access instruction as one stream of access accounts for at least 90% of the dynamic memory references for each program. For most of the programs the value is greater than 95%. This suggests that in all these programs almost all load/store instructions originate from a stride pattern with a single stride value.

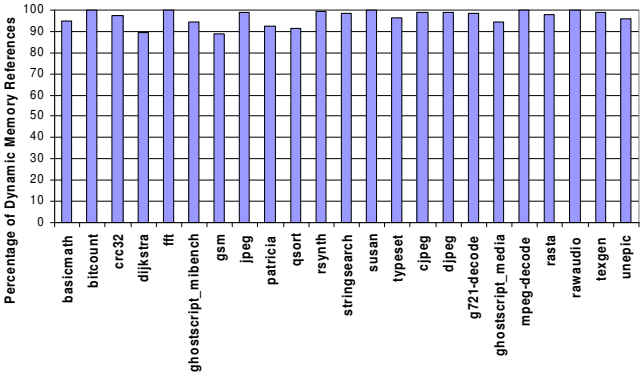

Figure 3. Percentage of dynamic memory references that exhibit a stride pattern with a single stride value.

Based on the results of these characterization experiments we propose a first-order model to generate access patterns in the synthetic benchmark clone. Our goal is to keep microarchitecture-independent workload model simple, so that it provides us with the flexibility to study "what-if" scenarios (by altering the memory access pattern of the program), which is almost impossible with a more complex model. When constructing the statistical flow graph, described in section 3.1.1, we record the most frequently used stride value for every static memory access instruction in every node in the statistical flow graph. Also, we find the average value of the length of each stream – calculated by averaging the length of each unique stream across all the unique stream pools in the program. When constructing a synthetic benchmark clone or a synthetic memory address trace we approximate each static load/store instruction in the program as accessing a fixed length stream with a stride value obtained from this workload characterization.

3.1.5 Control Flow Predictability

In order to incorporate synthetic branch predictability it is essential to understand the property of branches that makes 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 loop exit branches, 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 branch directions 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.* [20], for capturing the branch behavior in programs. Transition rate of a static branch is defined as the number of times it switches between taken and not-taken directions as it is executed, divided by the total number of times that it is executed. By definition, the branches with low transition rates are always biased towards either taken or not-taken. It has been well observed that such branches are easy to predict. Also, the branches with a very high transition rate always toggle between taken and not-taken directions and are also highly predictable. However, branches that transition between taken and not-taken sequences at a moderate rate are relatively more difficult to predict. In order to incorporate synthetic branch predictability we annotate every node in the statistical flow graph with its transition rate. When generating the synthetic benchmark clone we ensure that the distribution of the transition rates for static branches in the synthetic stream of instructions is similar to that of the original program. We achieve this by configuring each basic block in the synthetic stream of instructions to alternate between taken and not-taken directions, such that the branch exhibits the desired transition rate. Typically, when a branch instruction

does not have a very high or very low transition rate we use a divide operation (that performs modulo operation) followed by a conditional branch to control whether a synthetic branch will be taken or not-taken. The algorithm for generating the synthetic benchmark program in the next section describes the details of this mechanism.

3.2 Synthetic Benchmark Clone Generation

The next step is to generate a synthetic benchmark clone by modeling all the microarchitecture-independent workload characteristics, generated in the workload profiling phase, into a synthetic program. The basic structure of the algorithm used to generate the synthetic benchmark program is similar to the one proposed by Bell *et al.*[24]. However the memory and branching model is replaced with the microarchitecture-independent models described in section 3.1.

The following algorithm describes the details of how the synthetic benchmark clone is generated from the workload characteristics:

(1) Generate a random number in the interval [0, 1] and use this value to select a node in the statistical flow graph (using the cumulative distribution function based on the occurrence frequency of each node).
(2) Use the instruction mix statistics for each node in the statistical flow graph to populate the basic block with instructions; the last instruction should always be a conditional branch instruction.
(3) For each instruction, a dependency distance is assigned to satisfy the data dependency distance distribution for the node.
(4) For each static load and store instruction in the basic block assign a stream value – the most frequently used stride value for that load or store operation from the workload profile. Essentially, each static load and store instruction is modeled as a congruence class with a fixed stride value.
(5) A modulo operation using a logical left shift (divide) instruction is inserted in basic blocks where the transition rate is not very high or very low. The outcome of the modulo operation causes the conditional branch to be either taken or not taken depending on the number of the iteration (the entire sequence of basic blocks generated using this algorithm are executed in a loop). This mechanism is used to satisfy the transition rate of every basic block in the program, effectively capturing the control flow predictability into the synthetic benchmark clone.
(6) The occurrence of that node in the statistical flow graph is then decremented.
(7) Increment the count of the total number of basic blocks generated.
(8) A cumulative distribution function based on the probabilities of the outgoing edges of the nodes is then used to determine the next basic block to instantiate. If a node does not have any outgoing edges, go to step 1.
(9) If the target number of basic blocks has been generated, go to step 10, otherwise go to step 1.
(10) All the architected register usages in the synthetic benchmark are assigned to each instruction in the program, such that the data dependencies in step 3 are satisfied. The specifics of how the registers are selected and assigned are similar to the register assignment procedure outlined in [24].
(11) The generated sequence of instructions is made part of one big loop. Controlling the number of iterations of the loop effectively controls the number of dynamic instructions in the program. Each static load/store instruction in the program is configured to access a fixed length stream with a stride value obtained from this workload characterization. Also, after a certain number of iterations of the program (depending on the value of the stride length), each static load or store instruction resets to the first element sequence of strided access and re-walks the entire stream. The size of the data footprint can be controlled by varying the number of iterations after which the stride walk is to be reset.
(12) A code generator takes the set of representative instructions and generates a C-code with embedded assembly instruction using the *asm* construct. The instructions are targeted towards a specific ISA, alpha in our case. However, the code generator can be modified to emit instructions for a RISC ISA of interest. The code is encompassed in a *main* header and *malloc* library call is used to statically allocate memory for the data streams. The use of *volatile* directive for each *asm* statement prevents the compiler from optimizing out the machine instructions in the program.

The synthetic benchmark clone does not have a separate input data set. The characteristics of the input data set used by the real application are manifested in the workload characterization from which the synthetic benchmark clone has been generated. Therefore, one can think of the input set being assimilated into the synthetic benchmark clone. The synthetic benchmark clone generated from this step can be compiled and executed on an execution driven simulator or real hardware.

4. Experimental Setup

We used embedded benchmark programs from the MiBench and MediaBench benchmark suite to evaluate the proposed performance cloning methodology. All benchmark programs were compiled on an Alpha machine using the native Compaq cc v6.3-025 compiler with –O3 compiler optimization. Table 1 shows the benchmarks and the embedded application domains that they represent. For the benchmarks from the MiBench suite, we used the small input sets.

We used a modified version of the SimpleScalar functional simulator sim-safe to measure the workload

characteristics of the programs. However, as mentioned earlier, using a binary instrumentation tool would be a more efficient method to perform microarchitecture-independent workload characterization of a real world application program. In order to evaluate and compare the performance characteristics of the real benchmark and its synthetic clone, we used simulators from the SimpleScalar Toolset. In order to measure the power characteristics of the benchmarks we used the Wattch simulator [5].

We used the technique described in the previous section to construct a workload profile for each benchmark and then use it to generate a synthetic benchmark clone. The advantage of our performance cloning technique is that the synthetic clone is generated from microarchitecture-independent program characteristics, and can be used across a wide range of microarchitecture configurations. In order to evaluate our technique we compared the cache, branch predictor, and overall performance in terms of Instructions-Per-Cycle (IPC) of the real benchmark program with its synthetic clone.

Table 1. Embedded benchmark programs used for the evaluation.

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

5. Evaluation

We now evaluate whether the synthetic benchmark clones generated using the proposed approach indeed correlate well with the application from which they were generated. We perform our evaluation by changing the cache configurations and various aspects of the pipeline microarchitecture, and by comparing how well the synthetic benchmark clone correlates with the difference in performance exhibited by the real benchmark.

5.1. Tracking Changes Across Cache Configurations

In order to evaluate the model for incorporating synthetic data locality we used 28 different L1 D-caches with sizes ranging from 256 Bytes to 16 KB with direct-mapped, 2-way set-associative, 4-way set-associative, and fully associative configurations. The Least Recently Used replacement policy was used for all the cache configurations, and the cache line size was set to 32 bytes. We simulated the real benchmark program and the synthetic clone across these 28 different cache configurations and measured the number of misses-per-instruction. As described earlier, the primary objective of the synthetic benchmark clone is to be able to make design decisions and tradeoffs; where relative accuracy is of primary importance. We quantify the relative accuracy for the synthetic benchmark clones using the Pearson's linear correlation coefficient between the misses-per-instruction metric for the 27 different cache configurations relative to the 256 Byte direct-mapped cache configuration - for the original benchmark and the synthetic benchmark clone. Specifically, the Pearson's correlation coefficient is given by: $R_P = S_{XY} / (S_X . S_Y)$, where X and Y respectively refer to the misses-per-instruction of the synthetic benchmark clone and the original benchmark relative to the 256 Byte direct-mapped cache configuration. The value of correlation, R, can range from -1 to 1. The Pearson's correlation coefficient reflects how well the synthetic benchmark clone tracks the changes in cache configurations – a high positive correlation indicates that the synthetic benchmark clone tracks the actual change in misses-per-instruction, *i.e.* perfect relative accuracy.

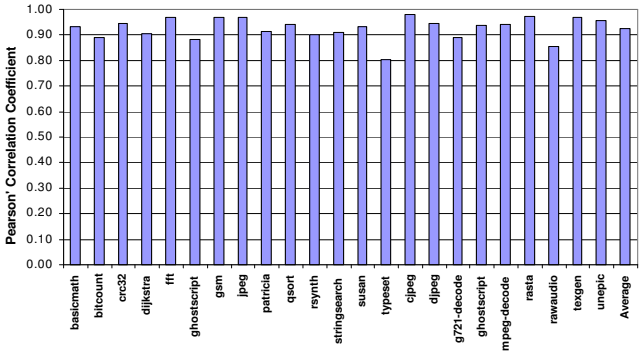

Figure 4. Pearson Correlation coefficient showing the efficacy of the synthetic benchmark clones in tracking the design changes across 28 different cache configurations.

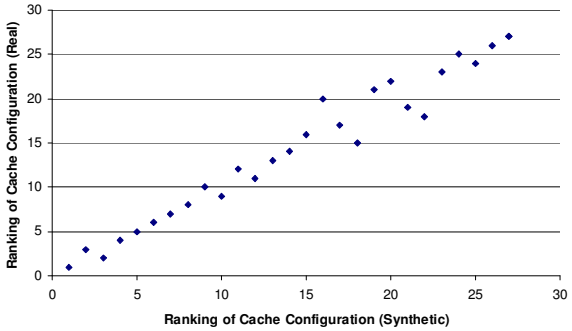

Figure 5. Scatter plot showing ranking of the cache configuration estimated by the synthetic benchmark clone and the real benchmark.

Figure 4 shows the Pearson's correlation coefficient for each benchmark program. The average correlation coefficient is 0.93, indicating very high correlation between the synthetic benchmark clone and the original benchmark application across all the applications. The benchmark typeset shows the smallest correlation (0.80) of all the benchmark suites. A plausible explanation for this observation is that the `typeset` benchmark needed 66

different unique streams to model its stride behavior, in compared to an average of 18 unique streams for the other benchmark program. This suggests that programs that require a larger number of unique stream values to capture the inherent data locality characteristics of a programs, introduce larger errors in the synthetic clone. This is perhaps due to the fact that having a large number of streams creates a larger number of possibilities of how the streams intermingle with each other, which is probably not accurately captured by our first-order synthetic benchmark generation method.

Figure 5 shows a scatter plot of the average rankings (cache with smallest misses-per-instruction is ranked the highest) of the 28 cache configurations predicted by the synthetic benchmark clones and the ones obtained using the real benchmark programs. Each point in the scatter plot represents a cache configuration. If the synthetic benchmarks accurately predicted all the rankings of the 28 cache configurations, all the points in the scatter plot will be along a line that passes through the origin and makes an angle of 45 degrees with the axes. From the chart it is evident that rankings predicted by the synthetic benchmark clone and those of the real benchmark are high correlated (all points are close to the 45 degree line passing through origin).

As such, based on the results in Figures 4 and 5, we can conclude that the synthetic benchmark clone is capable of tracking changes in cache sizes and associativities, and can be effectively used as a proxy for the real application in order to perform cache design studies.

5.2. Performance and Power Correlation Across Microarchitecture Changes

First, we compare the performance and power characteristics of the real benchmark and the synthetic clone on a base configuration. Table 2 shows the base microarchitecture configuration that we used for this experiment. We simulated the original benchmark and the synthetic benchmark clone on this configuration and measured the performance in terms of the Instructions-Per-Cycle (IPC) and the total power consumed. Figures 6 and 7 respectively show the absolute IPC and power consumption of the original benchmark program and the synthetic benchmark. The average absolute error for the synthetic benchmark clone across all the benchmark configurations is 8.73% for IPC and 6.44% for power consumption.

Table 2. Base Configuration used to evaluate the performance and power characteristics exhibited by the synthetic benchmark clone.

L1 I-cache	16 KB/2-way/32 B
L1 D-cache	16 KB/2-way/32 B
L2 Unified cache	64 KB/4-way/64 B
Fetch, Decode, and Issue Width	1-wide out-of-order
Fetch Queue	8 entry
Branch Predictor	2-level GAp predictor
Functional Units	2 Integer ALU, 1 FP Multiplication Unit, 1 FP ALU
Reorder Buffer	16 entries
Load Store Queue	8 entries
Memory (Bus Width, First Block Latency)	8 B, 40 cycles

When comparing the benchmark clone and the original benchmark program on the base configuration we only considered the absolute performance/power prediction accuracy so far, *i.e.,* the error in one design point. However, as mentioned before, for computer architects and designers the relative accuracy or the ability to predict a performance trend is often of primary importance. To evaluate the synthetic benchmark clone in this perspective we study how the synthetic benchmark clone tracks performance and power trends by successively altering various architectural parameters with respect to the base configuration. Specifically, we performed the following 5 experiments: (1) Doubled the number of entries in the reorder buffer and load store queue *i.e.,* from 16 and 8 entries to 32 and 16 entries respectively, (2) Reduced the L1-D cache size to half *i.e.,* from 16 KB to 8 KB, (3) Doubled the fetch, decode, and issue width, (4) Changed the branch predictor from a 2-level GAp predictor to an always not-taken branch predictor, and (5) Changed the instruction issue policy from out-of-order to in-order. For each of these configurations we simulated the original and the synthetic benchmark clone and the original benchmark.

For each experiment we measure the relative accuracy of the synthetic benchmark clone as: $RE_X = | M_{X,S}/M_{Y,S} - M_{Y,R}/M_{X,R}| / (M_{Y,R}/M_{X,R})$, where RE_X is the relative error when moving from design point Y (base configuration in our case) to design point X (each of the 5 design points), M is the target metric of interest (IPC or power consumption in our case), and R refers to the real benchmark, and S refers to the synthetic clone.

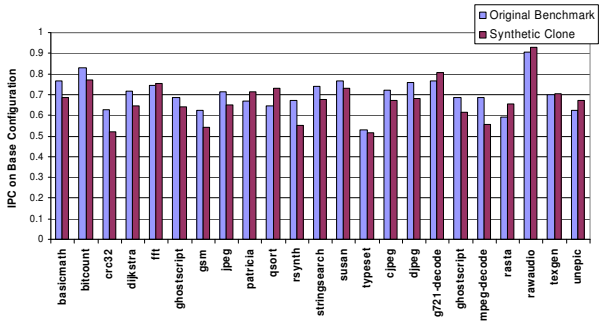

Figure 6. Comparison of the IPC of the original benchmark and the synthetic benchmark clone on the base configuration.

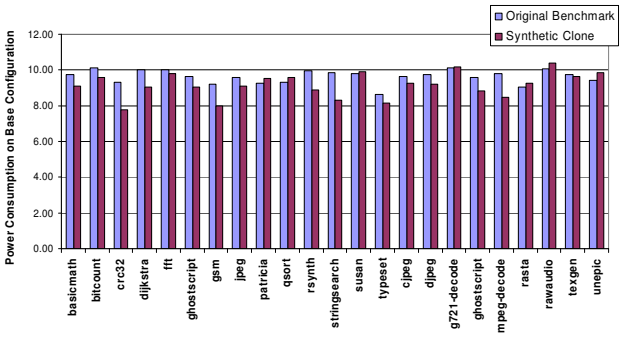

Figure 7. Comparison of the power consumed by the original benchmark and the synthetic benchmark clone on the base configuration.

Table 3 shows the relative error in IPC and power consumption of the synthetic clone for the 5 design changes described. The relative accuracy has been averaged across all the benchmark programs. We observe that in general, the relative errors across the 5 design changes are on an average 4.49 % (worst case 6.51%) for IPC and 2.28% for power (worst case 4.59%). The design change of doubling the fetch, decode, and issue width, resulted in the largest averaged speedup (1.72) across all the real benchmarks. Therefore, as an example, we illustrate the change in IPC and power consumption exhibited by the benchmarks for this design change. Figures 8 and 9 respectively show the speedup in IPC and the relative increase in power consumption for each real benchmark and its corresponding synthetic clone. It is encouraging that the relative errors are typically smaller than the absolute errors. The small errors in relative error in IPC and power consumption suggest that the synthetic benchmark clone can be effectively used to make design decisions, in lieu of the original application program.

Table 3. Average Relative Error in IPC and Power for the synthetic benchmark clone in response to 5 design changes.

Design Change	Average Relative Error in IPC	Average Relative Error in Power
Double the number of entries in the reorder buffer and load store queue	5.81%	3.41%
Reduce the L1 cache size to half	1.48%	0.39%
Double the fetch, decode, and issue width	5.41%	4.59%
Change the predictor from a 2-level to a not-taken predictor	6.51%	1.80%
Change the instruction issue policy to in-order	3.26%	1.22%

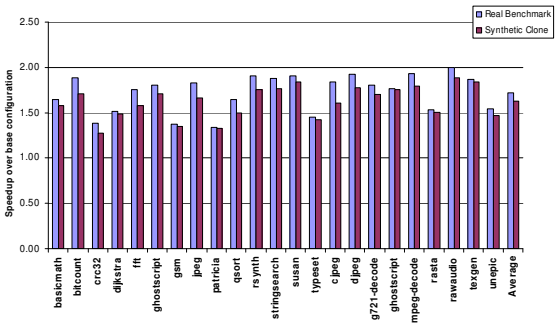

Figure 8. Relative speedup in IPC for real and synthetic benchmarks in response to the design change of doubling the fetch, decode, and issue width.

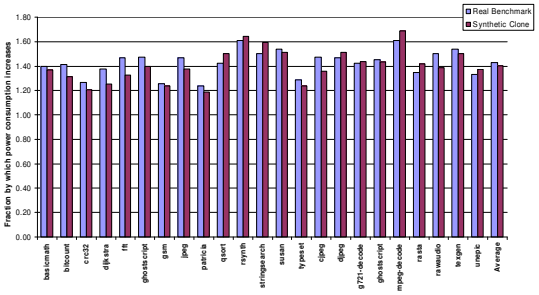

Figure 9. Relative increase in power consumption for real and synthetic benchmarks in response to the design change of doubling the fetch, decode, and issue width.

6. Discussion

As mentioned before, the advantage of the performance cloning approach proposed in this paper as compared to previously proposed workload synthesis techniques is that all the workload characteristics modeled into the synthetic benchmark clone are microarchitecture-independent. This makes the benchmarks portable across a wide range of microarchitecture configurations. However, a limitation of the proposed technique is that the synthetic benchmark clone is dependent on the compiler technology that was used to compile the real workload binary. Therefore, the generated synthetic benchmark clone may have limited application to the compiler community for studying the effects of various compiler optimizations on a benchmark.

A second note that we would like to make is that the synthetic benchmark clones that we generate contain instruction set architecture (ISA) specific assembly instructions embedded in C-code. Therefore, a separate benchmark clone would have to be synthesized for all target embedded architectures (*e.g.*, ARM, PowerPC, *etc.*) of interest. Typically, every embedded microprocessor designer would be interested only in his particular architecture and therefore this may not be a severe problem in practice. However, if the synthetic benchmark clone is to be made truly portable across ISAs, it would be important to address this concern. One possibility could be to generate the synthetic benchmark clone using a virtual instruction set architecture

that can then be consumed by compilers for different ISAs. Another possibility would be to binary translate the synthetic benchmark clone binary to the ISA of interest. Investigating this issue is a part of our ongoing research work.

A final note is that the abstract workload model presented in this paper is fairly simple by construction, *i.e.*, the characteristics that serve as input to the synthetic benchmark generation, such as the branching model and the data locality model, are far from being complicated. This was our intention: we wanted to build a model that is simple, yet accurate enough for predicting performance trends for embedded workloads on embedded processors. However, we anticipate that applying this approach to general-purpose workloads with more complex control flow behavior and data locality behavior could result in less accurate performance predictions. Just to name one example, the data behavior associated with code that applies pointer chasing through a linked list cannot be modeled using a stride model as we do in this paper. As such, as part of our future work, we plan to further extend this framework in order to be able to accurately model more complex workloads.

7. Conclusions

In this paper we explored a workload synthesis technique that can be used to clone a real-world proprietary application into a synthetic benchmark clone that can be made available to architects and designers. The synthetic benchmark clone has similar performance/power characteristics as the original application but generates a very different stream of dynamically executed instructions. By consequence, the synthetic clone does not compromise on the proprietary nature of the application. In order to develop a synthetic clone using pure microarchitecture-independent workload characteristics, we develop memory access and branching models to capture the inherent data locality and control flow predictability of the program into the synthetic benchmark clone. We developed synthetic benchmark clones for a set of benchmarks from the MiBench and MediaBench benchmark suites, and showed that the synthetic benchmark clones exhibit good accuracy in tracking design changes across 28 different cache configurations and 5 microarchitecture design changes.

The technique proposed in this paper will benefit embedded architects and designers to gain access to real world applications, in the form of synthetic benchmark clones, when making design decisions. Moreover, the synthetic benchmark clones will help the vendors to make informed purchase decisions, because they would have the ability to benchmark an embedded microprocessor using an application of their interest.

8. Acknowledgements

This research is supported in part by NSF grant 0429806, the IBM Systems and Technology Division, IBM CAS Program, and AMD. Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research – Flanders (Belgium) (F.W.O Vlaanderen) and is also supported by Ghent University, IWT, the HiPEAC Network of Excellence, and the European SARC project No. 27648.

9. References

[1] A. Srivastava and A. Eustace, “ATOM: A system for building customized program analysis tools”, Technical Report 94/2, Western Research Lab, Compaq, March 1994.

[2] AJ KleinOsowski and David J. Lilja, “MinneSPEC: A New SPEC Benchmark Workload Simulation-Based Computer Architecture Research,” *Computer Architecture Letters,* vol.1, June 2002.

[3] C. Hsieh and M. Pedram, "Microprocessor power estimation using profile-driven program synthesis," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 17(11), pp. 1080-1089, November 1998.

[4] C. Luk *et al.,* “PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation,”, *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[5] D. Brooks, V. Tiwari, and M. Martonosi, “Wattch: A framework for architectural-level power analysis and optimizations,” *Proceedings of International Symposium on Computer Architecture,* 2000, pp. 83-94

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

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

[8] D. Thiebaut, “On the Fractal Dimension of Computer Programs and its Application to the Prediction of the Cache Miss Ratio,” *IEEE Transaction on Computers*, vol. 38(7), pp. 1012-1026 July 1989.

[9] D. Genbrugge, L. Eeckhout and K. De Bosschere, “Accurate Memory Data Flow Modeling in Statistical Simulation”, in *Proceedings of the 2006 International Conference on Supercomputing*, pp. 87-96, June 2006.

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

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

[12] G. Ganger, “Generating Representative Synthetic Workloads: An Unsolved Problem,” in *Proceedings of Computer Management Group Conference*”, 1995, pp. 1263-1269.

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

[14] J. Henning, “SPEC CPU2000: Measuring CPU Performance in the New Millennium”, *IEEE Computer*, vol. 33(7), pp. 28-35, July 2000.

[15] K. Keaton and D. Patterson, “Towards a Simplified Database Workload for Computer Architecture Evaluations,” in *Proceedings of IEEE Workshop on Workload Characterization*, 1999, pp.115-124.

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

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

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

[19] L. Eeckhout, S. Nussbaum, J.E. Smith, and K.De Bosschere, “Statistical Simulation: Adding Efficiency to the Computer Designer’s Toolbox,” *IEEE Micro*, vol. 23(5), pp. 26-38, Sept/Oct 2003.

[20] 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, pp. 241-250.

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

[22] P. Barford and M. Crovella, “Generating Representative Web Workloads for Network and Server Performance Evaluation,” in *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* 1998, pp. 151-160.

[23] R. Bell Jr. and L. John, “Efficient Power Analysis using Synthetic Testcases,” in *Proceedings of International Symposium on Workload Characterization,* 2005, pp. 110-118.

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

[25] R. Bodnarchuk and R. Bunt, “A synthetic workload model for a distributed system file server”, in *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* 1991, pp. 50-59.

[26] R. Weiker, “Dhrystone: A Synthetic Systems Programming Benchmark,” *Communications of the ACM*, pp. 1013-1030, Oct 1984.

[27] R. Wunderlich, T. Wenish, B. Falsafi, and J. Hoe, “SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling,” in *Proceedings of International Symposium on Computer Architecture*, June 2003, pp. 84-95.

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

[29] S. Sair, T. Sherwood, and B. Calder, “Quantifying Load Stream Behavior,”, *Proceedings of International Symposium on High Performance Computer Architecture*, 2002.

[30] 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.

[31] 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.

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

[33] W. Wong and R. Morris, "Benchmark Synthesis Using the LRU Cache Hit Function," IEEE Transactions on Computers, vol. 37(6), pp. 637-645, June 1998.

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

[35] The Embedded Microprocessor Benchmark Consortium http://www.eembc.org/

[36] Standard Performance Evaluation Corporation www.spec.org/benchmarks.html

[37] “Challenges in Capturing Real World Workloads into Benchmarks”, *Panel Discussion at Workshop on Workload Characterization 2004.* http://www.iiswc.org/wwc7_slides/david.pdf