Dispersing Proprietary Applications as Benchmarks through Code Mutation

Luk Van Ertvelde Lieven Eeckhout

Department of Electronics and Information Systems (ELIS), Ghent University
St.-Pietersnieuwstraat 41, B-9000 Gent, Belgium
{lvertvel,leeckhou}@elis.UGent.be

Abstract

Industry vendors hesitate to disseminate proprietary applications to academia and third party vendors. By consequence, the benchmarking process is typically driven by standardized, open-source benchmarks which may be very different from and likely not representative of the real-life applications of interest.

This paper proposes code mutation, a novel technique that mutates a proprietary application to complicate reverse engineering so that it can be distributed as a benchmark. The benchmark mutant then serves as a proxy for the proprietary application. The key idea in the proposed code mutation approach is to preserve the proprietary application's dynamic memory access and/or control flow behavior in the benchmark mutant while mutating the rest of the application code. To this end, we compute program slices for memory access operations and/or control flow operations trimmed through constant value and branch profiles; and subsequently mutate the instructions not appearing in these slices through binary rewriting.

Our experimental results using SPEC CPU2000 and MiBench benchmarks show that code mutation is a promising technique that mutates up to 90% of the static binary, up to 50% of the dynamically executed instructions, and up to 35% of the at run time exposed inter-operation data dependencies. The performance characteristics of the mutant are very similar to those of the proprietary application across a wide range of microarchitectures and hardware implementations.

Categories and Subject Descriptors C. Computer Systems Organization [*C.4 Performance of Systems*]: Measurement Techniques, Modeling Techniques

General Terms Experimentation, Measurement, Performance

Keywords Benchmark Generation, Code Mutation

1. Introduction

Benchmarking is the key tool for assessing computer system performance. Researchers and computer engineers quantify performance by running a set of benchmarks and by timing the benchmarks' execution times. One use of benchmarking is to compare design alternatives during research or development. Another use of benchmarking is to compare computer systems for guiding purchasing decisions. In order to enable a fair evaluation of computer system performance, organizations such as EEMBC, TPC and SPEC standardize the benchmarking process, and for this purpose these organizations provide industry-standard benchmark suites. For example, SPEC provides the CPU benchmark suite for evaluating the performance of general-purpose processors.

These industry-standard benchmarks are typically derived from open-source programs [11]. The pitfall and limitation though is that these open-source benchmarks may not be representative of real-life applications and may be very different from the application(s) of interest. An alternative would be to use real-life applications of interest. Unfortunately, real-life applications are very often proprietary, and industry hesitates to share them with academia or third party vendors because of intellectual property issues. Nevertheless, it would be in the industry's interest to be able to distribute real-life proprietary applications so that the computer systems are designed to provide good performance for these applications.

This paper proposes *code mutation*, a novel technique that addresses the concern of distributing proprietary applications to third parties. Code mutation first profiles the execution of a proprietary application of interest to collect a variety of properties in a so-called execution profile. This execution profile is then used for binary rewriting the proprietary application into a benchmark mutant. The mutant features two key properties: (i) the functional semantics of the proprietary application cannot be revealed, or, at least, are very hard to reveal through reverse engineering on the mutant, and (ii) the performance characteristics of the mutant resemble those of the proprietary application very well so that the mutant can serve as a proxy for the proprietary application during benchmarking experiments.

In this paper, we make the following contributions:

- We propose code mutation, a novel methodology for constructing benchmarks that hide the functional semantics of proprietary applications while exhibiting similar performance characteristics.
- We explore a number of approaches to code mutation at the binary code level. These approaches differ in the way they preserve the proprietary application's memory access and control flow behavior in the mutant.
- We propose various metrics for quantifying the efficacy of code mutation in terms of how well the mutant hides the functional meaning of the proprietary application.
- We demonstrate that the mutants generated as proxies for the SPEC CPU2000 and MiBench benchmarks hide the functional meaning well, i.e., a mutant mutates up to 90% of the instructions in the application binary, mutates up to 50% of the in-

ASPLOS'08, March 1–5, 2008, Seattle, Washington, USA.
Copyright © 2008 ACM 978-1-59593-958-6/08/03...$5.00

structions in the dynamic instruction stream, and mutates up to 35% of the at run-time exposed data dependencies. These mutations complicate reverse engineering using both the static binary as well as the dynamic instruction stream. In addition, we show that the performance characteristics by the mutants correlate very well with the performance characteristics of the proprietary application across a variety of microarchitectures and hardware implementations.

Code mutation has the potential to become a way of developing benchmarks that can be shared among third party industry vendors, as well as between industry and academia. This would make the benchmarking process in industry both more accurate and more straightforward — no more need to use published SPEC numbers for doing a competitors' analysis for driving purchasing decisions because the benchmarking can be done using a mutant. At the same time it would make performance evaluation in academia more realistic. Instead of driving research based on the performance results obtained using SPEC CPU for example, a representative benchmark mutant may provide more realistic performance assessments, which may eventually lead to more fruitful research directions. In addition, developing benchmarks is both hard and time-consuming to do in academia [24], for which code mutation may be a solution.

2. Code Mutation

2.1 Design options

There are a number of high-level design options that need to be chosen when building a code mutation framework. Before describing our framework in great detail, we first motivate our design choices.

Trace mutation versus benchmark mutation. A first design option is to distribute a mutant in the form of a trace or a benchmark. A trace, or a sequence of dynamically executed instructions, is harder to reverse engineer than a benchmark. Techniques borrowed from statistical simulation [8, 20, 21] to probabilistically instantiate program characteristics in a synthetic trace, or coalescing representative trace fragments [5, 14, 22, 23, 29] could be used to mutate the original trace so that the functional meaning is hidden. A major limitation for a trace mutant though is that it cannot be executed on execution-driven simulators or real hardware. We therefore choose to create benchmark mutants instead of trace mutants.

Input to be provided versus built-in input. A second design option is to mutate the proprietary application so that the resulting mutant can still take its input at run time, or to intermingle the input with the application when creating the mutant. We choose for the latter option for three reasons: (i) intermingling the input and the application enables more aggressive code mutations; (ii) it does not require to distribute a potentially proprietary input; and (iii) it prevents a malicious individual to try reverse engineer the proprietary application by applying different inputs to the mutant.

Binary level versus source code level mutation. Benchmark mutation can be applied at the program source code level or at the binary level. Source code level mutation has the advantage of being easier to port across platforms, and it enables distributing mutants for compiler research and development. Making sure though that the performance characteristics of the mutant correspond well to those of the proprietary application is far from trivial because the compiler may affect the performance characteristics differently for the mutant than for the proprietary application, up to the point where the mutant can no longer serve as a proxy. The latter argument made us choose for benchmark mutation at the binary level: it is easier to maintain the proprietary application's behavioral execution characteristics in the mutant when applying code mutation at the binary level.

Existing program obfuscation techniques. Code obfuscation [4] is a growing research topic of interest which converts a program into an equivalent program that is more difficult to understand and reverse engineer. Benchmark mutation has some properties in common with code obfuscation, however, there are two fundamental differences. First, code obfuscation requires that the obfuscated program version is functionally equivalent to the original program, and produces the same output. A mutant on the other hand does not require to be functionally equivalent, and may even produce meaningless output. Second, a mutant should exhibit similar behavioral characteristics as the proprietary application. An obfuscated program version on the other hand does not have this requirement, and as a matter of fact, many code obfuscation transformations change the behavioral execution characteristics through control flow and data flow transformations and by consequence introduce significant run time overheads, see for example [9]. These differences call for a completely different approach for code mutation compared to code obfuscation.

Bottom-up versus top-down benchmark mutation. Recent work on synthetic benchmark generation [2, 15] extracts a number of program characteristics from a program execution, and subsequently generates a synthetic benchmark from it. Although both synthetic benchmark generation and code mutation serve the same goal which is to distribute proprietary applications as benchmarks, they expose different research challenges. Synthetic benchmark generation is a bottom-up approach for which the research challenge is to identify the key program characteristics that when modeled in the synthetic benchmark, resemble the original application well. Code mutation on the other hand, is a top-down approach that starts from the proprietary application and mutates its code to hide its functional meaning. The advantage of synthetic benchmark generation is that hiding functional meaning is easier, whereas code mutation eases achieving the goal of preserving the behavioral characteristics of the proprietary workload.

2.2 Framework

Figure 1 illustrates the general framework for code mutation. As a first step, we profile the proprietary program execution, i.e., we run the proprietary application along with a proprietary input within an instrumentation framework for collecting a so-called *execution profile*. This execution profile is then used in a second step to transform the application code into a mutant through binary rewriting. The mutant can then be distributed to third parties for benchmarking purposes.

2.2.1 General idea

The challenge at hand is to transform a proprietary program so that the functional meaning is hidden while preserving the behavioral execution characteristics of the proprietary application in the mutant. We started from the observation [16] that performance on contemporary superscalar processors is primarily determined by miss events, i.e., branch mispredictions and cache misses, and to a lesser extent by inter-operation dependencies and instruction types; inter-operation dependencies and instruction execution latencies are typically hidden by superscalar instruction processing. This observation suggests that the mutant, in order to exhibit similar behavioral characteristics as the proprietary workload, should mimick the branch and memory access behavior well without worrying too much about inter-operation dependencies and instruction types. In order to do so, we determine all operations that affect the program's branch and/or memory access behavior; we do this through dynamic program slicing which will be explained later in Section 2.2.3. We retain the operations appearing in these slices unchanged in the mutant, and all other operations in the program

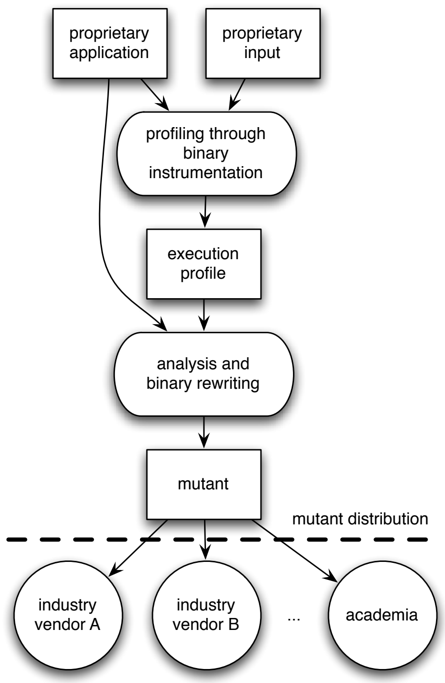


Figure 1. The code mutation framework.

can be overwritten to hide the proprietary application's functional meaning.

We now discuss the various steps in our framework: collecting the execution profile, program analysis and binary rewriting.

2.2.2 Collecting the execution profile

The execution profile consists of three main program execution properties: (i) the inter-operation dependency profile, (ii) the constant value profile, and (iii) the branch profile. The execution profile will be used in the next step by the slicing algorithm for determining which operations affect the branch and/or memory access behavior.

Inter-operation dependency profile. The inter-operation dependency profile captures the data dependencies between instructions. Specifically, it computes the read-after-write (RAW) dependencies between instructions through both registers and memory. The inter-operation dependency profile then enumerates all the static instructions that a static instruction depends upon (at least once) at run time.

For example, consider the example given in Figure 2, where instruction (d) consumes registers `r3` and `r6`; instruction (a) produces `r3`, and both instructions (b) and (c) produce `r6`. If it turns out that the path shown through the tick black line (A-B-D) is always executed, i.e., basic block C is never executed, then only instructions (a) and (b) will appear in the inter-operation dependency profile for instruction (d). Instruction (c) will not appear in the dependency profile because basic block C is never executed in this particular execution of the program. If, in contrast, basic block C would be executed at least once, then (a), (b) and (c) will appear in the dependency profile for instruction (d).

Constant value profile. For each static instruction we profile whether the register values that it consumes and the register value that it produces are constant over the entire program execution. In other words, for each instruction, we keep track of the register values it consumes and produces, and store the constant value register operands in the execution profile.

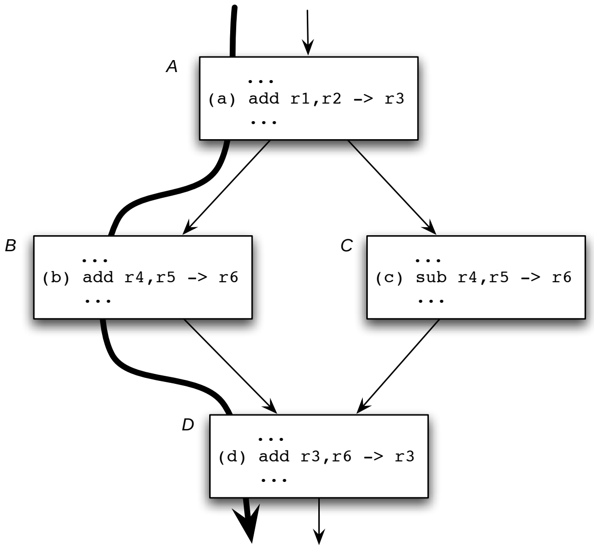


Figure 2. An example illustrating the inter-operation dependency profile.

Value locality [17], value profiling [3], and its applications have received research interest over the past decade. Various authors have reported that a substantial fraction of all variables produced by a program at run time is invariant, see for example [3]. There are several ways of leveraging invariant (and semi-invariant or predictable) data values such as hardware value prediction, code specialization, partial evaluation, and adaptive and dynamic optimization. We will use the constant value profiles to trim the program slices as we will explain in Section 2.2.3.

Branch profile. The branch profile captures a number of profiles concerning a program's control flow behavior:

- We store the branch direction in case a conditional branch is always taken or always not-taken.
- We store the branch target in case an indirect jump always branches to the same target address.
- In case of an unconditional jump, we determine a condition flag that is constant at the jump point across the entire program execution.
- And finally, we also capture the taken/not-taken sequence for infrequently executed conditional branches, or in case the branch is executed less than 32 times during the entire program execution.

The execution profile can be collected through binary instrumentation such as ATOM [25] or PIN [18]. We use PIN in our framework. Dynamic binary instrumentation using PIN adds instrumentation code to the application code as it runs that measures the program execution properties of interest.

Non-deterministic system calls complicate the computation of the constant value and branch profiles: a variable in one program run may have a different value in another program run with the same input. We take a pragmatic solution and run each program multiple times and then subsequently compute the constant value profiles across these program runs. A more elegant solution may be to record/replay system effects as done in pinSEL [19].

2.2.3 Program analysis

We use the execution profile to compute the operations that affect the branch and memory access behavior of a program execution. Computing these operations is done through program slicing.

Program slicing. As alluded to before, we use *program slicing* [26, 28], a powerful technique for tracking chains of dependencies between operations in a program. Program slicing is found useful for various purposes in software development, testing and debugging, as well as in optimizing performance through identifying critical operations [32]. A *program slice* consists of the instructions that (potentially) affect the values computed at some point of interest in the program execution, referred to as the *slicing criterion*. In this work we consider *backward* slices, or the sequence of instructions leading to the slicing criterion. The backward slice, or slice for short, can be computed through a backward traversal of the program starting at the slicing criterion. An important distinction is to be made between a *static* and a *dynamic* slice. The former does not make any assumptions about the program's input whereas the latter relies on a specific test case.

Our framework uses dynamic slicing because we have a specific proprietary input available, and because dynamic slices are typically thinner, i.e., contain fewer instructions, than static slices. This enables us to more aggressively apply code mutation. We use an imprecise dynamic slicing algorithm because of the high computational complexity both in time and space of precise dynamic slicing [31], especially for computing many slices for long-running applications. The slices produced through imprecise slicing are less accurate than through precise slicing, nevertheless they are conservative meaning that they are a superset of precise slices. We use Algorithm II as described by Agrawal and Horgan [1] and Zhang et al. [31]. This algorithm starts from the slicing criterion and recursively builds the backward slice using the inter-operation dependency profile: starting from the dependency profile for the slicing criterion, it recursively computes prior dependencies. The computational cost for this imprecise slicing algorithm is independent of the number of slices that need to be computed [31]. This is an important benefit for our purpose since we compute slices for all control flow operations and/or data memory accesses, as explained in the next section.

The constant value and branch profiles help trimming both the number of slices as well as the size of the slices that need to be computed. Specifically, we do not need to compute slices for branches that are either always taken or always not-taken. In addition, the recursive backward dependency tracking for computing the slices stops upon a constant value.

Code marking. In our evaluation, we consider two scenarios with different slicing criteria.

Memory access and control flow operation (MA-CFO) slicing: The first scenario computes slices for all control flow operations as well as for all effective data addresses generated through loads and stores. This scenario will ensure that the control flow and memory access behavior of the mutant will be identical to the proprietary application.

Control flow operation (CFO) slicing: The second scenario only computes slices for all control flow operations. This criterion will be less accurate than the former because it does not compute slices for data memory accesses. This has the potential benefit of enabling much aggressive code mutation at the potential cost of the mutant being less representative of the proprietary application.

Once the slices are computed, all the instructions not part of a slice are marked. This includes code that is never executed as well as code that gets executed but produces unused data (dead code). In addition, all instruction operands (input as well as output operands) that hold constant values at run time are marked as such.

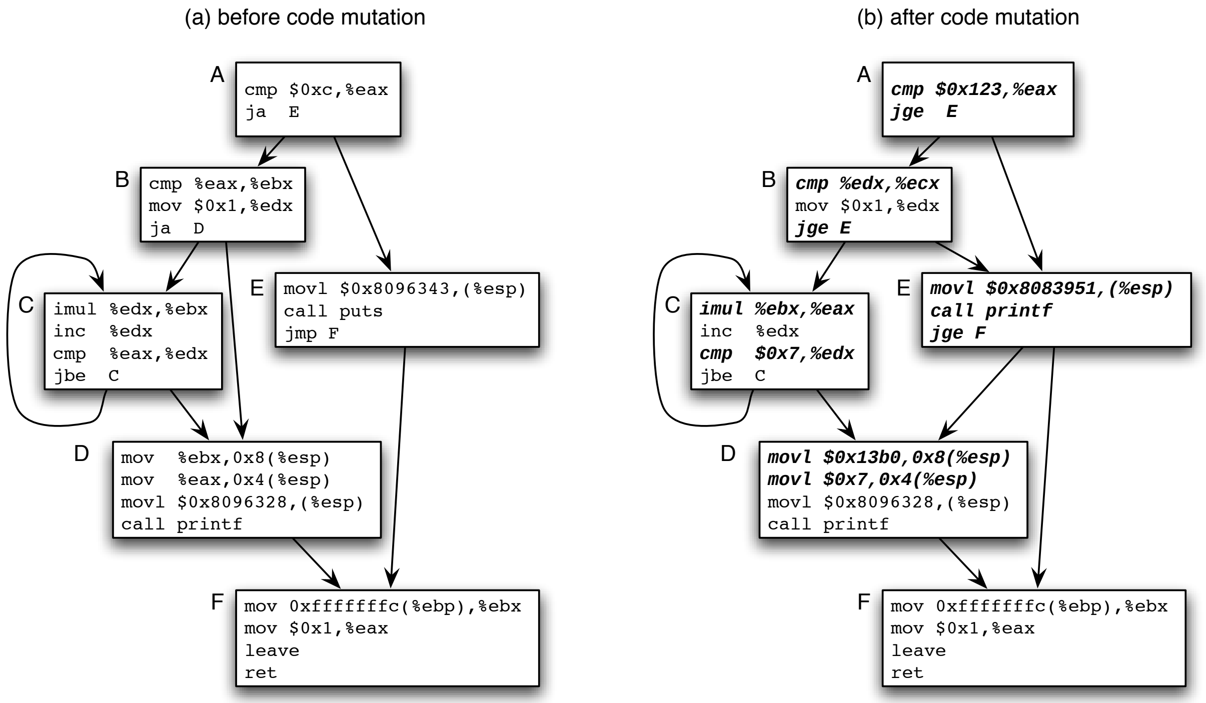
2.2.4 Binary rewriting

Once the code marking is done, we then perform the actual code mutation. We employ binary rewriting to mutate the proprietary application into a benchmark mutant; we use Diablo [7] as our binary rewriting tool. Applying mutation through binary rewriting poses some challenges in terms of preserving the code layout. Rewriting instructions may change the code layout and may thereby affect the instruction cache performance. We therefore strive at keeping the basic block size the same before and after mutation.

Control flow mutations. As mentioned before, we do not compute slices for branches that have a constant branch target. Instead, we mutate those branches to complicate the understanding of the mutant. To do so, we use an opaque variable or predicate [4]. An opaque variable is a variable that has some property that is known a priori to the code mutator, but which is difficult for a malicious person to deduce. In our work, we use as the opaque variable a condition flag that has a constant value at the branch point during the course of the program execution but which is different from the condition flag in the original program. We mutate conditional branches that are either always taken or always not-taken, indirect jumps with a constant branch target, and we also convert unconditional branches into conditional branches. Conditional branches that jump based on an opaque condition flag do not alter the execution flow of the mutant but complicate the understanding of the mutant binary. In addition, control flow edges that are never taken are altered in the mutant.

Rewriting code and breaking data dependencies. The unexecuted and dead code (the marked instructions) is overwritten. The rewriting is done by randomly reassigning an instruction type, and register input and output operands. This random reassignment assures that there is no one-to-one mapping of instruction types and operands which makes reverse engineering impossible, or at least very difficult. Nevertheless, for dead code, i.e., code that gets executed but which produces unused data, we ensure that the instruction mix, i.e., the relative occurrence of instruction types, in the mutant is similar to that in the proprietary application so that the run time behavior characteristics of the mutant match those of the proprietary application. Likewise, we generate inter-operation data dependencies in such a way that the distribution of inter-operation dependencies of the mutant is similar to the one of the proprietary application in order to preserve the amount of ILP in the mutant. The instruction operands that are marked as holding constant values are replaced by immediate constants. By doing so, we break inter-operation dependencies making it harder to understand the proprietary application. For the output register operands, we use non-live register operands to make sure the inserted code mutations do not affect the execution flow of the mutant.

Example. Figure 3 illustrates code mutation that preserves the control flow behavior on a simple example that computes the factorial of 7. The input to the function, which is '7', is hold in register `%eax`. Basic block A checks whether the input is larger than 12. If yes, error handling code is executed in E; if no, the factorial is computed and printed in B, C, and D. Figure 3(a) shows the original code, and (b) shows the code after code mutation; the instructions shown in bold italics in (b) are overwritten. The branch instruction in A is a conditional branch that is never taken for the given input '7': we overwrite this branch as well as the `cmp` instruction in its slice. E is never executed, and by consequence, we can completely overwrite E. Also the conditional branch in B is never taken — the input differs from '1' — and we thus overwrite the branch and the instructions in its slice. In C, both `%eax` and `%edx` from the `cmp`



(a) before code mutation

```
A:
cmp $0xc,%eax
ja  E

B:
cmp %eax,%ebx
mov $0x1,%edx
ja  D

C:
imul %edx,%ebx
inc  %edx
cmp  %eax,%edx
jbe  C

D:
mov  %ebx,0x8(%esp)
mov  %eax,0x4(%esp)
movl $0x8096328,(%esp)
call printf

E:
movl $0x8096343,(%esp)
call puts
jmp F

F:
mov 0xfffffffc(%ebp),%ebx
mov $0x1,%eax
leave
ret
```

(b) after code mutation

```
A:
cmp $0x123,%eax
jge  E

B:
cmp %edx,%ecx
mov $0x1,%edx
jge E

C:
imul %ebx,%eax
inc  %edx
cmp  $0x7,%edx
jbe  C

D:
movl $0x13b0,0x8(%esp)
movl $0x7,0x4(%esp)
movl $0x8096328,(%esp)
call printf

E:
movl $0x8083951,(%esp)
call printf
jge F

F:
mov 0xfffffffc(%ebp),%ebx
mov $0x1,%eax
leave
ret
```

Figure 3. An example, the factorial function with input '7', illustrating the mechanism of code mutation: (a) before code mutation, and (b) after code mutation.

instruction appear in the slice for the conditional branch at the end of C. However, the value for `%eax` is invariant for this particular execution, and we thus overwrite the `%eax` argument by its constant value which is 7. For register `%edx`, the slice includes the `cmp` and `inc` instructions in C and the `mov` instruction in B; these instructions thus remain unchanged in the mutant. The values in `%eax` and `%ebx` in D are constant, and are overwritten by constant values. As a result of that, we can overwrite the `imul` instruction in C. The end result of code mutation is a mutant, shown in Figure 3(b), that looks fairly different from the original application shown in Figure 3(a). It will be very hard to reveal the functional meaning of the original application from its mutant.

Infrequent branches. We do not compute slices for infrequent branches, in our case, conditional branches that are executed less than 32 times over the entire program execution. For those branch instructions we record the branch taken/not-taken sequence in the branch profile (as mentioned above), and replay this branch sequence in the mutant at run time. We refer to this transformation as *enforced control flow (ECF)*. ECF is illustrated in Figure 4 through a code snippet, a basic block from the crafty benchmark. We enforce the taken/not-taken behavior for the conditional branch by loading the branch sequence from memory, shift left the branch sequence, and store it back to memory; the sign bit then determines the branch direction, see instructions (4) through (7) in Figure 4(b). An important benefit of ECF is that it increases the number of instructions eligible for code mutation. For example, in Figure 4, the instructions (1) through (3) are mutated because of not having to retain these instructions in the slice leading up to the conditional branch.

Figure 5 quantifies for what fraction of branches in the static binary we can apply ECF — we will describe the experimental setup later in Section 4. We can apply ECF to about 18% of the branches. Together with the on average 54% of the branches that

(a) original application: before ECF

```
(1)   and   %edx,%eax
(2)   mov   0x154(%esp),%ebx
(3)   mov   %eax,0x150(%esp)
(4)   or    0x150(%esp),%ebx
(5)   jne   0x807026f
```

(b) benchmark mutant: after ECF

```
(1)   xor   %ebx,%eax
(2)   mov   0x154(%esp),%ecx
(3)   mov   %ebx,0x150(%esp)
(4)   mov   0x80637e9,%ecx
(5)   shl   %ecx
(6)   mov   %ecx,0x80637e9
(7)   js    0x80639f6
```

Figure 4. Example basic block from crafty illustrating the Enforced Control Flow approach: (a) before ECF, (b) after ECF.

are always taken or not-taken, this means that we need to compute slices for only 28% of the branches on average.

3. Quantifying Mutation Efficacy

There are two issues when quantifying the efficacy of code mutation. The first one concerns with how well the performance characteristics of the proprietary application are preserved in the mutant. This is straightforward to do: this can be done by comparing performance numbers of the mutant against the proprietary application across a number of microarchitectures and hardware platforms. The second issue is much more challenging and concerns with how well the functional meaning of the proprietary application is hidden

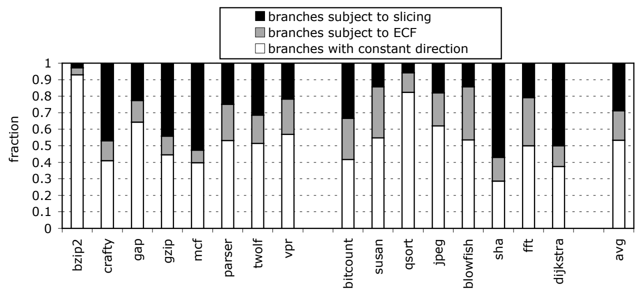


Figure 5. Fraction branches in the binary (i) with constant taken/not-taken direction, (ii) that are subject to ECF, and (iii) that are subject to slicing.

by the mutant. An industry vendor will only release a proprietary application as a benchmark mutant based on a thorough efficacy evaluation. This section discusses metrics for quantifying how well the functional meaning of a proprietary application is hidden by the mutant.

There exists a vast body of work on software complexity metrics which typically count various textual properties of the source code into a complexity measure. Code obfuscation uses some of those metrics to quantify the efficacy of code transformations [4]. These metrics relate to code size, data flow complexity, control flow complexity, data structure complexity, etc. These metrics only partially achieve what we want a mutation efficacy metric to measure. Recall that the aim of code obfuscation is to complicate the understanding of the proprietary application, however, the obfuscated program is still functionally equivalent to the original application. This means that the obfuscated program preserves the semantics of the original application, and, intuitively speaking, still contains the same 'information' as the original application — code obfuscation just makes it harder to grasp the proprietary 'information'. Code mutation goes one step further in the sense that a mutant removes 'information' from the proprietary application through binary rewriting, i.e., the mutant is no longer functionally equivalent to the proprietary application.

These considerations call for metrics specifically targeted towards quantifying how well code mutation hides the proprietary application. We therefore develop a set of metrics to quantify mutation efficacy. The first three metrics count instructions that are mutated, i.e., the fraction instructions that appear in a different format in the mutant than in the proprietary application. The next two count the number of data dependencies that are broken in the mutant with respect to the proprietary application — hiding data dependencies and introducing artificial data dependencies complicate the understanding of the mutant. In this work, when reporting these metrics, we only report about the application, not the libraries — the reason is to stress code mutation in the evaluation because most library code is only infrequently executed (if at all).

Static Instruction Ratio (SIR): The SIR computes the ratio of the number of instructions in the binary that are mutated to the total number of instructions in the binary.

Instruction Ratio (IR): The IR computes the ratio of the number of instructions in the binary that are mutated and executed at least once, relative to the number of instructions in the binary that are executed at least once.

Weighted Instruction Ratio (WIR): The WIR computes the ratio of the number of instructions that are mutated, weighted with their execution frequency relative to the total dynamic instruction count. In other words, the WIR computes the fraction mutated instructions executed relative to the dynamic instruction count.

Dependence Ratio (DR): The DR metric computes the fraction inter-operation data dependencies that appear at least once at run time and that are broken.

Weighted Dependence Ratio (WDR): The WDR metric weighs the DR metric with the execution frequency for each of the dependencies.

suite	benchmark	input	cnt(M)
SPEC CPU2000	bzip2	lgred.source	1,417
	crafty	lgred	940
	gap	lgred	50
	gzip	smred.log	452
	mcf	lgred	87
	parser	lgred	288
	twolf	lgred	781
	vpr	small.arch	214
MiBench	bitcount	1125000	713
	susan	large	754
	qsort	large	371
	jpeg	large	85
	blowfish	large	973
	sha	large	230
	fft	8 32768	298
	dijkstra	large	281

Table 1. Benchmarks used in this study along with their inputs and dynamic instruction counts (in millions).

The SIR is a metric that quantifies the efficacy of code mutation for making *static reverse engineering* hard, i.e., reverse engineering of the proprietary application by inspecting the binary of the benchmark mutant. The other four metrics quantify the efficacy for making *dynamic reverse engineering* hard, i.e., reverse engineering by inspecting the dynamic execution of the mutant.

4. Experimental Setup

In our evaluation we consider a number of general-purpose SPEC CPU2000 benchmarks as well as a number of benchmarks from the embedded MiBench benchmark suite [10]. The benchmarks are tabulated in Table 1 along with their inputs and dynamic instruction counts. For the SPEC CPU2000 benchmarks, we use MinneSPEC inputs in order to limit the simulation time of complete benchmark executions[1]. For MiBench, we consider the largest input available. All the benchmarks are compiled on an x86 platform (Intel Pentium 4 running Linux) using the `gcc` compiler version 3.2.2 with optimization level `-O3`; all binaries are statically compiled.

The baseline processor configuration is tabulated in Table 2. We model a 4-wide superscalar out-of-order processor with a three-level cache hierarchy. The simulations are done using PTLsim [30], an execution-driven x86 superscalar processor simulator.

We also provide real hardware performance results and consider three Intel Pentium 4 machines. These machines differ in terms of their clock frequency, microarchitecture, memory hierarchy, and implementation technology; see Table 3 for the most significant differences.

[1] We were unable to include all the SPEC CPU2000 integer benchmarks because of difficulties in interoperating the various tools (PIN, Diablo and PTLsim) in our experimental setup.

ROB	128 entries
load queue	48 entries
store queue	32 entries
issue queues	4 16-entry issue queues
processor width	4 wide fetch, decode, dispatch, issue, commit
latencies	load (2), mul (3), div (20)
L1 I-cache	16KB 4-way set-assoc, 1 cycle
L1 D-cache	16KB 4-way set-assoc, 1 cycle
L2 cache	unified, 128KB 16-way set-assoc, 6 cycles
L3 cache	unified, 1MB 16-way set-assoc, 20 cycles
main memory	250 cycle access time
branch predictor	hybrid bimodal/gshare predictor
frontend pipeline	8 stages

Table 2. Baseline processor model assumed in our simulations.

processor	generation	freq	L2 size	MEM size
machine 1	Northwood	2.0 GHz	512 KB	1 GB
machine 2	Northwood	2.8 GHz	512 KB	2 GB
machine 3	Prescott	3.0 GHz	1 MB	1 GB

Table 3. The Intel Pentium 4 machines considered in our setup.

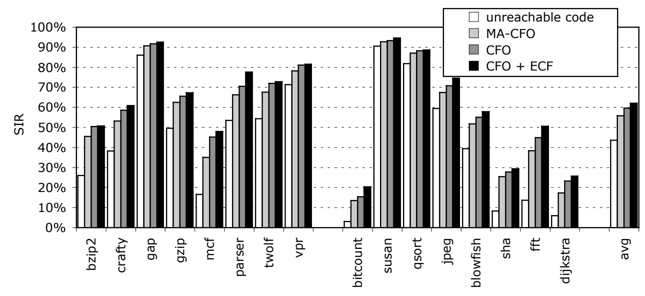


Figure 6. Quantifying the efficacy of benchmark mutation using the SIR metric.

5. Evaluation

We now evaluate the proposed code mutation approaches: (i) MA-CFO (memory access and control flow operation slicing) aiming at preserving the memory access and control flow behavior in the mutant, (ii) CFO (control flow operation slicing) aiming at preserving only the control flow behavior in the mutant, and (iii) CFO plus ECF (CFO plus enforced control flow of infrequent branches). We evaluate the efficacy of these approaches along two dimensions: their ability to hide functional semantics, and their ability to preserve the performance characteristics in the mutant with respect to the proprietary application.

5.1 Hiding functional semantics

SIR. Figure 6 quantifies the SIR metric, or the fraction of the application binary that can be mutated, which is an indication for how well binary mutation protects against static reverse engineering. There are four bars in this graph. The first bar quantifies the SIR metric by overwriting unexecuted code. On average, this results in a 44% SIR metric. The next three bars quantify the SIR metric for MA-CFO, CFO, and CFO plus ECF code mutation; these approaches achieve a SIR metric of 56%, 60% and 62%, respectively. CFO achieves a higher SIR score than MA-CFO, and CFO plus ECF achieves a higher SIR score than CFO. The reason is that fewer slices need to be computed which increases the number of instructions eligible for code mutation. The relative increase is limited though: we expected that the SIR metric would be much higher for CFO compared to MA-CFO. However, the relatively small increase seems to suggest that there is significant overlap between the slices of memory accesses and the slices of control flow operations. Not computing memory access slices does not reveal that many additional instructions eligible for code mutation. Put another way, by striving at preserving a program's control flow behavior, we also preserve most of the memory access behavior. Another interesting note is that the achievable SIR is benchmark specific, and for some benchmarks more than 90% of the application binary can be mutated, see for example gap, susan and qsort. The high SIR score for susan and qsort correlates well with the small number of branches subject to slicing, see Figure 5. For other benchmarks though, the small number of branches subject to slicing does not translate into a high SIR score, see for example bzip2: a small number of control flow slices cover a large fraction of the entire program code.

IR and WIR. Figures 7 and 8 report similar results for the IR and WIR metrics, which are measures for how well the code mutation protects against dynamic reverse engenieering. The IR and WIR metrics have lower values than the SIR metric: average IR and WIR scores of 36% and 20%, respectively, compared to the average 62% SIR score for CFO plus ECF code mutation. Also, the WIR metric is typically lower than the IR. This suggests that code mutation primarily mutates code in less frequently executed code regions. The susan benchmark is an extreme example which has an SIR metric of 95%, an IR metric of 43% and a WIR metric of 8%. For other benchmarks on the other hand, such as qsort, code mutation mutates frequently executed code, and achieves a WIR score (53%) that is higher than its IR score (38%).

DR and WDR. Figures 9 and 10 show the DR and WDR metrics, respectively. The average DR and WDR metric scores are 29% and 16%, respectively, and goes up to 40% and 35%, respectively. This result shows that a substantial fraction of the at run-time exposed

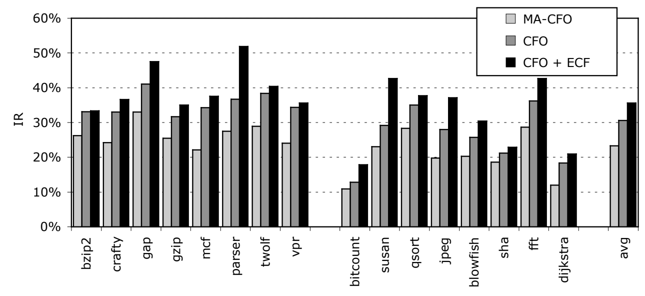


Figure 7. Quantifying the efficacy of benchmark mutation using the IR metric.

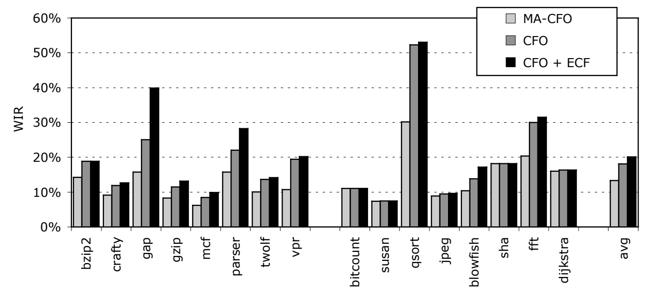


Figure 8. Quantifying the efficacy of benchmark mutation using the WIR metric.

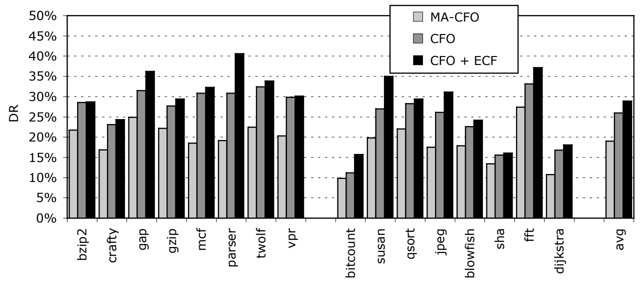


Figure 9. Quantifying the efficacy of benchmark mutation using the DR metric.

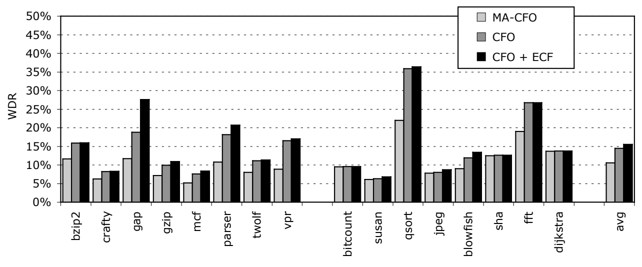


Figure 10. Quantifying the efficacy of benchmark mutation using the WDR metric.

data dependencies are broken through code mutation, which will complicate reverse engineering.

5.2 Preserving performance characteristics

We now evaluate how well the mutant preserves the performance characteristics of the original application. We do this in three steps. We first consider our baseline simulated processor configuration, and subsequently evaluate how well the mutant tracks the original application across a microarchitecture design space. Finally, we consider three real hardware platforms.

Simulation results. Figure 11 quantifies the execution time deviation for the mutant compared to the original application. The average (absolute) performance deviation equals 0.7%, 1% and 1.2% for MA-CFO, CO, and CFO plus ECF, respectively. The maximum performance deviation is limited to 6%, see qsort which is also the benchmark with the highest WIR and WDR metric values.

Microarchitecture design space. We also evaluated code mutation across a microarchitecture design space in which we vary the processor width and cache hierarchy. We vary the width from 2 up to 8, and consider four (L1/L2/L3) cache hierarchies: config 1: 8KB, 64KB, 512KB; config 2: 16KB, 128KB, 1MB; config 3: 32KB, 256KB, 2MB; and config 4: 64KB, 512KB, 4MB. The average deviation across this design space equals 0.8%, 1.3% and 1.4% for MA-CFO, CFO, and CFO plus ECF, respectively. Figure 12 illustrates this further: normalized execution time is shown across the four cache hierarchy configurations for a four-wide superscalar processor. The mutant tracks the performance sensitivities across the memory hierarchy very well.

Real hardware results. The results presented so far are obtained from simulation. Figures 13 and 14 show results obtained from real hardware measurements on three Intel Pentium 4 machines, see Table 3. Figure 13 quantifies the performance deviation of the mutant with respect to the original application on the Prescott Intel Pentium 4 (machine 3). The execution time deviation is small: 1.4% on average; the maximum deviation 6% is observed for qsort.

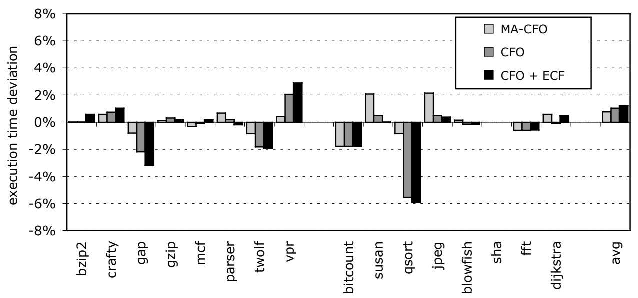


Figure 11. Execution time deviation for the mutant against the original application for the baseline processor configuration.

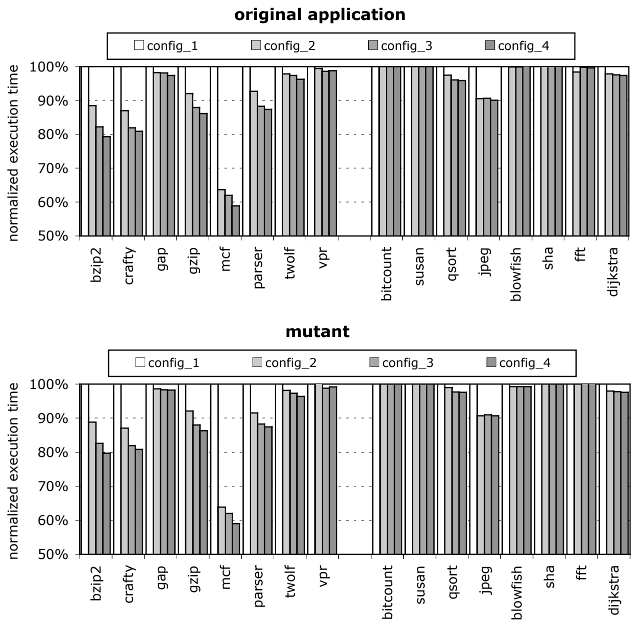


Figure 12. Normalized execution time for the original application (top graph) and its mutant assuming CFO plus ECF (bottom graph) across four cache hierarchy configurations for a four-wide superscalar processor.

Figure 14 shows normalized execution times across the three Intel Pentium 4 machines for the original applications are their mutants; the results are normalized to the execution time of the original application on machine 3. These results show that the mutants track the relative performance differences of the original application very well across the different hardware platforms.

6. Related Work

Statistical simulation [8, 20, 21] 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 several orders of magnitude smaller than for today's industry-standard benchmarks, making it a useful simulation speedup technique for quickly identifying a region of interest in a large microprocessor design space. Because the synthetic trace is generated from characteristics, it is very hard to reveal the functional

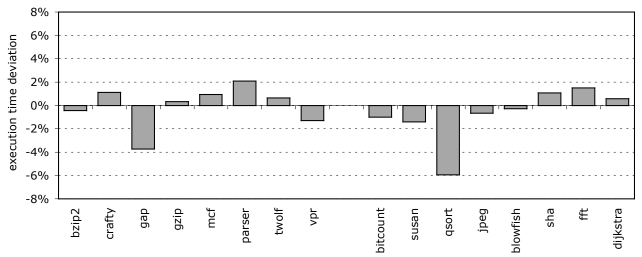


Figure 13. Execution time deviation for a mutant (CFO plus ECF) against its original application for 'machine 3', a 3.0 GHz Prescott Intel Pentium 4 machine.

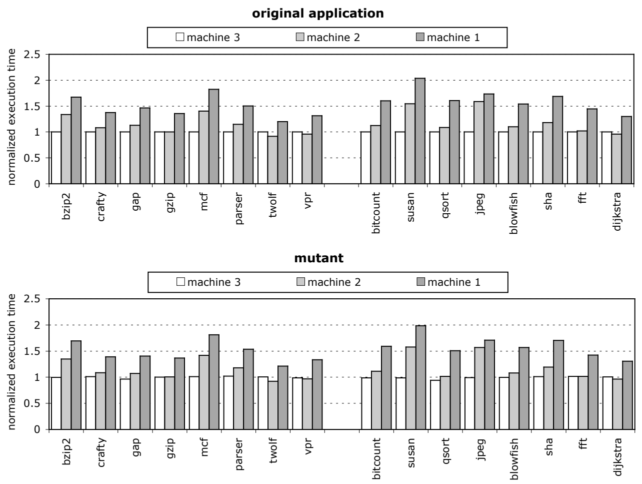


Figure 14. Normalized execution time for three real hardware platforms for the original application (top graph) and the mutant assuming CFO plus ECF (bottom graph); the results are normalized to the execution time of the original application on machine 3.

semantics of the proprietary application from a synthetic trace. The limitation of statistical simulation as a benchmark mutation approach though is that the synthetic trace cannot be simulated on an execution-driven simulator or real hardware.

Synthetic benchmarks such as Whetstone [6] and Dhrystone [27] are manually crafted benchmarks that aimed at representing real workloads. Manually building benchmarks though is both tedious and time-consuming. Therefore, recent work proposed automated synthetic benchmark generation [2, 13, 15] which builds on the statistical simulation approach but generates a synthetic benchmark rather than a synthetic trace. Synthetic benchmark generation is a bottom-up approach whereas code mutation is a top-down approach, as discussed before.

Hoste et al. [12] take a different approach for estimating the performance of a (proprietary) application of interest. They propose to characterize the application of interest, and compare its behavioral characteristics against those of a well known benchmark suite, such as SPEC CPU. The performance of the application of interest is then estimated by weighting the performance numbers of the benchmarks proportional to their similarity with the application of interest. This approach does not need to distribute the proprietary application, but builds on the implicit assumption that the application of interest is similar to (some of) the benchmarks in the benchmark suite.

Sampled simulation theory [5, 14, 22, 23, 29] 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 valuable information.

7. Summary and Future Work

Summary. This paper presented code mutation, a novel methodology for deriving benchmarks from proprietary applications by hiding functional semantics while preserving performance characteristics. Code mutation will be most useful for companies that offer (in-house built) services to remote customers. Such companies are reluctant to distribute their proprietary software. As an alternative, they can use mutated benchmarks as proxies for their proprietary software. The mutated benchmarks can help drive performance evaluation by third parties as well as guide purchasing decisions of hardware infrastructure.

The best performing code mutation approach proposed in this paper computes control flow slices for frequently executed, non-constant branches; and mutates instructions that are not part of any of these slices. The slices are trimmed using constant value profiles. Our results obtained for a selection of SPEC CPU2000 and MiBench benchmarks report that up to 90% of the binary can be mutated, up to 50% of the dynamic executed instructions, and up to 35% of the at run time exposed inter-operation data dependencies.

Future work. We believe that code mutation is a promising technique for dispersing proprietary applications as benchmarks, and there are various future research avenues to be explored. First, future work could further improve the information hiding in the mutant by employing novel and more aggressive program analyses and transformations. One potential direction could be to exploit semi-invariant program behavior (next to invariant or constant program behavior) in order to mutate an even larger fraction of the proprietary application.

Second, our current framework mutates the proprietary application at the binary code level. An alternative approach would mutate the application at an intermediate code level or even at the source code level, so that the mutant can be compiled and optimized for a particular ISA of interest. This would broaden the applicability of the mutants to compiler and code generation research and development.

Third, the execution time of a mutant is very similar to the execution time of the original proprietary application, on purpose. This is a viable solution for real hardware performance evaluation, but for simulation purposes one may want to have shorter running mutants. An interesting research challenge thus is to reduce the dynamic instruction count of the mutant while preserving the performance characteristics of the proprietary application.

Finally, extending the code mutation concept to multi-threaded workloads as well as applications written in type-safe managed languages (such as Java and C#) is also part of our future work. We believe that both are possible — the general concept of code mutation applies to these workloads as well while posing a number of additional constraints. In particular, multi-threaded workloads incur an additional constraint in that accesses to shared memory should be preserved in the mutated benchmark in order to exhibit similar inter-thread communication in the mutant as in the original application. As such, slices will need to be computed for shared memory accesses, and instructions appearing in these slices should not be overwritten through code mutation. For type-safe managed languages, code mutation will be restricted by type safety, i.e., an operation can only be overwritten by another operation if both operate on the same type.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. This work is supported in part through the UGent-BOF project 01J14407, FWO project G.0255.08, and HiPEAC. Lieven Eeckhout is supported by a Postdoctoral Fellowship with the Fund for Scientific Research – Flanders (Belgium).

References

[1] H. Agrawal and J. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 246–256, June 1990.

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

[3] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 259–269, Dec. 1997.

[4] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, The University of Auckland, July 1997.

[5] 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)*, pages 468–477, Oct. 1996.

[6] H. J. Curnow and B. A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43–49, 1976.

[7] B. De Bus, D. Kaestner, D. Chanet, L. Van Put, and B. De Sutter. Post-pass compaction techniques. *Communications of the ACM*, 46(8):41–46, Aug. 2003.

[8] 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 Annual International Symposium on Computer Architecture (ISCA)*, pages 350–361, June 2004.

[9] J. Ge, S. Chaudhuri, and A. Tyagi. Control flow based obfuscation. In *Proceedings of the ACM Workshop on Digital Rights Management (DRM)*, pages 83–92, Nov. 2005.

[10] 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 Annual Workshop on Workload Characterization (WWC)*, Dec. 2001.

[11] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[12] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 114–122, Sept. 2006.

[13] 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*, 17(11):1080–1089, Nov. 1998.

[14] V. S. Iyengar, L. H. Trevillyan, and P. Bose. Representative traces for processor models with infinite cache. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture (HPCA)*, pages 62–73, Feb. 1996.

[15] A. M. Joshi, L. Eeckhout, R. H. Bell Jr., and L. K. John. Performance cloning: A technique for disseminating proprietary applications as benchmarks. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 105–115, Oct. 2006.

[16] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 338–349, June 2004.

[17] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 138–147, Oct. 1996.

[18] 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)*, pages 190–200, June 2005.

[19] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application level architecture simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 216–227, June 2006.

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

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

[22] 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)*, pages 78–88, Mar. 2005.

[23] 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)*, pages 45–57, Oct. 2002.

[24] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai. Challenges in computer architecture evaluation. *IEEE Computer*, 36(8):30–36, Aug. 2003.

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

[26] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[27] R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, Oct. 1984.

[28] M. Weiser. Program slicing. *IEEE Transaction on Software Engineering*, 10(4):352–357, July 1984.

[29] 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)*, pages 84–95, June 2003.

[30] 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)*, pages 23–34, Apr. 2007.

[31] X. Zhang, R. Gupta, and Y. Zhang. Cost and precision tradeoffs of dynamic data slicing algorithms. *ACM Transactions on Programming Languages and Systems*, 27(4):631–661, July 2005.

[32] C. B. Zilles and G. S. Sohi. Understanding the backward slices of performance degrading instructions. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 172–181, June 2000.