Benchmark Synthesis for Architecture and Compiler Exploration

Luk Van Ertvelde Lieven Eeckhout
Ghent University, Belgium

Abstract— **This paper presents a novel benchmark synthesis framework with three key features. First, it generates synthetic benchmarks in a high-level programming language (C in our case), in contrast to prior work in benchmark synthesis which generates synthetic benchmarks in assembly. Second, the synthetic benchmarks hide proprietary information from the original workloads they are built after. Hence, companies may want to distribute synthetic benchmark clones to third parties as proxies for their proprietary codes; third parties can then optimize the target system without having access to the original codes. Third, the synthetic benchmarks are shorter running than the original workloads they are modeled after, yet they are representative. In summary, the proposed framework generates small (thus quick to simulate) and representative benchmarks that can serve as proxies for other workloads without revealing proprietary information; and because the benchmarks are generated in a high-level programming language, they can be used to explore both the architecture and compiler spaces.**

The results obtained with our initial framework are promising. We demonstrate that we can generate synthetic proxy benchmarks for the MiBench benchmarks, and we show that they are representative across a range of machines with different instruction-set architectures, microarchitectures, and compilers and optimization levels, while being 30 times shorter running on average. We also verify using software plagiarism detection tools that the synthetic benchmark clones hide proprietary information from the original workloads.

I. Introduction

Benchmarking is at the foundation of architecture and compiler research and development. Computer architects and compiler designers make extensive use of benchmarks to evaluate their products and research ideas. Current benchmarking practice is to employ benchmarks that are derived from real-life applications. Although this is an effective approach, there are two major limitations. First, contemporary benchmarks have very large dynamic instruction counts, and as a result simulation is very time-consuming — it can easily take days or weeks to run a benchmark simulation to completion. Second, the available benchmarks may not be truly representative for real-life applications. In many cases, the real-life applications are proprietary, and companies are not willing to share their codes. Hence, third parties need to resort to (most often open-source) benchmarks that may not be truly representative for the real-life applications.

In this paper, we present a novel benchmark synthesis approach that aims at addressing these two limitations: the synthetic benchmarks are shorter than the original workloads they are modeled after (and hence they simulate faster), and they do not expose proprietary information (and can thus be distributed to third parties), yet they are representative with respect to real-life applications. The key novelty of the approach is that the synthetic benchmarks are generated in a high-level programming language, C in our case.

Our preliminary implementation and experimental evaluation using the MiBench benchmarks [1], multiple hardware platforms with very different instruction-set architectures (ISAs) and microarchitectures, and different compilers and optimization levels demonstrate the potential of the approach: the synthetic benchmarks mimick the real workloads well across architectures and compiler optimizations. We also provide evidence that a synthetic benchmark's source code does not provide any similarity with the original workload, hence it does not reveal proprietary information.

This work shares some commonalities with the recent line of research in statistical simulation [2]. The basic idea in statistical simulation is to collect a set of program characteristics by profiling a workload; these program characteristics are typically measured in the form of distributions. This statistical profile then serves as input for a synthetic workload generator. The synthetic workload then exhibits similar execution behavior as the original workload, so that it can be used as a proxy. Early proposals in statistical simulation generated synthetic traces [3], [4], [5]. While this is a reasonable approach for trace-driven simulation, it cannot be used on execution-driven simulators nor on real hardware. For that reason, researchers have proposed frameworks that generate synthetic benchmarks instead of synthetic traces [6], [7]. The synthetic benchmarks are generated at the binary level, hence they are tied to a particular ISA. Our work takes a significant step forward by generating synthetic benchmarks in a high-level programming language, which enables both architecture and compiler research. In addition, our framework models a program's control flow behavior more accurately than prior work and it generates synthetic code sequences using pattern recognition, not through statistics nor distributions of program characteristics.

More in particular, this paper makes the following contributions.

- We propose a framework and methodology for generating synthetic benchmarks in a high-level programming language that are representative for other workloads. Prior work generates synthetic benchmarks at the binary level which excludes using them across instruction-set architectures and compilers.
- We propose a novel structure, called the SFGL (Sta-

tistical Flow Graph with Loop information), to capture a program's control flow behavior in a statistical way. In particular, the SFGL captures a program's loop and basic block execution patterns. The SFGL enables the framework to generate function calls, (nested) loops and conditional control flow behavior in the synthetic benchmark. Prior work instead generated a linear sequence of basic blocks, but no loops nor function calls.

- We evaluate our framework on x86 and IA64 hardware. Prior work in synthetic benchmark generation was limited to evaluation through simulation, and/or considered RISC ISAs (Alpha, PowerPC) — RISC ISAs are easier to handle than CISC ISAs such as x86. We consider multiple hardware platforms with different ISAs, microarchitectures, compilers and optimization levels, and our preliminary results demonstrate good correspondence between the synthetic benchmarks and the original workloads in terms of performance sensitivity to the architecture, microarchitecture and compiler optimization level. In addition, the synthetic benchmarks are shown to be shorter running than the original workloads, and substantial simulation speedups are obtained.
- We demonstrate that the synthetic benchmarks do not reveal proprietary information. Two existing tools for identifying software plagiarism, Moss and JPlag, confirm that the synthetic benchmark does not show any similarity with the original workload.

II. Benchmark synthesis framework

The key problem to be solved in this paper is the following. We want to generate a synthetic benchmark in a high-level programming language that is similar to a real workload in terms of its execution behavior across architectures and compilers, yet it should not expose proprietary information and it should be short-running compared to the real workload. This is a non-trivial problem to solve. We now describe how we approach this problem at a high level — we will delve into the details later.

A. Framework overview

Figure 1 provides a high-level view of the overall framework. We start off from a real workload. This could be a proprietary application with a proprietary input. This workload is then compiled at a low optimization level, e.g., `-O0` in GNU's GCC. We then run the resulting binary and profile its execution, i.e., we count how often each function is called, how many times a loop is iterated, how often a branch is taken, how often a basic block is executed, etc. — this information is stored in the SFGL structure. In addition, we record memory access patterns as well as branch taken and transition rates. Finally, we employ a (simple) pattern recognizer that scans the executed code to identify C code statements that correspond to sequences of instructions observed at the binary level. This pattern recognizer translates the binary code to C code in a semi-random fashion in order to obfuscate proprietary information. All the characteristics that we collect are comprised

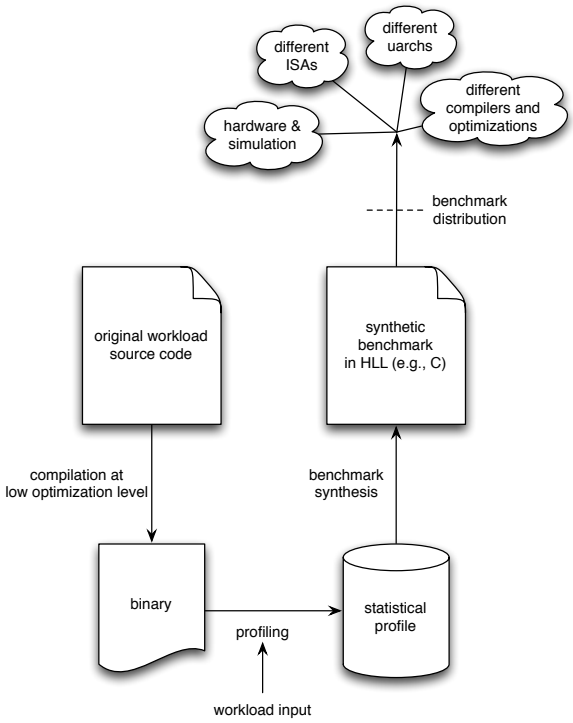

Fig. 1. Benchmark synthesis framework overview.

in a so called statistical profile that captures the behavior of the original workload and its input. We then generate a synthetic benchmark from this statistical profile. This is done in a high-level programming language (HLL), in our case C: we generate sequences of C code statements (basic blocks), as well as if-then-else statements, loops and function calls, and we add inter-statement dependencies as well as data memory access patterns. The C code structures are generated pro rata their occurrences in the original workload. However, we force the synthetic benchmark to execute fewer instructions than the original workload, by construction. This is done by reducing the execution frequencies of basic blocks, loops and function calls by a given reduction factor R. The end result is a synthetic benchmark that executes fewer instructions than the original workload while being representative for the original workload.

The synthetic benchmark does not expose proprietary information (because of the semi-random binary to source code translator, and the workload reduction) and can thus be distributed to third parties. Because the synthetic benchmarks are generated in a high-level programming language, they enable exploring both the architecture and compiler spaces, and compare systems with different compilers, and optimization levels, as well as different instruction-set architectures, microarchitectures and implementations. The synthetic benchmarks can run on execution-driven simulators as well as on real hardware.

An important aspect of our approach is that we compile the original workload at a low compiler optimization level before profiling. The reason for doing so is to force the compiler not

to perform aggressive optimizations. This facilitates the pattern recognition and translation from binary code to C code, and, more importantly, it enables generating synthetic benchmarks that can later be used to explore the compiler space, as we will demonstrate in this paper.

B. Applications

We believe this framework has a number of potential applications.

a) Distributing synthetic benchmarks as proxies for proprietary workloads: The most obvious application is to use the framework to generate synthetic clones for real-life proprietary workloads. There are many possible application scenarios, both in the embedded and server/datacenter spaces. For example, phone companies may not be willing to share their proprietary software with a processor vendor in order to optimize the processor architecture for the next-generation cell phone, yet they may be willing to share a synthetic clone. A similar application scenario applies to service providers in the cloud: they will be reluctant to share their platform software, yet they may want to distribute synthetic clones to third party hardware vendors. The same applies to compiler builders: they could evaluate their compiler performance based on the synthetic clones rather than the real workloads. Of course, co-optimization of hardware and software, which is an important focus today given the emphasis on energy-efficient computing, can also rely on synthetic benchmark clones.

The framework may also be an enabler for industry to share their workloads with their research partners in academia without revealing proprietary information. This will eventually lead to a more fruitful collaboration between industry and academia because the synthetic workloads may be more representative for the real-life commercial workloads than the open-source benchmarks used today.

b) Simulation time reduction: As mentioned earlier, the synthetic benchmarks are shorter running than the original workloads, i.e., their dynamic instruction count is significantly smaller. Because simulation time is an important concern in architecture research and development, benchmark synthesis also helps in reducing simulation time, and eventually the overall time-to-market. This is also important in the compiler space: for example, iterative compilation evaluates a very large number of compiler optimizations in order to find the optimum compiler optimizations for a given program [8], [9]. A synthetic clone that executes faster could reduce the overall compiler space exploration time.

c) Generate emerging workloads: The framework can also be used to generate emerging and future workloads. In particular, one can generate a statistical profile with performance characteristics that are to be expected for future emerging workloads. For example, one could generate specific sequences of C statements, a particular memory access behavior (e.g., large working set, random access patterns), etc. The synthetic benchmarks generated from these profiles can then be used to explore design alternatives for future computer systems.

d) Model hard-to-setup workloads: Similarly, one could build proxy benchmarks for workloads that are hard to setup. For example, database workloads and commercial workloads in general are non-trivial to setup [10]. Synthetics could be a way to facilitate the benchmarking process using commercial workloads. In fact, an additional advantage of generating synthetic benchmarks in a high-level programming language compared to assembly synthetic benchmarks is that interfacing libraries can be done easily using existing APIs. Although our current framework cannot be readily applied to mimicking commercial workloads, we believe it may be possible in future generations of our framework.

e) Benchmark consolidation: Multiple workloads can also be consolidated into a single synthetic benchmark. Basically, by putting together the statistical profiles from different workloads, one can generate a single consolidated synthetic benchmark that is representative for a set of workloads. Benchmark consolidation also helps hiding and obfuscating proprietary information.

III. Benchmark synthesis details

We now describe the details of our current framework. There are two major steps in the method: profiling a real workload and generating a synthetic benchmark clone.

A. Profiling

The profiling step can be implemented in a functional simulator or in a binary instrumenation tool, such as Pin [11] as we do in our framework. The profiler collects a number of characteristics.

1) Statistical Flow Graph with Loop annotation (SFGL): The central structure in the statistical profile is the SFGL which captures a program's control flow behavior in a statistical manner. Figure 2(a) shows an example. The nodes represent basic blocks and the edges represent control flow transitions. Each node is annotated with a basic block's execution frequency, and each edge is annotated with transition probabilities between nodes. The SFGL also identifies the loops along with the number of iterations that each loop executes.

For each instruction in each basic block we also record its instruction type. We consider a number of instruction types such as addition, subtraction, multiply, divide, etc., and we make a distinction between integer and floating-point instructions. We also keep track of the instruction's input operands (constant, register, memory) and output operand (register or memory).

2) Branch taken and transition rate: For each conditional branch that is not a loop back edge, we determine its taken and transition rate; the branch transition rate is defined as the number of times a branch changes between taken and not-taken during execution [12]. A low transition rate means that the branch is either mostly taken or mostly not taken, and a high transition rate means that the branch constantly changes between taken and not-taken. High and low transition rates typically suggest easy to predict branches. A medium

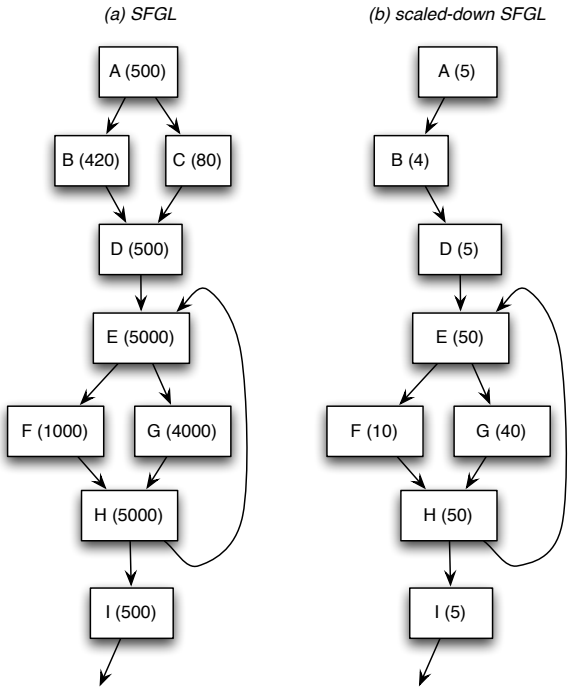

Fig. 2. An example SFGL: (a) computed SFGL and (b) scaled-down SFGL with a reduction factor $R = 100$.

class	*miss rate range*	*stride (bytes)*
0	0% - 6.25%	0
1	6.25% - 18.75%	4
2	18.75% - 31.25%	8
3	31.25% - 43.75%	12
4	43.75% - 56.25%	16
5	56.25% - 68.75%	20
6	68.75% - 81.25%	24
7	81.25% - 93.75%	28
8	93.75% - 100%	32

TABLE I

MEMORY ACCESS STRIDES FOR GENERATING A TARGET MISS RATE (ASSUMING A 32-BYTE CACHE LINE AND A 32-BIT ARCHITECTURE).

transition rate suggests hard to predict branches. The branch transition rate is independent of a particular branch predictor, and we classify branches into two classes, easy to predict branches (either high or low branch transition rate) and hard to predict branches.

3) Memory access patterns: For each memory access we record its cache hit/miss ratio. We do this by simulating a cache structure during profiling — there exist tools that compute cache miss rates across a range of cache organizations in a single pass [13]. We classify the memory accesses in a number of classes according to their hit/miss ratios. We identify 8 classes for different ranges of miss rates, see Table I; we will use these classes to generate specific memory access patterns, as we will describe later.

B. Synthetic benchmark generation

The second step in our framework is to generate a synthetic benchmark from the statistical profile. This is done in a number of steps.

1) Scale-down SFGL: We first compute a scaled-down SFGL by reducing the occurrences of the basic block and loop counts in the SFGL. This is done by dividing the basic block execution counts and loop iteration counts by a reduction factor R. For nested loops, we first scale the iteration count of the outer loop. If the iteration count of the outer loop is smaller than the reduction factor, we also downscale the nested loop, etc. Basic blocks and loops that are executed infrequently (i.e., less than R times) are removed from the SFGL. The purpose for downscaling is to generate short-running synthetic benchmarks, and obfuscate the original workload's semantics. Figure 2(b) shows the downscaled SFGL of the example shown in Figure 2(a). The reduction factor equals 100 in this example; basic block C does no longer appear in the down-scaled SFGL.

2) Generate basic blocks and loops: We now start generating the skeleton for the synthetic benchmark. We pick a random basic block based on the (reduced) execution counts, i.e., a basic block with a large execution count has a higher probability for being selected than a basic block with a lower execution count. If this basic block is part of a loop, we generate the loop that contains this basic block; for example, if basic block F would be picked in Figure 2(b), the framework would generate a loop comprising basic blocks E, F, G and H. If the loop itself is nested in a bigger loop, we first generate the outer loop and then generate the inner loops. If the basic block is not part of a loop, we determine its successor(s) and start building the control flow structure of the synthetic benchmark. If there are no successors to a basic block (because the successor basic blocks got removed during down-scaling), we re-start the generation algorithm and pick a random basic block. For each basic block and loop that we generate, we decrease the respective execution counts to reflect the fact that these basic blocks and loops have been generated. Basic blocks and loops with zero execution counts are removed from the SFGL. We continue this process until all basic blocks in the SFGL have been selected and the SFGL is empty.

3) Function assignment: We subsequently organize the basic blocks and loops to functions. This organization does not necessarily correspond to the functions observed in the original workload — again, this is to hide proprietary information in the synthetic benchmark.

4) Generate C statements: Once we have the skeleton synthetic benchmark consisting of functions, loops and basic blocks, we now populate the basic blocks with C statements. This is done by scanning the instruction types of all the instructions in each basic block, and by identifying C statements that correspond to these sequences of instructions. Table II shows the most important patterns and how they are translated into C statements. These patterns cover over 95% of the dynamic instructions for all the benchmarks. Coverage is not 100% (which again helps hiding proprietary information): to compensate for the uncovered instructions we keep track of the number of operations and types that have been translated so far, and we compensate for those instructions on a later occasion. For example, if we are lagging behind in the number

pattern	*example*	*C statement*
load-store	`movl t+512, %eax` `movl %eax, t+504`	`mem[i] = mem[j];`

load-arithmetic-store	`movl t+512, %eax`
	`addl $2,%eax`
	`movl %eax, t+504`
		`mem[i] = mem[j] op cst;`

load-load-arith-store	`movl t+508, %edx`
	`movl t+512, %eax`
	`leal (%edx,%eax), %eax`
	`movl %eax, t+504`
		`mem[i] = mem[j] op mem[k];`

load-load-arith-load-reg-arith-reg-store	`movl t+508, %edx`
	`movl t+512, %eax`
	`addl %eax, %edx`
	`movl t+516, %eax`
	`movl %edx, %ecx`
	`subl %eax, %ecx`
	`movl %ecx, %eax`
	`movl %eax, t+504`
		`mem[i] = mem[j] op mem[k] op mem[l];`

load-cmp-br	`movl t+504, %eax`
	`cmpl $3, %eax`
	`jbe`
		`if (mem[i] > cst)`

store	`movl $9, %eax`	`mem[i] = cst;`

TABLE II

GENERATING C STATEMENTS THROUGH PATTERN RECOGNITION. THE `op` REFERS TO AN OPERATION (E.G., ADDITION, SUBTRACTION, ETC.); THE `cst` REFERS TO A RANDOMLY GENERATED CONSTANT VALUE.

of loads, we try to generate a 'load-load-arith-store' pattern instead of a 'load-arith-store' pattern. Or, if we are lagging behind in the number of stores, we will generate an additional 'store' pattern.

When reaching the end of a basic block we generate a branch statement. This can be either a loop back edge or a conditional branch. In case of a loop back edge, we generate a `for` loop with the iteration count the number of times the loop needs to be iterated according to the scaled-down SFGL. For a non-loop branch, we generate an if-then-else statement, and we make a distinction between easy to predict branches and hard to predict branches. The easy to predict branches are assumed to be either always taken or always not-taken. The non-executed path is filled with C statements that print out the results that have been computed elsewhere in the synthetic benchmark — this is to force the compiler not to optimize code away that is needed to preserve representativeness while producing data that is never used. The hard to predict branches jump in one or the other direction based on their transition rate using a modulo operation on a loop iterator. For example, a conditional branch with a transition rate of 30% is modeled using an operation that computes modulo 3 on the iterator of its innermost outerloop.

Finally, we also generate memory access patterns. This is done by generating stride patterns for all memory accesses, following prior work by Joshi et al. [7] who found that over 90% of the memory references can be modeled as stride access patterns. These patterns walk through pre-allocated memory with a particular stride; the stride value is determined by the memory access' hit/miss ratio. Different hit/miss ratios lead to different stride values. For example, an always hit memory access is modeled through a zero stride. A 50% hit rate is modeled through a stride value of 4; this will lead to a 50% miss rate assuming a 32 byte cache line size and a 32-bit machine, see also Table I.

C. Example

Figure 3 shows an example for the fibonacci kernel: it shows the original code along with the automatically generated synthetic clone. The profiling was done with a particular input; this is reflected in the number of iterations that the loop is taking: the synthetic benchmark takes 20 iterations while this is an input parameter in the original program. That specific input never caused an overflow, hence the if-statement in the loop is never executed. This example illustrates that the fibonacci kernel is no longer recognizable in the synthetic clone because the data dependencies between the statements are different between the original program and its clone.

D. Limitations

The current framework has a number of limitations which we plan to address as part of our future work.

- Different program characteristics are modeled independently of each other — the framework currently takes a first-order approach and assumes that the characteristics are uncorrelated. For example, memory access behavior is modeled independently of control flow behavior and its interaction is not modeled. This is obviously not the case in real programs. Modeling second-order effects is likely to improve accuracy.
- The memory access behavior is based on cache miss rates and hence it is specific to a particular memory hierarchy. Although it is possible to measure cache miss rates for a range of caches in a single run, as mentioned before, a better solution would be to have a microarchitecture-independent way of modeling memory access behavior.

(a) original program

```
int fib (int n) {
  int a=0, b=1, i, sum=0;

  for (i=0; i<n; i++) {
    sum=a+b;
    if (sum<0) {printf("overflow"); break;}
    a=b;
    b=sum;
  }
  return sum;
}
```

(b) synthetic program clone

```
unsigned int mStream0[256];
int i,j;

int f () {
  for (i=0; i<20; i++) {
    mStream0[4]=mStream0[7] + mStream0[2];
    if (mStream0[0]==0x99) {
      for(j=0;j<256;j++)
         printf("%d;", mStream0[j]);
      }
    mStream0[6]=i;
    mStream0[7]=mStream0[6];
  }
}
```

Fig. 3. The original fibonacci kernel (a) and its synthetic clone (b).

Further, the current approach assumes that memory accesses can be modeled using stride patterns, which is a reasonable approach according to [7], as discussed above. Future work though may focus on modeling less regular memory access patterns.

- The ILP model is simplistic in our current setup, as we assume random dependencies between instructions. A more accurate approach would be based on profiling information so that the distribution of data dependencies in the synthetic matches the original workload. A downside of making the approach more accurate though is that it may reveal proprietary information — there is a trade-off in accuracy versus hiding proprietary information.
- The reduction factor is chosen empirically so that the synthetic benchmark executes approximately 10 million instructions, as we will describe later. A more accurate approach would base the reduction factor on how representative the synthetic workload is relative to the real workload.

IV. Experimental setup

We use the MiBench benchmark suite [1] in our experimental setup, which is representative for the embedded market. We use MiBench in this work because the embedded space is a target application domain for the framework proposed in this paper, cf. the use case mentioned before in which a cell phone company may be willing to distribute a synthetic benchmark to hardware vendors as a proxy for its proprietary software. Our initial results with these benchmarks are promising, as we will describe in the next section. We will consider more complicated workloads as part of our future work.

machine	*ISA*	*description*
Pentium 4, 3GHz	x86	Pentium 4 at 3GHz w/ 1MB L2
Core 2	x86_64	Core 2 at 2.2GHz w/ 2MB L2
Pentium 4, 2.8GHz	x86	Pentium 4 at 2.8GHz w/ 1MB L2
Itanium 2	IA64	Itanium 2 at 900MHz w/ 256KB L2
Core i7	x86_64	Core i7 at 2.67GHz w/ 8MB L2

TABLE III
MACHINES USED IN THIS STUDY.

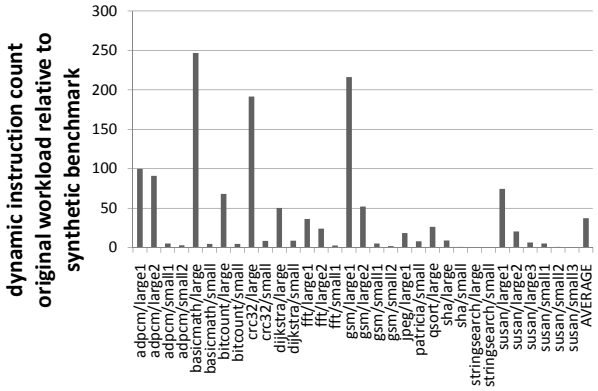

Fig. 4. Reduction in dynamic instruction count.

The profiling is done using Pin [11], which is a dynamic binary instrumentation tool. The cache simulations are done using Pin as well. The branch prediction results are obtained using PTLSim [14]; we consider a hybrid branch predictor with a bimodal component along with a history-based component. We also run detailed cycle-accurate simulations using PTLSim and we simulate a 2-wide out-of-order processor.

We also run real hardware experiments on five machines, see Table III. The machines include Pentium 4, Core 2, Core i7 and Itanium 2 processors, and three ISAs: x86, x86_64 and IA64.

We use GNU's GCC compiler v4.0.2 in all of our experiments for the x86 and x86_64 machines; we use GCC v3.3.2 on the Itanium 2 machine. We consider four compiler optimization levels: `-O0`, `-O1`, `-O2` and `-O3`.

V. Evaluation

The evaluation of the framework is done in a number of steps: we evaluate whether the synthetic benchmarks correspond to the real workloads with respect to their dynamic instruction count, instruction mix, cache performance, branch prediction behavior, and eventually overall performance across architectures and compilers.

A. Dynamic instruction counts

Figure 4 shows the reduction in dynamic instruction count between the synthetic benchmark and the original workload.

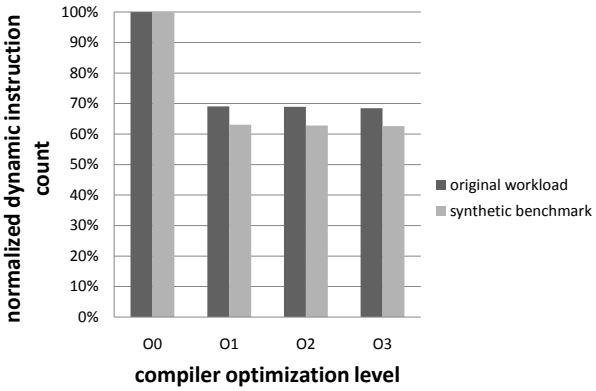

Fig. 5. Normalized dynamic instruction count across compiler optimization levels.

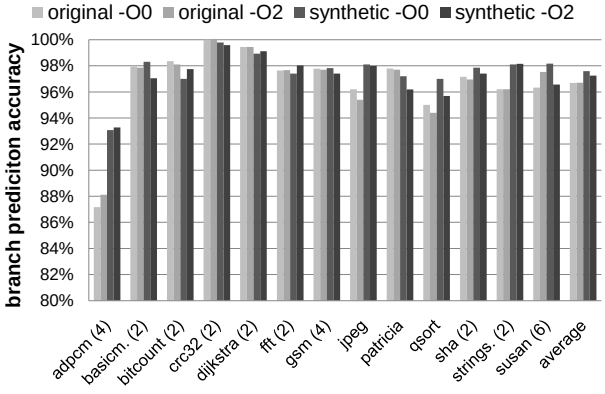

Fig. 9. Branch prediction rates for the original workloads and the synthetic benchmarks.

Recall that the synthetic benchmark generation process scales down the SFGL using a reduction factor. We choose the reduction factor such that the synthetic benchmark executes approximately 10 million instructions. This leads to a reduction factor ranging from 1 to 250. The fact that the reduction factor is low in a number of cases is due to the fact that some MiBench benchmarks are fairly short running, hence there is little simulation time reduction to be gained. On average though, we achieve a 30× reduction in dynamic instruction count.

Figure 5 shows the normalized dynamic instruction count across compiler optimization levels; we show average numbers here. The dynamic instruction count is an important optimization target for compilers, even on today's superscalar out-of-order processors [15]. The synthetic workload tracks the original workload fairly well: both suggest that the dynamic instruction count reduces by about a third when going from `-O0` to a higher optimization level.

B. Instruction mix, cache, and branch prediction behavior

Figure 6 shows the instruction mix at the `-O0` and `-O2` optimization levels. Figures 7 and 8 show similar graphs for the data cache behavior (we consider fairly small cache sizes in order to stress our framework); and Figure 9 shows results for the branch predictor behavior. All of these graphs basically lead to the same conclusion. Although the synthetic benchmarks do not yield a perfect match with the original workload, they most often yield the same conclusions and insights. For example, both the synthetics and the real workloads see a decrease in the fraction of load instructions along with an increase in the fraction of arithmetic instructions at a higher optimization level, see the average bars on the righthand side in Figure 6(a) and (b); the reason is that optimizations such as copy propagation eliminate load instructions. In terms of data cache behavior, `dijkstra` seems to be the benchmark that is most sensitive to cache space, see Figure 7(a), and a data cache size of 8KB seems to capture most of the benchmark's working set (i.e., there is a significant increase in data cache hit rate going from 4KB to 8KB but a minor increase going from 8KB to 16KB). We observe the same trend for the synthetic version of `dijkstra`, see Figure 7(b). Finally, for the branch predictor accuracy graph, see Figure 9, we observe that `adpcm` is most sensitive to the branch predictor; this is also captured by the synthetic workload.

C. Detailed cycle-accurate simulation

Figure 10 shows CPI for a 2-wide out-of-order processor while varying cache size; these results are obtained through detailed cycle-accurate simulation using PTLSim. The synthetics track fairly well overall performance across the benchmarks. For example, `fft` is the benchmark with the highest CPI (due to a large fraction of floating-point instructions) and `sha` the lowest CPI; we observe this for both the real and synthetic workloads. We also observe that the synthetic workload captures the performance trend as a function of data cache size well, see for example `dijkstra` and `qsort`. The remaining errors come from a number of potential sources. The current data dependency model can be improved to more accurately mimick real application behavior in the synthetic benchmarks (see `bitcount`); also, modeling the branch behavior can be improved upon (see `adpcm`), as well as the data cache behavior (see `stringsearch`). Improving the modeling of these program characteristics is likely to improve the representativeness of the synthetic benchmarks compared to the original workloads.

D. Overall performance across architectures and compilers

Figure 11 shows the normalized execution times for the original workloads and the synthetic clones across different architectures and compilers and optimization levels; we consider the real machines and a benchmark consolidation setup here and report average numbers. All the results are normalized to the `-O0` optimization level on the Pentium 4 3GHz machine. This graph shows that the synthetic workloads track the original workloads fairly well across architectures and compiler optimization levels. The error in predicting the speedup relative to `-O0` is less than 20% across all machines and optimization levels, with an average error of 7.4%. The synthetics track that the Core i7 yields the best overall performance, and the Itanium 2 the worst. A particularly encouraging

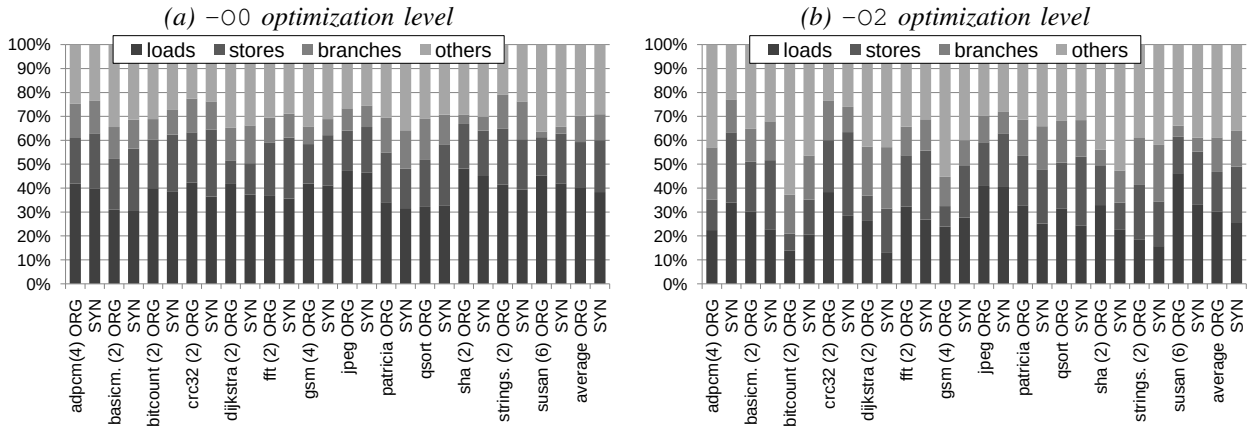

Fig. 6. Instruction mix for (a) the `-O0` optimization level and (b) the `-O2` optimization level.

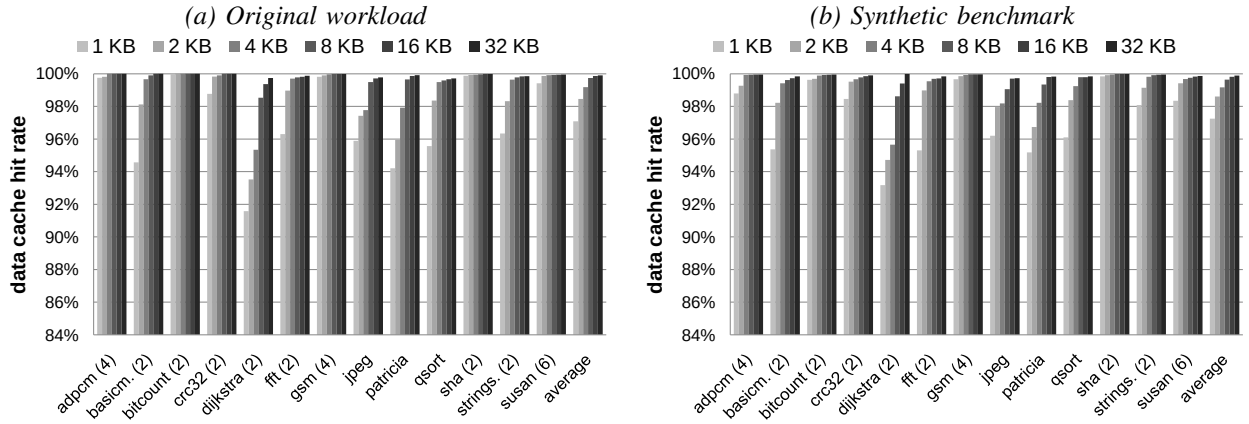

Fig. 7. Data cache hit rates for (a) the original workloads and (b) the synthetic benchmarks at the `-O0` optimization level.

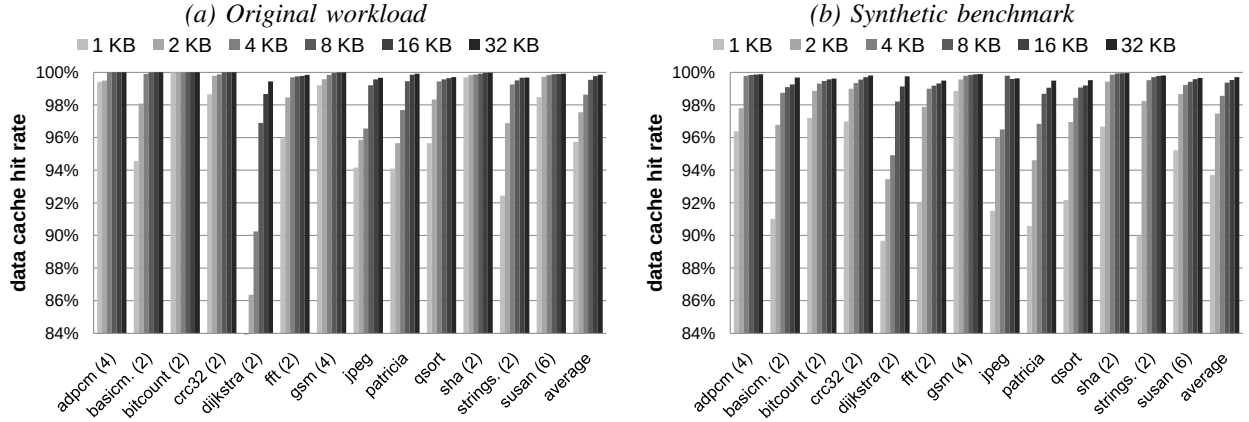

Fig. 8. Data cache hit rates for (a) the original workloads and (b) the synthetic benchmarks at the `-O2` optimization level.

result is that the synthetic workload is able to track that the `-O2` and `-O3` optimization levels yield a substantial 25% performance benefit over `-O1` on the Itanium 2 machine but not on the other machines. This performance benefit for the Itanium architecture is due to the fact that Itanium's EPIC architecture is sensitive to compiler optimizations — an EPIC architecture is a statistically scheduled architecture as opposed to a dynamically scheduled out-of-order processor, hence compiler optimizations may have a more significant impact on overall performance. Clearly, the synthetic workloads expose program constructs similar to the real workloads that enable the compiler to optimize in a similar vein.

E. Benchmark obfuscation

An important asset of benchmark synthesis is that it hides proprietary information, i.e., it is impossible, or at least very hard, to reverse engineer proprietary information from the synthetic benchmark. One way of evaluating whether this is really achieved is through manual inspection. By comparing

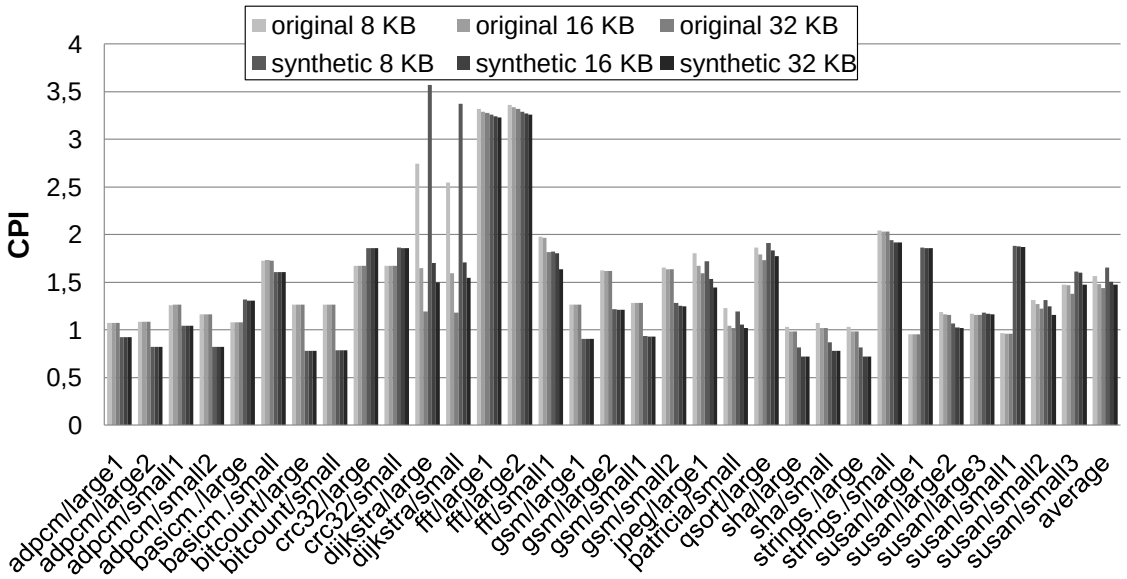

Fig. 10. CPI for the original and synthetic workloads on a 2-wide out-of-order processor while varying cache size.

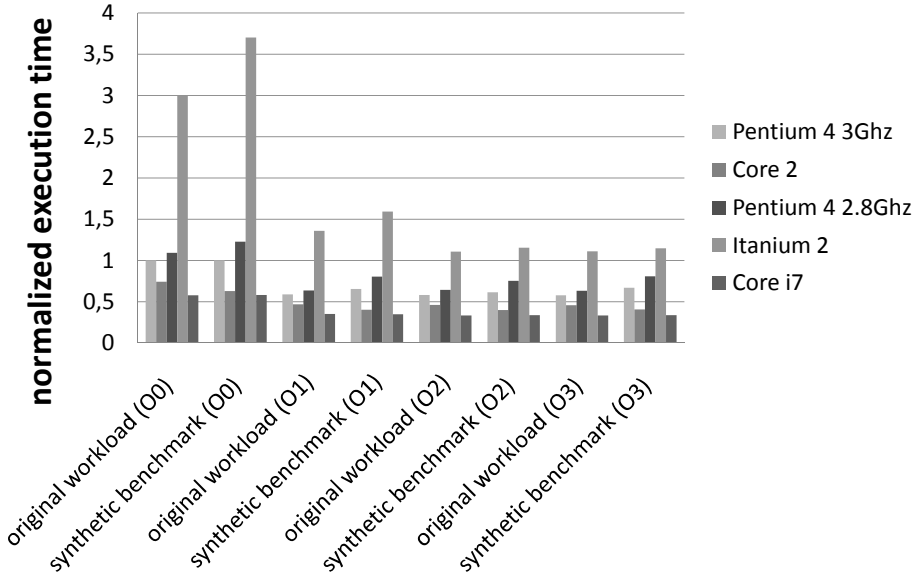

Fig. 11. Normalized average execution time for the original workloads and synthetic clones across architectures and compilers.

the synthetic benchmark against the original workload, one can assess whether any proprietary information is still left in the synthetic benchmark. Presumably, companies that plan on using benchmark synthesis will most likely do this validation process very carefully before distributing a synthetic clone.

We use existing tools for evaluating whether proprietary information is still present in the synthetic benchmark. We therefore use two existing tools, Moss [16] and JPlag [17], which are used to find plagiarism in software. Moss' main usage has been in detecting plagiarism in programming classes; JPlag is aware of programming language syntax and program structure. The way both tools work is that the user gives two source code files, and the tool returns whether there is any similarity between these two files. When giving the original workload and the synthetic benchmark, both Moss and JPlag return that the synthetic benchmark does not provide any similarity with the original workload.

VI. Related work

As mentioned in the introduction, this work shares some commonalities with statistical simulation. Statistical simulation [3], [4], [5] collects program characteristics from a program execution and subsequently generates a synthetic trace from it which is then simulated on a simple, statistical trace-driven processor simulator. The important advantage of statistical simulation is that the dynamic instruction count of a synthetic trace is very short, typically a few millions of instructions at most, making it a useful simulation speedup

technique for quickly identifying a region of interest in a large microprocessor design space. A synthetic trace hides proprietary information very well, however, a synthetic trace cannot be run on real hardware nor on an execution-driven simulator (which is current practice as opposed to trace-driven simulation).

Synthetic benchmarks such as Whetstone [18] and Dhrystone [19] are manually crafted benchmarks. Manually building benchmarks though is both tedious and time-consuming, and in addition, these benchmarks are quickly outdated. Therefore, recent work proposed automated synthetic benchmark generation [6], [20], [7], [21] which builds on the statistical simulation approach but generates a synthetic benchmark rather than a synthetic trace. Van Ertvelde and Eeckhout [22] proposed code mutation which is a different approach to hiding proprietary information. They mutate an existing benchmark so that reverse engineering the benchmark gets more complicated, while preserving similar execution behavior. The advantage of these approaches compared to the original proposals in statistical simulation is that the synthetic benchmarks can be executed on real hardware as well as on execution-driven simulators. However, the benchmarks are generated at the binary level, hence they can be used to drive architecture exploration only; they cannot be used for compiler research and development, nor can they be used for hardware/software co-optimization.

Sampled simulation [23], [24], [25], [26] selects a number of representative samples from the dynamic instruction stream. Simulating these samples instead of the complete dynamic instruction stream yields simulation speedups of several orders of magnitude. Although having only samples to analyze will complicate the understanding the functional semantics of the proprietary application, it may still reveal sensitive information, i.e., if the sampled trace is representative for the entire program execution, it will most likely reveal proprietary information.

Our framework borrows some concepts proposed in prior work. Our proposal extends the statistical flow graph (SFG) [3] with loop information. By doing so, our synthetic benchmarks consist of many (nested) loops as observed in real workloads. Prior work in benchmark synthesis [6] on the other hand, generates a linear sequence of instructions that is iterated in a big loop until convergence. Our framework also borrows the idea of using the branch transition rate for modeling the branch behavior [7] and the stride-based memory access pattern modeling approach [6]. The main difference with this prior work though is that (i) we target synthetic benchmarks in a high-level programming language, whereas prior frameworks generated synthetic traces or benchmarks in assembly, (ii) we generate fine-grained loop structures using the SFGL, and (iii) we use pattern recognition rather than statistics to generate synthetic code sequences.

Code obfuscation [27] converts a program into an equivalent program that is more difficult to understand and reverse engineer. There is a fundamental difference between code obfuscation and benchmark synthesis though. The goal of program obfuscation is to generate a transformed program that is functionally equivalent to the original program, i.e., when given the same input, the transformed program should produce the same output as the original program. The performance characteristics of the transformed program can be — and in practice they are — very different from the original program. Benchmark synthesis on the other hand generates a synthetic program that exhibits the same performance characteristics as the original program, however, its functionality can be very different. Not having to preserve functionality has an important implication for benchmark synthesis because it allows for generating a synthetic benchmark for a specific input, hence benchmark synthesis can also hide proprietary information as part of the input.

VII. Conclusion and future work

This paper proposed a novel benchmark synthesis paradigm that generates synthetic benchmarks in a high-level programming language. It generates small but representative benchmarks that can serve as proxies for other workloads without revealing proprietary information; and because the benchmarks are generated in a high-level programming language, they can be used to explore the architecture and compiler space. Our experimental results are promising and demonstrate the feasibility and effectiveness of the approach. We demonstrate good correspondence between the synthetic and original workloads across instruction-set architectures, microarchitectures and compiler optimizations. This paradigm has many potential applications: distribution of proprietary applications as proxies, drive architecture and compiler research and development, speed up simulation, model emerging and hard-to-setup workloads, and benchmark consolidation.

While the results are promising, there is ample room for improvement and future work. There are various aspects that could be modeled more accurately, such as modeling data dependencies following dependence patterns seen in the original workload, modeling the memory access patterns in a microarchitecture-independent way, etc. However, increasing the representativeness and similarity of the synthetic benchmark with respect to the original workload in terms of its execution behavior, may lead to revealing some proprietary information — there is a trade-off in representativeness versus hiding proprietary information. Being able to generate multi-threaded workloads is another obvious extension to our framework. Subsequently, extending the framework towards more complex workloads, i.e., commercial workloads, is yet another avenue of future work: this will enable the framework to be used in the high-end server market segment as well.

Acknowledgments

We would like the reviewers for their thoughtful comments and suggestions. This research is supported in part by the FWO projects G.0232.06, G.0255.08, and G.0179.10, and the UGent-BOF projects 01J14407 and 01Z04109.

REFERENCES

[1] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization (WWC)*, Dec. 2001.

[2] 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, no. 5, pp. 26–38, Sept/Oct 2003.

[3] L. Eeckhout, R. H. Bell Jr., B. Stougie, K. De Bosschere, and L. K. John, "Control flow modeling in statistical simulation for accurate and efficient processor design studies," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, June 2004, pp. 350–361.

[4] S. Nussbaum and J. E. Smith, "Modeling superscalar processors via statistical simulation," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2001, pp. 15–24.

[5] M. Oskin, F. T. Chong, and M. Farrens, "HLS: Combining statistical and symbolic simulation to guide microprocessor design," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, June 2000, pp. 71–82.

[6] R. Bell, Jr. and L. K. John, "Improved automatic testcase synthesis for performance model validation," in *Proceedings of the 19th ACM International Conference on Supercomputing (ICS)*, June 2005, pp. 111–120.

[7] A. M. Joshi, L. Eeckhout, R. Bell, Jr., and L. K. John, "Distilling the essence of proprietary workloads into miniature benchmarks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 5, no. 2, Aug. 2008.

[8] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, May 1999, pp. 1–9.

[9] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones, "Fast searches for effective optimization phase sequences," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2004, pp. 171–182.

[10] M. Shao, A. Ailamaki, and B. Falsafi, "DBmbench: Fast and accurate database workload representation on modern microarchitecture," in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, Oct. 2005, pp. 254–267.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, June 2005, pp. 190–200.

[12] M. Huangs, P. Sallee, and M. Farrens, "Branch transition rate: A new metric for improved branch claassification analysis," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, Jan. 2000, pp. 241–150.

[13] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Transactions on Computers*, vol. 38, no. 12, pp. 1612–1630, Dec. 1989.

[14] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2007, pp. 23–34.

[15] S. Eyerman, L. Eeckhout, and J. E. Smith, "Studying compiler optimizations on superscalar processors through interval analysis," *Proceedings of the 2008 International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, pp. 320–334, Jan. 2008.

[16] A. Aiken, "Moss: A system for detecting software plagiarism," `http://theory.stanford.edu/~aiken/moss/`.

[17] G. Malpohl, "JPlag: Detecting software plagiarism," `https://www.ipd.uni-karlsruhe.de/jplag/`.

[18] H. J. Curnow and B. A. Wichmann, "A synthetic benchmark," *The Computer Journal*, vol. 19, no. 1, pp. 43–49, 1976.

[19] R. P. Weicker, "Dhrystone: A synthetic systems programming benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013–1030, Oct. 1984.

[20] C. Hsieh and M. Pedram, "Micro-processor 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, Nov. 1998.

[21] C. Hughes and T. Li, "Accelerating multi-core processor design space evaluation using automatic multi-threaded workload synthesis," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, Sept. 2008, pp. 163–172.

[22] L. Van Ertvelde and L. Eeckhout, "Dispersing proprietary applications as benchmarks through code mutation," in *The International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2008, pp. 201–210.

[23] T. M. Conte, M. A. Hirsch, and K. N. Menezes, "Reducing state loss for effective trace sampling of superscalar processors," in *Proceedings of the International Conference on Computer Design (ICCD)*, Oct. 1996, pp. 468–477.

[24] J. Ringenberg, C. Pelosi, D. Oehmke, and T. Mudge, "Intrinsic checkpointing: A methodology for decreasing simulation time through binary modification," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2005, pp. 78–88.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2002, pp. 45–57.

[26] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, June 2003, pp. 84–95.

[27] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," The University of Auckland, Tech. Rep. 148, July 1997.