

STABILIZER: Statistically Sound Performance Evaluation

Charlie Curtsinger Emery D. Berger

Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
{charlie,emery}@cs.umass.edu

Abstract

Researchers and software developers require effective performance evaluation. Researchers must evaluate optimizations or measure overhead. Software developers use automatic performance regression tests to discover when changes improve or degrade performance. The standard methodology is to compare execution times before and after applying changes.

Unfortunately, modern architectural features make this approach unsound. Statistically sound evaluation requires multiple samples to test whether one can or cannot (with high confidence) reject the null hypothesis that results are the same before and after. However, caches and branch predictors make performance dependent on machine-specific parameters and the exact layout of code, stack frames, and heap objects. A single binary constitutes just one sample from the space of program layouts, regardless of the number of runs. Since compiler optimizations and code changes also alter layout, it is currently impossible to distinguish the impact of an optimization from that of its layout effects.

This paper presents STABILIZER, a system that enables the use of the powerful statistical techniques required for sound performance evaluation on modern architectures. STABILIZER forces executions to sample the space of memory configurations by repeatedly re-randomizing layouts of code, stack, and heap objects at runtime. STABILIZER thus makes it possible to control for layout effects. Re-randomization also ensures that layout effects follow a Gaussian distribution, enabling the use of statistical tests like ANOVA. We demonstrate STABILIZER's efficiency ($< 7\%$ median overhead) and its effectiveness by evaluating the impact of LLVM's optimizations on the SPEC CPU2006 benchmark suite. We find that, while -02 has a significant impact relative to -01 , the performance impact of -03 over -02 optimizations is indistinguishable from random noise.

Categories and Subject Descriptors C.4 [Performance of Systems]; D.2.0 [Software Engineering]: General; D.3.4 [Programming Languages]: Compilers

Keywords Randomization, Measurement Bias, Performance Evaluation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.
Copyright © 2013 ACM 978-1-4503-1870-9/13/03...\$15.00

1. Introduction

The task of performance evaluation forms a key part of both systems research and the software development process. Researchers working on systems ranging from compiler optimizations and runtime systems to code transformation frameworks and bug detectors must measure their effect, evaluating how much they improve performance or how much overhead they impose [7, 8]. Software developers need to ensure that new or modified code either in fact yields the desired performance improvement, or at least does not cause a performance regression (that is, making the system run slower). For large systems in both the open-source community (e.g., Firefox and Chromium) and in industry, automatic performance regression tests are now a standard part of the build or release process [25, 28].

In both settings, performance evaluation typically proceeds by testing the performance of the actual application in a set of scenarios, or a range of benchmarks, both before and after applying changes or in the absence and presence of a new optimization, runtime system, etc.

In addition to measuring *effect size* (here, the magnitude of change in performance), a statistically sound evaluation must test whether it is possible with a high degree of confidence to reject the *null hypothesis*: that the performance of the new version is indistinguishable from the old. To show that a performance optimization is statistically significant, we need to reject the null hypothesis with high confidence (and show that the direction of improvement is positive). Conversely, we aim to show that it is not possible to reject the null hypothesis when we are testing for a performance regression.

Unfortunately, even when using current best practices (large numbers of runs and a quiescent system), the conventional approach is unsound. The problem is due to the interaction between software and modern architectural features, especially caches and branch predictors. These features are sensitive to the addresses of the objects they manage. Because of the significant performance penalties imposed by cache misses or branch mispredictions (e.g., due to aliasing), their reliance on addresses makes software exquisitely sensitive to memory layout. Small changes to code, such as adding or removing a stack variable, or changing the order of heap allocations, can have a ripple effect that alters the placement of every other function, stack frame, and heap object.

The impact of these layout changes is unpredictable and substantial: Mytkowicz et al. show that just changing the size of environment variables can trigger performance degradation as high as 300% [22]; we find that simply changing the link order of object files can cause performance to decrease by as much as 57%.

Failure to control for layout is a form of *measurement bias*: a systematic error due to uncontrolled factors. All executions constitute just one sample from the vast space of possible memory layouts. This limited sampling makes statistical tests inapplicable, since they depend on multiple samples over a space, often with a

known distribution. As a result, it is currently not possible to test whether a code modification is the direct cause of any observed performance change, or if it is due to incidental effects like a different code, stack, or heap layout.

Contributions

This paper presents STABILIZER, a system that enables statistically sound performance analysis of software on modern architectures. To our knowledge, STABILIZER is the first system of its kind.

STABILIZER forces executions to sample over the space of all memory configurations by efficiently and repeatedly randomizing the placement of code, stack, and heap objects at runtime. We show analytically and empirically that STABILIZER’s use of randomization makes program execution independent of the execution environment, and thus eliminates this source of measurement bias. Re-randomization goes one step further: it causes the performance impact of layout effects to follow a Gaussian (normal) distribution, by virtue of the Central Limit Theorem. In many cases, layout effects dwarf all other sources of execution time variance [22]. As a result, STABILIZER often leads to execution times that are normally distributed.

By generating execution times with Gaussian distributions, STABILIZER enables statistically sound performance analysis via parametric statistical tests like ANOVA [11]. STABILIZER thus provides a push-button solution that allows developers and researchers to answer the question: does a given change to a program affect its performance, or is this effect indistinguishable from noise?

We demonstrate STABILIZER’s efficiency ($< 7\%$ median overhead) and its effectiveness by evaluating the impact of LLVM’s optimizations on the SPEC CPU2006 benchmark suite. Across the SPEC CPU2006 benchmark suite, we find that the `-O3` compiler switch (which includes argument promotion, dead global elimination, global common subexpression elimination, and scalar replacement of aggregates) does not yield statistically significant improvements over `-O2`. In other words, the effect of `-O3` versus `-O2` is indistinguishable from random noise.

We note in passing that STABILIZER’s low overhead means that it could be used at deployment time to reduce the risk of performance outliers, although we do not explore that use case here. Intuitively, STABILIZER makes it unlikely that object and code layouts will be especially “lucky” or “unlucky.” By periodically re-randomizing, STABILIZER limits the contribution of each layout to total execution time.

Outline

The remainder of this paper is organized as follows. Section 2 provides an overview of STABILIZER’s operation and statistical properties. Section 3 describes the implementation of STABILIZER’s compiler and runtime components, and Section 4 gives an analysis of STABILIZER’s statistical properties. Section 5 demonstrates STABILIZER’s avoidance of measurement bias, and Section 6 demonstrates the use of STABILIZER to evaluate the effectiveness of LLVM’s standard optimizations. Section 7 discusses related work. Finally, Section 8 presents planned future directions and Section 9 concludes.

2. STABILIZER Overview

This section provides an overview of STABILIZER’s operation, and how it provides properties that enable statistically rigorous performance evaluation.

2.1 Comprehensive Layout Randomization

STABILIZER dynamically randomizes program layout to ensure it is independent of changes to code, compilation, or execution environment. STABILIZER performs extensive randomization: it

dynamically randomizes the placement of a program’s functions, stack frames, and heap objects. Code is randomized at a per-function granularity, and each function executes on a randomly placed stack frame. STABILIZER also periodically *re-randomizes* the placement of functions and stack frames during execution.

2.2 Normally Distributed Execution Time

When a program is run with STABILIZER, the effect of memory layout on performance follows a normal distribution because of layout re-randomization. Layout effects make a substantial contribution to a program’s execution. In the absence of other large sources of measurement bias, STABILIZER causes programs to run with normally distributed execution times.

At a high level, STABILIZER’s re-randomization strategy induces normally distributed executions as follows: Each random layout contributes a small fraction of total execution time. Total execution time, the sum of runtimes with each random layout, is proportional to the mean of sampled layouts. The *Central Limit Theorem* states that “the mean of a sufficiently large number of independent random variables . . . will be approximately normally distributed” [11]. With a sufficient number of randomizations (30 is typical), and no other significant sources of measurement bias, execution time will follow a Gaussian distribution. Section 4 provides a more detailed analysis of STABILIZER’s effect on execution time distributions.

2.3 Sound Performance Analysis

Normally distributed execution times allow researchers to evaluate performance using *parametric* hypothesis tests, which provide greater *statistical power* by leveraging the properties of a known distribution (typically the normal distribution). Statistical power is the probability of correctly rejecting a false null hypothesis. Parametric tests typically have greater power than non-parametric tests, which make no assumptions about distribution. For our purposes, the null hypothesis is that a change had no impact. Failure to reject the null hypothesis suggests that more samples (benchmarks or runs) may be required to reach confidence, or that the change had no impact. Powerful parametric tests can correctly reject a false null hypothesis—that is, confirm that a change did have an impact—with fewer samples than non-parametric tests.

2.4 Evaluating Code Modifications

To test the effectiveness of any change (known in statistical parlance as a *treatment*), a researcher or developer runs a program with STABILIZER, both with and without the change. Each run is a sample from the treatment’s *population*: the theoretical distribution from which samples are drawn. Given that execution times are drawn from a normally distributed population, we can apply the Student’s t-test [11] to calculate the significance of the treatment.

The null hypothesis for the t-test is that the difference in means of the source distributions is zero. The t-test’s result (its *p-value*) tells us the probability of observing the measured difference between sample means, assuming both sets of samples come from the same source distribution. If the p-value is below a threshold α (typically 5%), the null hypothesis is rejected; that is, the two source distributions have different means. The parameter α is the probability of committing a type-I error: erroneously rejecting a true null hypothesis.

It is important to note that the t-test can detect *arbitrarily small* differences in the means of two populations (given a sufficient number of samples) regardless of the value of α . The difference in means does not need to be 5% to reach significance with $\alpha = 0.05$. Similarly, if STABILIZER adds 4.8% overhead to a program, this does not prevent the t-test from detecting differences in means that are smaller than 4.8%.

2.5 Evaluating Compiler and Runtime Optimizations

To evaluate a compiler or runtime system change, we instead use a more general technique: analysis of variance (ANOVA). ANOVA takes as input a set of results for each combination of benchmark and treatment, and partitions the total variance into components: the effect of random variations between runs, differences between benchmarks, and the collective impact of each treatment across all benchmarks [11]. ANOVA is a generalized form of the t-test that is less likely to commit type I errors (rejecting a true null hypothesis) than running many independent t-tests. Section 6 presents the use of STABILIZER and ANOVA to evaluate the effectiveness of compiler optimizations in LLVM.

Evaluating Layout Optimizations. All of STABILIZER’s randomizations (code, stack, and heap) can be enabled independently. This independence makes it possible to evaluate optimizations that target memory layout. For example, to test an optimization for stack layouts, STABILIZER can be run with only code and heap randomization enabled. These randomizations ensure that incidental changes, such as code to pad the stack or to allocate large objects on the heap, will not affect the layout of code or heap memory. The developer can then be confident that any observed change in performance is the result of the stack optimization and not its secondary effects on layout.

3. STABILIZER Implementation

STABILIZER uses a compiler transformation and runtime library to randomize program layout. STABILIZER performs its transformations in an optimization pass run by the LLVM compiler [17]. STABILIZER’s compiler transformation inserts the necessary operations to move the stack, redirects heap operations to the randomized heap, and modifies functions to be independently relocatable. STABILIZER’s runtime library exposes an API for the randomized heap, relocates functions on-demand, generates random padding for the stack, and re-randomizes both code and stack at regular intervals.

3.1 Building Programs with Stabilizer

When building a program with STABILIZER, each source file is first compiled to LLVM bytecode. STABILIZER builds Fortran programs with `gfortran` and the `dragonegg` GCC plugin, which generates LLVM bytecode from the GCC front-end [27]. C and C++ programs can be built either with `gcc` and `dragonegg`, or LLVM’s `clang` front-end [26].

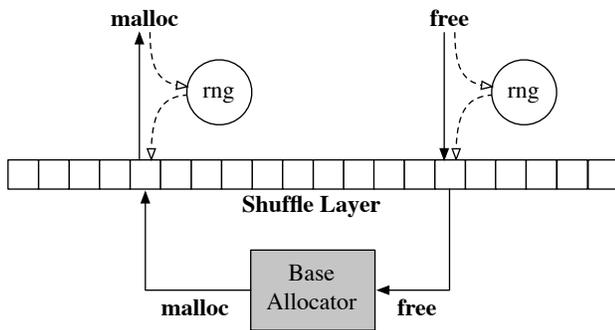


Figure 1. STABILIZER efficiently randomizes the heap by wrapping a deterministic base allocator in a shuffling layer. At startup, the layer is filled with objects from the base heap. The `malloc` function generates a random index, removes the indexed object from the shuffling layer, and replaces it with a new one from the base heap. Similarly, the `free` function generates a random index, frees the indexed object to the base heap, and places the newly freed object in its place.

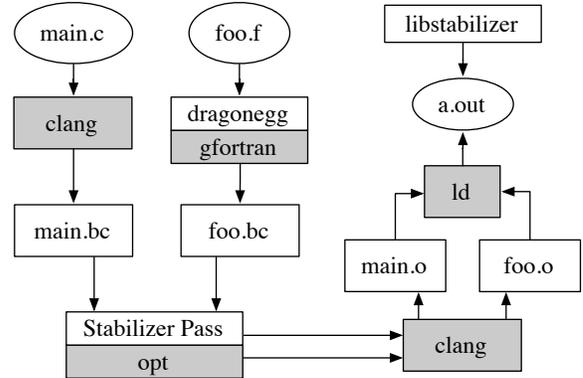


Figure 2. The procedure for building a program with STABILIZER. This process is automated by the `szc` compiler driver.

The compilation and transformation process is shown in Figure 2. This procedure is completely automated by STABILIZER’s compiler driver (`szc`), which is compatible with the common `clang` and `gcc` command-line options. Programs can easily be built and evaluated with STABILIZER by substituting `szc` for the default compiler/linker and enabling randomizations with additional flags.

3.2 Heap Randomization

STABILIZER uses a power of two, size-segregated allocator as the base for its heap [33]. Optionally, STABILIZER can be configured to use TLSF (two-level segregated fits) as its base allocator [19]. STABILIZER was originally implemented with the DieHard allocator [3, 24]. DieHard is a bitmap-based randomized allocator with power-of-two size classes. Unlike conventional allocators, DieHard does not use recently-freed memory for subsequent allocations. This lack of reuse and the added TLB pressure from the large virtual address space can lead to very high overhead.

While STABILIZER’s base allocators are more efficient than DieHard, they are not fully randomized. STABILIZER randomizes the heap by wrapping its base allocator in a shuffling layer built with `HeapLayers` [4]. The shuffling layer consists of a size N array of pointers for each size class. The array for each size class is initialized with a fill: N calls to `Base::malloc` are issued to fill the array, then the array is shuffled using the Fisher-Yates shuffle [10]. Every call to `Shuffle::malloc` allocates a new object p from `Base::malloc`, generates a random index i in the range $[0, N)$, swaps p with `array[i]`, and returns the swapped pointer. `Shuffle::free` works in much the same way: a random index i is generated, the freed pointer is swapped with `array[i]`, and the swapped pointer is passed to `Base::free`. The process for `malloc` and `free` is equivalent to one iteration of the inside-out Fisher-Yates shuffle. Figure 1 illustrates this procedure. STABILIZER uses the Marsaglia pseudo-random number generator from DieHard [3, 18].

The shuffled heap parameter N must be large enough to create sufficient randomization, but values that are too large will increase overhead with no added benefit. It is only necessary to randomize the index bits of heap object addresses. Randomness in lower-order bits will lead to misaligned allocations, and randomized higher order bits impose additional pressure on the TLB. NIST provides a standard statistical test suite for evaluation pseudorandom number generators [2]. We test the randomness of values returned by `libc’s lrand48` function, addresses returned by the DieHard allocator, and the shuffled heap for a range of values of N . Only the index bits (bits 6-17 on the Core2 architecture) were used. Bits used by branch predictors differ significantly across architectures, but are typically low-order bits generally in the same range as cache index bits.

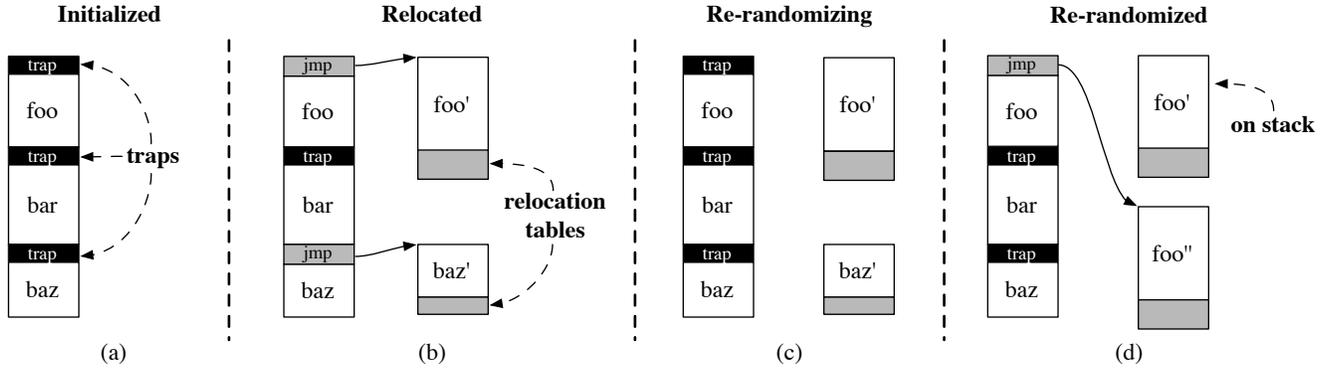


Figure 3. (a) During initialization, STABILIZER places a trap instruction at the beginning of each function. When a trapped function is called, it is relocated on demand. (b) Each randomized function has an adjacent relocation table, populated with pointers to all referenced globals and functions. (c) A timer triggers periodic re-randomizations (every 500ms by default). In the timer signal handler, STABILIZER places traps at the beginning of every randomized function. (d) Once a trapped function is called, STABILIZER walks the stack, marks all functions with return addresses on the stack, and frees the rest (`baz'` is freed in the example). Any remaining functions (`foo'`) will be freed after a future re-randomization once they are no longer on the stack. Future calls to `foo` will be directed to a new, randomly located version (`foo''`).

The `rand48` function passes six tests for randomness (Frequency, BlockFrequency, CumulativeSums, Runs, LongestRun, and FFT) with $> 95\%$ confidence, failing only the Rank test. DieHard passes these same six tests. STABILIZER’s randomized heap passes the same tests with the shuffling parameter $N = 256$. STABILIZER uses this heap configuration to randomly allocate memory for both heap objects and functions.

3.3 Code Randomization

STABILIZER randomizes code at function granularity. Every transformed function has a *relocation table* (see Figure 3(b)), which is placed immediately following the code for the function. Functions are placed randomly in memory using a separate randomized heap that allocates executable memory.

Relocation tables are not present in a binary built with STABILIZER. Instead, they are created at runtime immediately following each randomly located function. The sizes of functions are not available in the program’s symbol table, so the address of the next function is used to determine the function’s endpoint. A function refers to its adjacent relocation table with a PC-relative offset. This approach means that two randomly located copies of the same function do not share a relocation table.

Some constant floating point operands are converted to global variable references during code generation. STABILIZER converts all non-zero floating point constants to global variables in the IR so accesses can be made indirect through the relocation table.

Operations that convert between floating-point and integers do not contain constant operands, but still generate implicit global references during code generation. STABILIZER cannot rewrite these references. Instead, STABILIZER adds functions to each module to perform int-to-float and float-to-int conversions, and replaces the LLVM `fptosi`, `fptoui`, `sitofp`, and `uitofp` instructions with calls to these conversion functions. The conversion functions are the only code that STABILIZER cannot safely relocate.

Finally, STABILIZER renames the `main` function. The STABILIZER runtime library defines its own `main` function, which initializes runtime support for code randomization before executing any randomized code.

Initialization. At compile time, STABILIZER replaces the module’s `libc` constructors with its own constructor function. At startup, this constructor registers the module’s functions and any construc-

tors from the original program. Execution of the program’s constructors is delayed until after initialization.

The `main` function, defined in STABILIZER’s runtime, overwrites the beginning of every relocatable function with a software breakpoint (the `int 3 x86` instruction, or `0xCC` in hex); see Figure 3(a). A pointer to the function’s runtime object is placed immediately after the trap to allow for immediate relocation (not shown).

Relocation. When a trapped function is executed, the STABILIZER runtime receives a `SIGTRAP` signal and relocates the function (Figure 3(b)). Functions are relocated in three stages: first, STABILIZER requests a sufficiently large block of memory from the code heap and copies the function body to this location. Next, the function’s relocation table is constructed next to the new function location. STABILIZER overwrites the beginning of the function’s original base address with a static jump to the relocated function (replacing the trap instruction). Finally, STABILIZER adds the function to the set of “live” functions.

Re-randomization. STABILIZER re-randomizes functions at regular time intervals (500ms by default). When the re-randomization timer expires, the STABILIZER runtime places a trap instruction at the beginning of every live function and resumes execution (Figure 3(c)). Re-randomization occurs when the next trap is executed. This delay ensures that re-randomization will not be performed during the execution of non-reentrant code.

STABILIZER uses a simple garbage collector to reclaim memory used by randomized functions. First, STABILIZER adds the memory used by each live functions to a set called the “pile.” STABILIZER then walks the stack. Every object on the pile pointed to by a return address on the stack is marked. All unmarked objects on the pile are freed to the code heap.

3.4 Stack Randomization.

STABILIZER randomizes the stack by adding a random amount of space (up to 4096 bytes) between each stack frame. STABILIZER’s compiler pass creates a 256 byte stack pad table and a one-byte stack pad index for each function. On entry, the function loads the index-th byte, increments the index, and multiplies the byte by 16 (the required stack alignment on `x86_64`). STABILIZER moves the stack down by this amount prior to each function call, and restores the stack after the call returns.

The STABILIZER runtime fills every function’s stack pad table with random bytes during each re-randomization. The stack pad

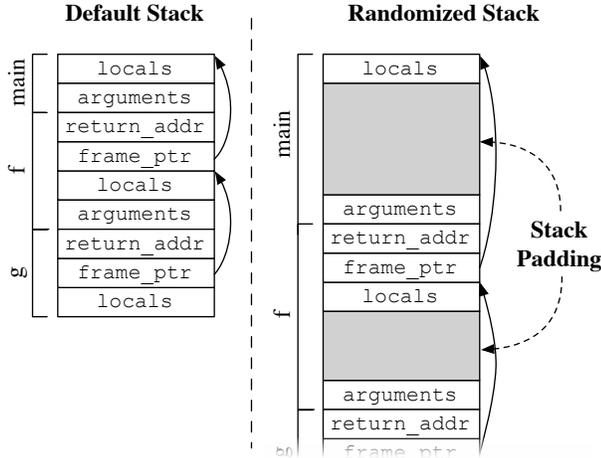


Figure 4. STABILIZER randomizes the stack by adding up to a page of random padding at each function call. Functions are instrumented to load a pad size from the stack pad table. STABILIZER periodically refills this table with new random values to re-randomize the stack.

index may overflow, wrapping back around to the first entry. This wraparound means functions may reuse a random stack pad several times between re-randomizations, but the precise location of the stack is determined by the stack pad size used for each function on the call stack. This combination ensures that stack placement is sufficiently randomized.

3.5 Architecture-Specific Implementation Details

STABILIZER runs on the x86, x86_64 and PowerPC architectures. Most implementation details are identical, but STABILIZER requires some platform-specific support.

x86_64

Supporting the x86_64 architecture introduces two complications for STABILIZER. The first is for the jump instructions: jumps, whether absolute or relative, can only be encoded with a 32-bit address (or offset). STABILIZER uses `mmap` with the `MAP_32BIT` flag to request memory for relocating functions, but on some systems (Mac OS X), this flag is unavailable.

To handle cases where functions must be relocated more than a 32-bit offset away from the original copy, STABILIZER simulates a 64-bit jump by pushing the target address onto the stack and issuing a return instruction. This form of jump is much slower than a 32-bit relative jump, so high-address memory is only used after low-address memory is exhausted.

PowerPC and x86

PowerPC and x86 both use PC-relative addressing for control flow, but data is accessed using absolute addresses. Because of this, the relocation table must be at a fixed absolute address rather than adjacent to a randomized function. The relocation table is only used for function calls, and does not need to be used for accesses to global data.

4. STABILIZER Statistical Analysis

This section presents an analysis that explains how STABILIZER’s randomization results in normally distributed execution times for most programs. Section 5 empirically verifies this analysis across our benchmark suite.

The analysis proceeds by first considering programs with a reasonably trivial structure (running in a single loop), and successively weakens this constraint to handle increasingly complex programs.

We assume that STABILIZER is only used on programs that consist of more than a single function. Because STABILIZER performs code layout randomization on a per-function basis, the location of code in a program consisting of just a single function will not be re-randomized. Since most programs consist of a large number of functions, we do not expect this to be a problem in practice.

Base case: a single loop. Consider a small program that runs repeatedly in a loop, executing at least one function. The space of all possible layouts l for this program is the population L . For each layout, an iteration of the loop will have an execution time t . The population of all iteration execution times is E . Clearly, running the program with layout l for 1000 iterations will take time:

$$T_{random} = 1000 * t$$

For simplicity, assume that when this same program is run with STABILIZER, each iteration is run with a different layout l_i with execution time t_i (we refine the notion of “iteration” below).

Running this program with STABILIZER for 1000 iterations will thus have total execution time:

$$T_{stabilized} = \sum_{i=1}^{1000} t_i$$

The values of t_i comprise a sample set x from the population E with mean:

$$\bar{x} = \frac{\sum_{i=1}^{1000} t_i}{1000}$$

The central limit theorem tells us that \bar{x} must be normally distributed (30 samples is typically sufficient for normality). The value of \bar{x} only differs from $T_{stabilized}$ by a constant factor. Multiplying a normally distributed random variable by a constant factor simply shifts and scales the distribution. The result remains normally distributed. Therefore, for this simple program, STABILIZER leads to normally distributed execution times. Note that the distribution of E was never mentioned—the central limit theorem guarantees normality regardless of the sampled population’s distribution.

The above argument relies on two conditions. The first is that STABILIZER runs each iteration with a different layout. STABILIZER actually uses wall clock time to trigger re-randomization, but the analysis still holds. As long as STABILIZER re-randomizes roughly every n iterations, we can simply redefine an “iteration” to be n passes over the same code. The second condition is that the program is simply a loop repeating the same code over and over again.

Programs with phase behavior. In reality, programs have more complex control flow and may even exhibit phase-like behavior. The net effect is that for one randomization period, where STABILIZER maintains the same random layout, one of any number of different portions of the application code could be running. However, the argument still holds.

A complex program can be recursively decomposed into subprograms, eventually consisting of subprograms equivalent to the trivial looping program described earlier. These subprograms will each comprise some fraction of the program’s total execution, and will all have normally distributed execution times. The total execution time of the program is thus a weighted sum of all the subprograms. A similar approach is used by SimPoint, which accelerates architecture simulation by drawing representative samples from all of a program’s phases [13].

Because the sum of two normally distributed random variables is also normally distributed, the program will still have a normally distributed execution time. This decomposition also covers the case

Distribution of Runtimes with STABILIZER's Repeated and One-Time Layout Randomization

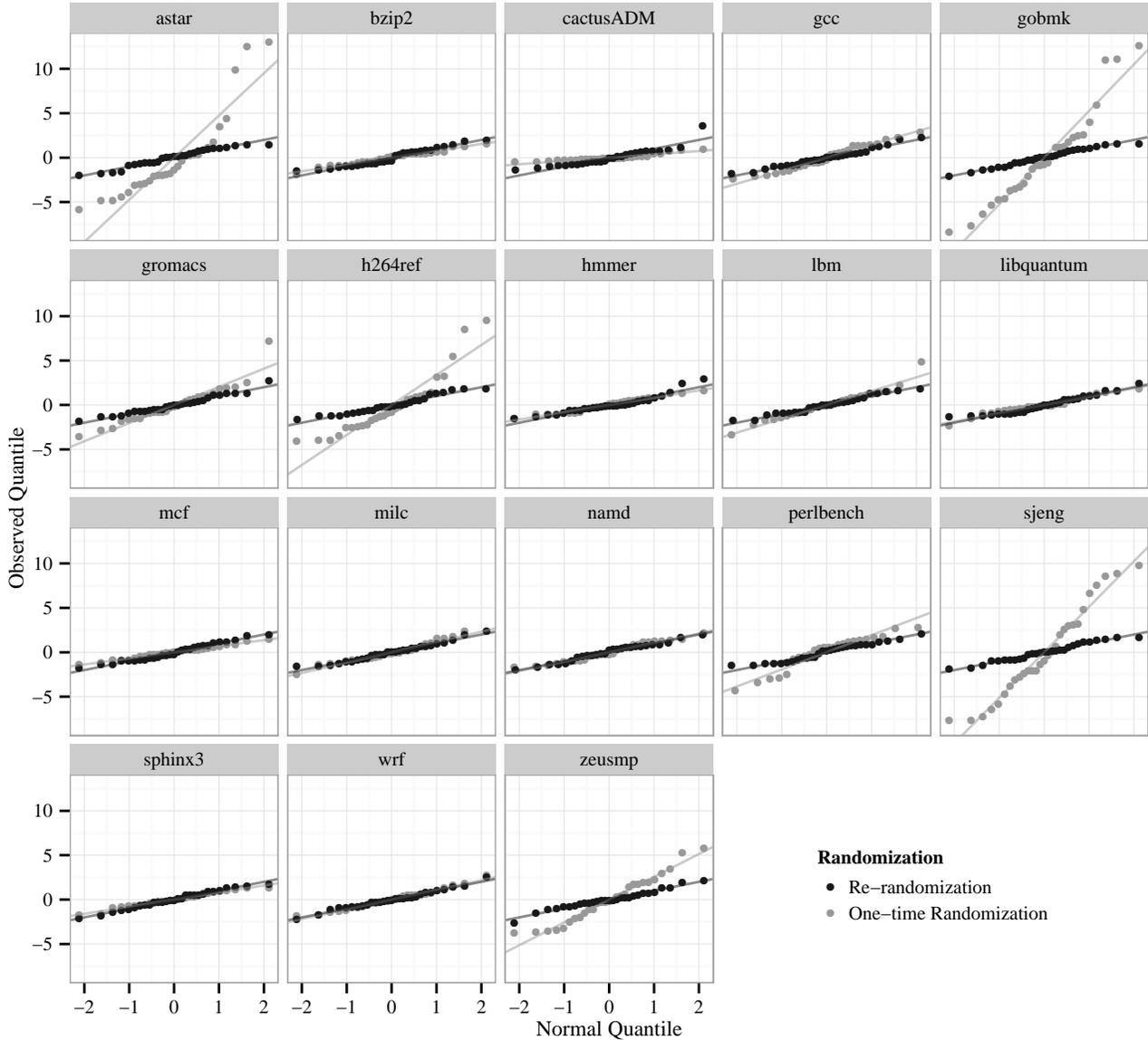


Figure 5. Gaussian distribution of execution time: Quantile-quantile plots comparing the distributions of execution times to the Gaussian distribution. Samples are shifted to a mean of zero, and normalized to the standard deviation of the re-randomized samples. Solid diagonal lines show where samples from a Gaussian distribution would fall. Without re-randomization, `astar`, `cactusADM`, `gromacs`, `h264ref`, and `perlbench` have execution times that are not drawn from a Gaussian distribution. With re-randomization, all benchmarks except `cactusADM` and `hmmer` conform to a Gaussian distribution. A steeper slope on the QQ plot indicates a greater variance. See Section 5.1 for a discussion of these results.

where STABILIZER's re-randomizations are out of phase with the iterations of the trivial looping program.

Heap accesses. Every allocation with STABILIZER returns a randomly selected heap address, but live objects are not relocated because C/C++ semantics do not allow it. STABILIZER thus enforces normality of heap access times as long as the program contains a sufficiently large number of short-lived heap objects (allowing them to be effectively re-randomized). This behavior is common for most applications and corresponds to the generational hypothesis for garbage collection, which has been shown to hold in unmanaged environments [9, 21].

STABILIZER cannot break apart large heap allocations, and cannot add randomization to custom allocators. Programs that use

custom allocators or operate mainly small objects from a single large array may not have normally distributed execution times because STABILIZER cannot sufficiently randomize their layout.

5. STABILIZER Evaluation

We evaluate STABILIZER in two dimensions. First, we test the claim that STABILIZER causes execution times to follow a Gaussian distribution. Next, we look at the overhead added by STABILIZER with different randomizations enabled.

All evaluations are performed on a dual-core Intel Core i3-550 operating at 3.2GHz and equipped with 12GB of RAM. Each core has a 256KB L2 cache. Cores share a single 4MB L3 cache. The system runs version 3.5.0-22 of the Linux kernel (unmodified) built

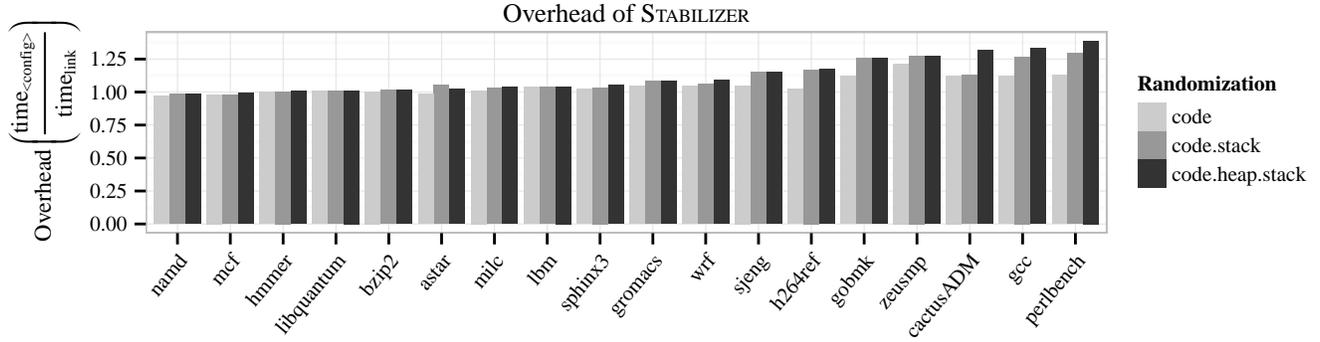


Figure 6. Overhead of STABILIZER relative to runs with randomized link order (lower is better). With all randomizations enabled, STABILIZER adds a median overhead of 6.7%, and below 40% for all benchmarks.

for x86_64. All programs are built using gcc version 4.6.3 as a front-end, with dragonegg and LLVM version 3.1.

Benchmarks. We evaluate STABILIZER across all C benchmarks in the SPEC CPU2006 benchmark suite. The C++ benchmarks `omnetpp`, `xalancbmk`, `dealII`, `soplex`, and `povray` are not run because they use exceptions, which STABILIZER does not yet support. We plan to add support for exceptions by rewriting LLVM’s exception handling intrinsics to invoke STABILIZER-specific runtime support for exceptions. STABILIZER is also evaluated on all Fortran benchmarks, except for `bwaves`, `calculix`, `games`, `GemsFDTD`, and `tonto`. These benchmarks fail to build on our system when using `gfortran` with the LLVM plugin.

5.1 Normality

We evaluate the claim that STABILIZER results in normally distributed execution times across the entire benchmark suite. Using the Shapiro-Wilk test for normality, we can check if the execution times of each benchmark are normally distributed with and without STABILIZER. Every benchmark is run 30 times each with and without STABILIZER’s re-randomization enabled.

Table 1 shows the p-values for the Shapiro-Wilk test of normality. Without re-randomization, five benchmarks exhibit execution times that are not normally distributed with 95% confidence: `astar`, `cactusADM`, `gromacs`, `h264ref`, and `perlbench`. With re-randomization, all of these benchmarks exhibit normally distributed execution times except for `cactusADM`. The `hmmer` benchmark has normally distributed execution times with one-time randomization, but not with re-randomization. This anomaly may be due to `hmmer`’s use of alignment-sensitive floating point operations.

Figure 5 shows the distributions of all 18 benchmarks on QQ (quantile-quantile) plots. QQ plots are useful for visualizing how close a set of samples is to a reference distribution (Gaussian in this case). Each data point is placed at the intersection of the sample and reference distributions’ quantiles. Points will fall along a straight line if the observed values come from the reference distribution family.

A steeper slope on the QQ plot indicates a greater variance. We test for homogeneity of variance using the Brown-Forsythe test [11]. For eight benchmarks, `astar`, `gcc`, `gobmk`, `gromacs`, `h264ref`, `perlbench`, `sjeng`, and `zeusmp`, re-randomization leads to a statistically significant decrease in variance. This decrease is the result of *regression to the mean*. Observing a very high execution time with re-randomization would require selecting many more “unlucky” than “lucky” layouts. In two cases, `cactusADM` and `mcf`, re-randomization yields a small but statistically significant increase in variance. The p-values for the Brown-Forsythe test are shown in Table 1.

Result: STABILIZER nearly always imposes a Gaussian distribution on execution time, and tends to reduce variance.

5.2 Efficiency

Figure 6 shows the overhead of STABILIZER relative to unrandomized execution. Every benchmark is run 30 times in each configuration. With all randomizations enabled, STABILIZER adds a median overhead of 6.7%.

Most of STABILIZER’s overhead can be attributed to reduced locality. Code and stack randomization both add additional logic to function invocation, but this has limited impact on execution time. Programs run with STABILIZER use a larger portion of the virtual address space, putting additional pressure on the TLB.

With all randomizations enabled, STABILIZER adds more than 30% overhead for just four benchmarks. For `gobmk`, `gcc`, and `perlbench`, the majority of STABILIZER’s overhead comes from stack randomization. These three benchmarks all have a large number of functions, each with its own stack pad table (described in Section 3).

Benchmark	Shapiro-Wilk		Brown-Forsythe
	Randomized	Re-randomized	
astar	0.000	0.194	0.001
bzip2	0.789	0.143	0.078
cactusADM	0.003	0.003	0.001
gcc	0.420	0.717	0.013
gobmk	0.072	0.563	0.000
gromacs	0.015	0.550	0.022
h264ref	0.003	0.183	0.002
hmmer	0.552	0.016	0.982
lbm	0.240	0.530	0.161
libquantum	0.437	0.115	0.397
mcf	0.991	0.598	0.027
milc	0.367	0.578	0.554
namd	0.254	0.691	0.610
perlbench	0.036	0.188	0.047
sjeng	0.240	0.373	0.000
sphinx3	0.727	0.842	0.203
wrf	0.856	0.935	0.554
zeusmp	0.342	0.815	0.000

Table 1. P-values for the Shapiro-Wilk test of normality and the Brown-Forsythe test for homogeneity of variance. A p-value less than $\alpha = 0.05$ is sufficient to reject the null hypothesis (indicated in bold). Shapiro-Wilk tests the null hypothesis that the data are drawn from a normal distribution. Brown-Forsythe tests whether the one-time randomization and re-randomization samples are drawn from distributions with the same variance. Boldface indicates statistically significant non-normal execution times and unequal variances, respectively. Section 5.1 explores these results further.

Impact of Optimizations

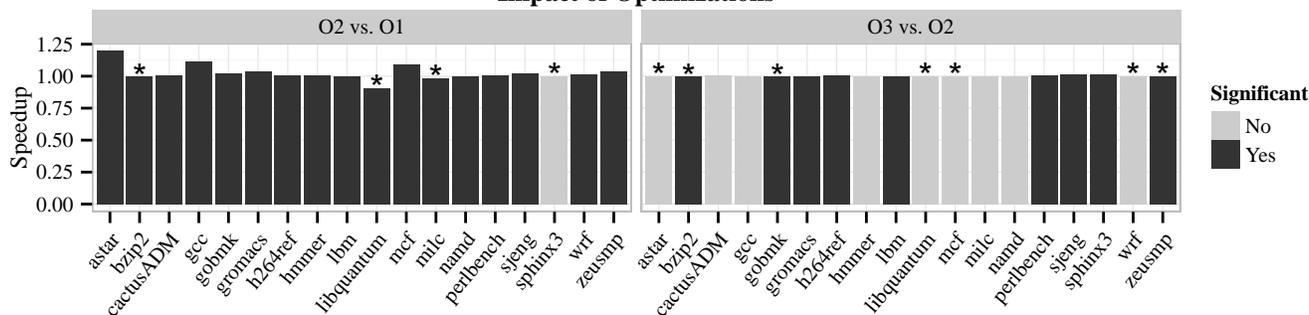


Figure 7. Speedup of $-O2$ over $-O1$, and $-O3$ over $-O2$ optimizations in LLVM. A speedup above 1.0 indicates the optimization had a positive effect. Asterisks mark cases where optimization led to slower performance. Benchmarks with dark bars showed a statistically significant average speedup (or slowdown). 17 of 18 benchmarks show a statistically significant change with $-O2$, and 9 of 18 show a significant change with $-O3$. In three cases for $-O2$ and three for $-O3$, the statistically significant change is a performance *degradation*. Despite per-benchmark significance results, the $-O3$ data do not show significance across the entire suite of benchmarks, and $-O2$ optimizations are only significant at a 90% level (Section 6.1).

The increased working set size increases cache pressure. If STABILIZER allowed functions to share stack pad tables, this overhead could be reduced. STABILIZER’s heap randomization adds most of the overhead to cactusADM. This benchmark allocates a large number of arrays on the heap, and rounding up to power of two size classes leads to a large amount of wasted heap space.

STABILIZER’s overhead does not affect its validity as a system for measuring the impact of performance optimizations. If an optimization has a statistically significant impact, it will shift the mean execution time over all possible layouts. The overhead added by STABILIZER also shifts this mean, but applies equally to both versions of the program. STABILIZER imposes a Gaussian distribution on execution times, which enables the detection of *smaller* effects than an evaluation of execution times with unknown distribution.

Performance Improvements

In four cases, STABILIZER (slightly) improves performance. `astar`, `hmmmer`, `mcf`, and `namd` all run faster with code randomization enabled. We attribute this to the elimination of branch aliasing [15]. It is highly unlikely that a significant fraction of a run’s random code layouts would exhibit branch aliasing problems. It is similarly unlikely that a significant fraction of random layouts would result in large performance improvements. The small gains with STABILIZER suggest the default program layout is slightly worse than the median layout for these benchmarks.

6. Sound Performance Analysis

The goal of STABILIZER is to enable statistically sound performance evaluation. We demonstrate STABILIZER’s use here by evaluating the effectiveness of LLVM’s $-O3$ and $-O2$ optimization levels. Figure 7 shows the speedup of $-O2$ and $-O3$, where speedup of $-O3$ is defined as:

$$\frac{\text{time}_{-O2}}{\text{time}_{-O3}}$$

LLVM’s $-O2$ optimizations include basic-block level common subexpression elimination, while $-O3$ adds argument promotion, global dead code elimination, increases the amount of inlining, and adds global (procedure-wide) common subexpression elimination.

Execution times for all but three benchmarks are normally distributed when run with STABILIZER. These three benchmarks, `hmmmer`, `wrf`, and `zeusmp`, have p-values below $\alpha = 0.05$ for the Shapiro-Wilk test. For all benchmarks with normally distributed execution times, we apply the two-sample t-test to determine whether

$-O3$ provides a statistically significant performance improvement over $-O2$, and likewise for $-O2$ over $-O1$. The three non-normal benchmarks use the Wilcoxon signed-rank test, a non-parametric equivalent to the t-test [32].

At a 95% confidence level, we find that there is a statistically significant difference between $-O2$ and $-O1$ for 17 of 18 benchmarks. There is a significant difference between $-O3$ and $-O2$ for 9 of 18 benchmarks. While this result is promising, it does come with a caveat: `bzip2`, `libquantum`, and `milc` show a statistically significant *increase* in execution time with $-O2$ optimizations. The `bzip2`, `gobmk`, and `zeusmp` benchmarks show a statistically significant performance degradation with $-O3$.

6.1 Analysis of Variance

Evaluating optimizations with pairwise t-tests is error prone. This methodology runs a high risk of erroneously rejecting the null hypothesis (a type-I error). The parameter $\alpha = 0.05$ is the probability of observing the measured speedup, given that the optimization actually has no effect. Figure 7 shows the results for 36 hypothesis tests, each with a 5% risk of a false positive. We expect $36 * 0.05 = 1.8$ of these tests to show that an optimization had a statistically significant impact when in reality it did not.

Analysis of variance (ANOVA) allows us to test the significance of each optimization level over all benchmarks simultaneously. ANOVA relies on a normal assumption, but has been shown to be robust to modest deviations from normality [11]. We run ANOVA with the same 18 benchmarks to test the significance of $-O2$ over $-O1$ and $-O3$ over $-O2$.

ANOVA takes the total variance in execution times and breaks it down by source: the fraction due to differences between benchmarks, the impact of optimizations, interactions between the independent factors, and random variation between runs. Differences between benchmarks should not be included in the final result. We perform a one-way analysis of variance within subjects to ensure execution times are only compared between runs of the same benchmark.

For the speedup of $-O2$, the results show an F-value of 3.235 for one degree of freedom (the choice between $-O1$ and $-O2$). The F-value is drawn from the F distribution [11]. The cumulative probability of observing any value drawn from $F(1) > 3.235 = 0.0898$ is the p-value for this test. The results show that $-O2$ optimizations are significant at a 90% confidence level, but not at the 95% level. The F-value for $-O3$ is 1.335, again for one degree of freedom. This gives a p-value of 0.264. We fail to reject the null hypothesis and must conclude that compared to $-O2$, $-O3$ optimizations are not statistically significant.

System	Base Randomization			Fine-Grain Randomization			Implementation		
	code	stack	heap	code	stack	heap	recompilation	dynamic	re-randomization
Address Space Layout Randomization [20, 29]		✓	✓					✓	
Transparent Runtime Randomization [35]		✓	✓					✓	
Address Space Layout Permutation [16]	✓	✓	✓	✓				✓	
Address Obfuscation [5]	✓	✓	✓	✓	✓		✓	✓*	
Dynamic Offset Randomization [34]	✓			✓*			✓	✓	
Bhatkar <i>et al.</i> [6]	✓	✓	✓	✓	✓		✓	✓	
DieHard [3]			✓			✓		✓	✓
STABILIZER	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 2. Prior work in layout randomization includes varying degrees of support for the randomizations implemented in STABILIZER. The features supported by each project are marked by a checkmark. Asterisks indicate limited support for the corresponding randomization.

7. Related Work

Randomization for Security. Nearly all prior work in layout randomization has focused on security concerns. Randomizing the addresses of program elements makes it difficult for attackers to reliably trigger exploits. Table 2 gives an overview of prior work in program layout randomization.

The earliest implementations of layout randomization, Address Space Layout Randomization (ASLR) and PaX, relocate the heap, stack, and shared libraries in their entirety [20, 29]. Building on this work, Transparent Runtime Randomization (TRR) and Address Space Layout permutation (ASLP) have added support for randomization of code or code elements (like the global offset table) [16, 35]. Unlike STABILIZER, these systems relocate entire program segments.

Fine-grained randomization has been implemented in a limited form in the Address Obfuscation and Dynamic Offset Randomization projects, and by Bhatkar, Sekar, and DuVarney [5, 6, 34]. These systems combine coarse-grained randomization at load time with finer granularity randomizations in some sections. These systems do not re-randomize programs during execution, and do not apply fine-grained randomization to every program segment. STABILIZER randomizes code and data at a fine granularity, and re-randomizes during execution.

Heap Randomization. DieHard uses heap randomization to prevent memory errors [3]. Placing heap objects randomly makes it unlikely that use after free and out of bounds accesses will corrupt live heap data. DieHarder builds on this to provide probabilistic security guarantees [23]. STABILIZER can be configured to use DieHard as its substrate, although this can lead to substantial overhead.

Predictable Performance. Quicksort is a classic example of using randomization for predictable performance [14]. Random pivot selection drastically reduces the likelihood of encountering a worst-case input, and converts a $O(n^2)$ algorithm into one that runs with $O(n \log n)$ in practice.

Randomization has also been applied to probabilistically analyzable real-time systems. Quiñones *et al.* show that random cache replacement enables probabilistic worst-case execution time analysis, while maintaining good performance. This probabilistic analysis is a significant improvement over conventional hard real-time systems, where analysis of cache behavior relies on complete information.

Performance Evaluation. Mytkowicz *et al.* observe that environmental sensitivities can degrade program performance by as much as 300% [22]. While Mytkowicz *et al.* show that layout can dramatically impact performance, their proposed solution, *experimental setup randomization* (the exploration of the space of different link orders and environment variable sizes), is substantially different.

Experimental setup randomization requires far more runs than STABILIZER, and cannot eliminate bias as effectively. For example, varying link orders only changes inter-module function placement, so that a change of a function’s size still affects the placement of all

functions after it. STABILIZER instead randomizes the placement of every function independently. Similarly, varying environment size changes the base of the process stack, but not the distance between stack frames.

In addition, any unrandomized factor in experimental setup randomization, such as a different shared library version, could have a dramatic effect on layout. STABILIZER does not require *a priori* identification of all factors. Its use of dynamic re-randomization also leads to normally distributed execution times, enabling the use of parametric hypothesis tests.

Alameldeen and Wood find similar sensitivities in processor simulators, which they also address with the addition of non-determinism [1]. Tsafirir, Ouaknine, and Feitelson report dramatic environmental sensitivities in job scheduling, which they address with a technique they call “input shaking” [30, 31]. Georges *et al.* propose rigorous techniques for Java performance evaluation [12]. While prior techniques for performance evaluation require many runs over a wide range of (possibly unknown) environmental factors, STABILIZER enables efficient and statistically sound performance evaluation by breaking the dependence between experimental setup and program layout.

8. Future Work

We plan to extend STABILIZER to randomize code at finer granularity. Instead of relocating functions, STABILIZER could relocate individual basic blocks at runtime. This finer granularity would allow for branch-sense randomization. Randomly relocated basic blocks can appear in any order, and STABILIZER could randomly swap the fall-through and target blocks during execution. This approach would effectively randomize the history portion of the branch predictor table, eliminating another potential source of bias.

STABILIZER is useful for performance evaluation, but its ability to dynamically change layout could also be used to improve program performance. Searching for optimal layouts *a priori* would be intractable: the number of possible permutations of all functions grows at the rate of $O(N!)$, without accounting for space between functions. However, sampling with performance counters could be used to detect layout-related performance problems like cache misses and branch mispredictions. When STABILIZER detects these problems, it could trigger a complete or partial re-randomization of layout to try to eliminate the source of the performance issue.

9. Conclusion

Researchers and software developers require effective performance evaluation to guide work in compiler optimizations, runtime libraries, and large applications. Automatic performance regression tests are now commonplace. Standard practice measures execution times before and after applying changes, but modern processor architectures make this approach unsound. Small changes to a program or its execution environment can perturb its layout, which affects

caches and branch predictors. Two versions of a program, regardless of the number of runs, are only two samples from the distribution over possible layouts. Statistical techniques for comparing distributions require more samples, but randomizing layout over many runs may be prohibitively slow.

This paper presents STABILIZER, a system that enables the use of the powerful statistical techniques required for sound performance evaluation on modern architectures. STABILIZER forces executions to sample the space of memory configurations by efficiently and repeatedly randomizing the placement of code, stack, and heap objects at runtime. Every run with STABILIZER consists of many independent and identically distributed (i.i.d.) intervals of random layout. Total execution time (the sum over these intervals) follows a Gaussian distribution by virtue of the Central Limit Theorem. STABILIZER thus enables the use of parametric statistical tests like ANOVA. We demonstrate STABILIZER's efficiency (< 7% median overhead) and its effectiveness by evaluating the impact of LLVM's optimizations on the SPEC CPU2006 benchmark suite. We find that the performance impact of -O3 over -O2 optimizations is indistinguishable from random noise.

We encourage researchers to download STABILIZER to use it as a basis for sound performance evaluation: it is available at <http://www.stabilizer-tool.org>.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1012195-CCF and the PROARTIS FP7 Project (European Union Grant No. 249100). The authors gratefully acknowledge Peter F. Sweeney, David Jensen, Daniel A. Jiménez, Todd Mytkowicz, Eduardo Quiñones, Leonidas Kosmidis, Jaime Abella, and Francisco J. Cazorla for their guidance and comments. We also thank the anonymous reviewers for their helpful comments.

References

- [1] A. Alameldeen and D. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *HPCA '03*, pp. 7–18. IEEE Computer Society, 2003.
- [2] L. E. Bassham, III, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks, N. A. Heckert, J. F. Dray, and S. Vo. SP 800-22 Rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. Tech. rep., National Institute of Standards & Technology, Gaithersburg, MD, United States, 2010.
- [3] E. D. Berger and B. G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *PLDI '06*, pp. 158–168. ACM, 2006.
- [4] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing High-Performance Memory Allocators. In *PLDI '01*, pp. 114–124. ACM, 2001.
- [5] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *USENIX Security '03*, pp. 8–8. USENIX Association, 2003.
- [6] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *SSYM '05*, pp. 271–286. USENIX Association, 2005.
- [7] S. M. Blackburn, A. Diwan, M. Hauswirth, A. M. Memon, and P. F. Sweeney. Workshop on Experimental Evaluation of Software and Systems in Computer Science (Evaluate 2010). In *SPLASH '10*, pp. 291–292. ACM, 2010.
- [8] S. M. Blackburn, A. Diwan, M. Hauswirth, P. F. Sweeney, et al. TR 1: Can You Trust Your Experimental Results? Tech. rep., Evaluate Collaboratory, 2012.
- [9] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining Generational and Conservative Garbage Collection: Framework and Implementations. In *POPL '90*, pp. 261–269. ACM, 1990.
- [10] R. Durstenfeld. Algorithm 235: Random Permutation. *Communications of the ACM*, 7(7):420, 1964.
- [11] W. Feller. *An Introduction to Probability Theory and Applications*, volume 1. John Wiley & Sons Publishers, 3rd edition, 1968.
- [12] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *OOPSLA '07*, pp. 57–76. ACM, 2007.
- [13] G. Hamerly, E. Perelman, J. Lau, B. Calder, and T. Sherwood. Using Machine Learning to Guide Architecture Simulation. *Journal of Machine Learning Research*, 7:343–378, Dec. 2006.
- [14] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.
- [15] D. A. Jiménez. Code Placement for Improving Dynamic Branch Prediction Accuracy. In *PLDI '05*, pp. 107–116. ACM, 2005.
- [16] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *ACSAC '06*, pp. 339–348. IEEE Computer Society, 2006.
- [17] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04*, pp. 75–86. IEEE Computer Society, 2004.
- [18] G. Marsaglia. Random Number Generation. In *Encyclopedia of Computer Science, 4th Edition*, pp. 1499–1503. John Wiley and Sons Ltd., Chichester, UK, 2003.
- [19] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A New Dynamic Memory Allocator for Real-Time Systems. In *ECRTS '04*, pp. 79–86. IEEE Computer Society, 2004.
- [20] I. Molnar. Exec-Shield. <http://people.redhat.com/mingo/exec-shield/>.
- [21] D. A. Moon. Garbage Collection in a Large LISP System. In *LFP '84*, pp. 235–246. ACM, 1984.
- [22] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! In *ASPLOS '09*, pp. 265–276. ACM, 2009.
- [23] G. Novark and E. D. Berger. DieHarder: Securing the Heap. In *CCS '10*, pp. 573–584. ACM, 2010.
- [24] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically Correcting Memory Errors with High Probability. *Communications of the ACM*, 51(12):87–95, 2008.
- [25] The Chromium Project. Performance Dashboard. <http://build.chromium.org/f/chromium/perf/dashboard/overview.html>.
- [26] The LLVM Team. Clang: a C Language Family Frontend for LLVM. <http://clang.llvm.org>, 2012.
- [27] The LLVM Team. Dragonegg - Using LLVM as a GCC Backend. <http://dragonegg.llvm.org>, 2013.
- [28] The Mozilla Foundation. Buildbot/Talos. <https://wiki.mozilla.org/Buildbot/Talos>.
- [29] The PaX Team. The PaX Project. <http://pax.grsecurity.net>, 2001.
- [30] D. Tsafir and D. Feitelson. Instability in Parallel Job Scheduling Simulation: the Role of Workload Flurries. In *IPDPS '06*. IEEE Computer Society, 2006.
- [31] D. Tsafir, K. Ouaknine, and D. G. Feitelson. Reducing Performance Evaluation Sensitivity and Variability by Input Shaking. In *MASCOTS '07*, pp. 231–237. IEEE Computer Society, 2007.
- [32] F. Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80–83, 1945.
- [33] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. *Lecture Notes in Computer Science*, 986, 1995.
- [34] H. Xu and S. J. Chapin. Improving Address Space Randomization with a Dynamic Offset Randomization Technique. In *SAC '06*, pp. 384–391. ACM, 2006.
- [35] J. Xu, Z. Kalbarczyk, and R. Iyer. Transparent Runtime Randomization for Security. In *SRDS '03*, pp. 260–269. IEEE Computer Society, 2003.