STABILIZER: Statistically Sound Performance Evaluation

Charlie Curtsinger Emery D. Berger

Dept. 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. Every binary constitutes one sample regardless of the number of runs, making statistical tests inapplicable. It is thus currently impossible to test whether a performance change is due to a code modification or coincidental because the memory layout changed.

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. This randomization ensures that execution times follow a Gaussian distribution by virtue of the Central Limit Theorem, enabling the use of statistical tests like ANOVA. We demonstrate STABILIZERs efficiency ($\approx 5\%$ 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.

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. 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 [10, 22].

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. A statistically sound evaluation would test whether it is possible with a high degree of confidence to reject (or not reject) the *null hypothesis* that the new results are no different than the originals. 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). To show that we have not caused a performance regression, we want to show that it is not possible to reject this null hypothesis (i.e., that new or modified code had no statistically significant impact on performance).

Unfortunately, even when using current best practices (large numbers of runs and a quiescent system), this 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, adding or removing a stack variable, or changing the order of heap allocations can have

a ripple effect that alters the placement in memory of every other function, stack frame, and heap object.

The effect of these 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% [19]; we find that simply changing the link order of object files can cause performance to decrease by up to 57%.

Every execution thus actually constitutes a single sample of the (vast) space of possible memory layouts. This is a form of *measurement bias* that makes statistical tests inapplicable, since they depend on multiple samples over a space with a known distribution. The result is that it is currently not possible to test whether a code modification is the direct cause of any observed performance change, or is simply coincidence because of 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) repeatedly re-randomizing layouts 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 measurement bias. Re-randomization goes one step further: it leads to execution times with a Gaussian (normal) distribution, by virtue of the Central Limit Theorem.

By generating execution times with Gaussian distributions, STABILIZER enables statistically sound performance analysis via statistical tests like ANOVA [9]. 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 ($\approx 5\%$ 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 further reduces these odds.

Outline

The remainder of this paper is organized as follows. Section 2 provides an overview of STABILIZER's operation and statistical guarantees. Section 3 discusses related work. Section 4 describes the implementation of STABILIZER's compiler and runtime components, and Section 5 gives an analysis of STABILIZER's statistical guarantees. Section 6 demonstrates STABILIZER's avoidance of measurement bias, and Section 7 demonstrates the use of STABILIZER to evaluate the effectiveness of LLVM's standard optimizations. 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 leads to statistical properties that enable predictable and analyzable performance.

Environmental sensitivity both undermines predictability and reliable performance evaluation because of a lack of independence. Any change to a program's code, compilation, or execution environment can lead to a different memory layout. Prior work has shown that changing the size of the shell environment variables can degrade performance by as much as 300% [19]. The unpredictability and magnitude of this effect makes it impossible to evaluate changes to code or compilation in isolation.

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 function granularity, and each function executes on a randomly-placed stack frame. STABILIZER also periodically *re- randomizes* code and stack frames during execution.

2.2 Normally-Distributed Execution Time

STABILIZER's re-randomization of memory layouts not only avoids measurement bias, but also makes performance predictable and analyzable by inducing normally distributed execution times.

At a high level, STABILIZER's re-randomization strategy leads to normally-executed distributions as follows: Each random layout contributes to the total execution time. Total execution time is the sum over all random layouts, which is proportional to the mean over all these layouts. The *central limit theorem* states that "the mean of a sufficiently large number of independent random variables ... will be approximately normally distributed" [9]. As long as STABILIZER re-randomizes layout a sufficient number of times (30 is typical), and each layout is chosen independently, then execution

time will follow a Gaussian distribution. Section 5 provides a more detailed analysis.

2.3 Sound Performance Analysis

Normally distributed execution times allow researchers to evaluate performance using powerful *parametric* hypothesis tests, which rely on the assumption of normality. These tests are "powerful" in the sense that they more readily reject a false null hypotheses than more general (non-parametric) tests that 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, but only if the data follow a Gaussian distribution.

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. Given that execution times are normally distributed, we can apply the Student's t-test [9] to determine whether performance differs between the two treatments. The t-test, given a set of execution times, tells us the probability of observing the given samples if both treatments result in the same distribution. If this probability is below a threshold α (typically 5%, for 95% confidence), we say that the null hypothesis has been rejected—the two populations have distinct means. Our confidence tells us that there is at most a 5% chance we have committed a type-I error, meaning the population means are actually indistinguishable.

It is important to note that the Student's 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%. Execution times are samples from two Gaussian distributions (before and after a code change). The t-test is a probability query from the distribution of differences in means of two populations. If the probability of observing both positive and negative values is less than α, the null hypothesis is rejected.

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 [9]. ANOVA is a generalized form of the t-test that is less likely to commit type I error (rejecting a true null hypothesis) than running many independent t-tests. Section 7 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 makes it possible to evaluate optimizations that target memory layout. To test an optimization for stack layout, STABILIZER should be run with code and heap randomization enabled. This guarantees that incidental changes, such as code to pad the stack or allocate large objects on the heap, will not affect the layout of code or heap memory. The developer can be confident that any observed changed in performance is the result of the stack optimization and not its secondary effects on layout.

3. 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 1 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 [17, 23]. 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) [15, 27]. 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, 26]. 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 all 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 [20]. 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 [13]. Random pivot selection drastically reduces the likelihood

Base Randomization	ASLR	TRR	ASLP	Addr. Obfuscation	Dyn. Offset	B.S.DV [6]	DieHard	STABILIZER
code			✓	✓	✓	✓		✓
stack	✓	✓	✓	✓		✓		✓
heap	✓	✓	✓	✓		✓		✓
Full Randomization								
code			✓	✓	✓*	✓		✓
stack				✓*		✓*		✓
heap							✓	✓
Implementation								
recompilation				✓	✓	✓		✓
dynamic	✓	✓	✓	✓*	✓	✓	✓	✓
re-randomization							✓	✓

Table 1. 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.

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 a random cache replacement policy enables probabilistic worst-case execution time analysis, while still providing 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% [19]. 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 to the size of a function 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 relative addresses of stack slots. STABILIZER randomizes each stack frame independently.

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 statistically sound hypothesis testing.

Alameldeen and Wood find similar sensitivities in processor simulators, which they also address with the addition of non-determinism [1]. Tsafrir, Ouaknine, and Feitelson report dramatic environmental sensitivities in job scheduling, which they address with a technique they call "input shaking" [24, 25]. Georges et al. propose rigorous techniques for Java performance evaluation [11]. 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.

4. STABILIZER Implementation

STABILIZER dynamically randomizes the layout of heap objects, functions, and stack frames. Each randomization consists of a compiler transformation and runtime support. When building a program with STABILIZER, each source file is first compiled to LLVM bytecode. C and C++ programs use the `clang` front-end, Fortran programs use `gfortran` along with LLVM's GCC plugin to generate bytecode before optimization. Each bytecode file is optimized per the user's request, then transformed by STABILIZER. This ensures STABILIZER cannot affect any optimization decisions made by the compiler, and that no optimizations can interfere with STABILIZER's transformations. Finally, bytecode files are translated to machine code and linked by `clang`.

Compilation can be performed using STABILIZER's `szc` compiler driver, 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 and enabling randomizations with additional flags.

4.1 Heap Randomization

STABILIZER uses the TLSF (two-level segregated fits) allocator as the base for its randomized heap [16]. The TLSF

allocator is $O(1)$ for all operations, always chooses the smallest suitable size for allocations, and aggressively coalesces freed memory. STABILIZER was originally implemented with the DieHard allocator [3, 21]. 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 TLSF is more efficient than DieHard, it is not a randomized allocator. STABILIZER randomizes the heap by wrapping TLSF 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 `TLSF::malloc` are issued to fill the array, then the array is shuffled using the Fisher-Yates shuffle [8]. Every call to `Shuffle::malloc` allocates a new object `p` from `TLSF::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 `TLSF::free`. The process for `malloc` and `free` is equivalent to one iteration of the inside-out Fisher-Yates shuffle.

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 TLSF 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.

The `lrand48` 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. The shuffled TLSF heap passes the same tests with the parameter $N = 256$. STABILIZER uses this heap configuration to randomly allocate memory for heap objects, stack frames, and functions.

4.2 Code Randomization

STABILIZER randomizes code at the function granularity. Every transformed function has a *relocation table* (see Figure 1), which is placed immediately following the code for the function. The relocation table contains a `users` counter that tracks the number of stack frames with a reference to the relocated function. Functions are placed randomly in memory using an instance of the shuffled TLSF heap configured to map executable pages.

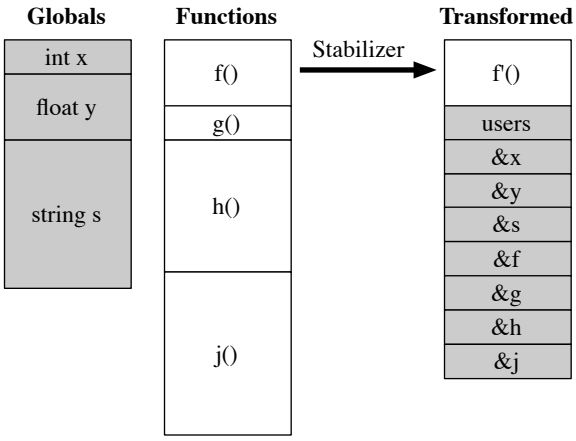


Figure 1. STABILIZER adds a relocation table to the end of each function, making every function independently relocatable. White boxes contain code and shaded boxes contain data.

Every function call or global access in the function is indirected through the relocation table. Relocation tables are not present in the program binary but are created on demand by the STABILIZER runtime.

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 end.

Functions refer to their relocation table with PC-relative addresses—Two randomly located copies of the same function do not share a relocation table. STABILIZER adds code to each function to increment its `users` counter on entry and decrement it on exit.

Initialization. During startup, STABILIZER overwrites the first byte of every relocatable function with a software breakpoint (the `int 3` x86 opcode, or `0xCC` in hex). When a function is called, STABILIZER intercepts the trap and relocates the function. Every random function location has a corresponding function location object, which is placed on the active locations list.

Relocation. 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 with the `users` counter set to `0`. Finally, STABILIZER overwrites the beginning of the function's original base address with a static jump to the relocated function.

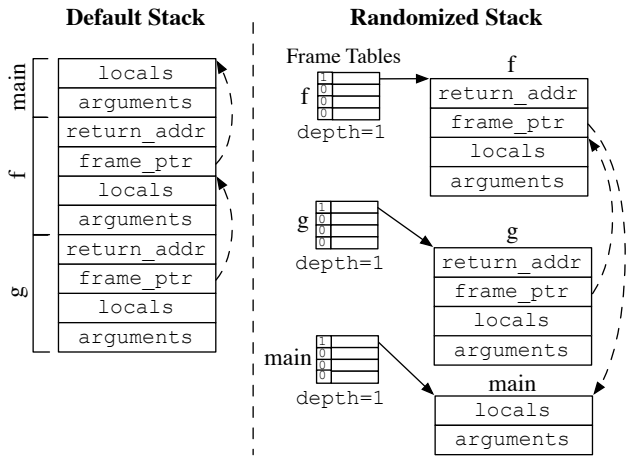


Figure 2. STABILIZER makes the stack non-contiguous. Each function has a frame table, which stores a frame for each recursion depth.

Re-randomization. STABILIZER re-randomizes functions at regular time intervals. When a timer signal is delivered, all running threads are interrupted. STABILIZER then processes every function location in the active locations list. The original base of the function is overwritten with a breakpoint instruction, and the function location is added to the defunct locations list. This list is scanned on every timer interrupt, and any locations with no remaining users are freed. The `users` counter will never increase for a defunct function location because future calls to the function will execute in a new location with its own `users` counter.

4.3 Stack Randomization

STABILIZER randomizes the stack by making it non-contiguous: each function call moves the stack to a random location. These randomly placed frames are also allocated using the shuffled TLSF allocator, and STABILIZER reuses them for some time before they are freed. This bounded reuse improves cache utilization and reduces the number of calls to the allocator while still enabling re-randomization.

Every function has a per-thread depth counter and frame table that maps the depth to the corresponding stack frame. The depth counter is incremented at the start of the function an decremented just before returning. On every call, the function loads its stack frame address from the frame address array (`frame_table[depth]`). If the frame address is `NULL`, the STABILIZER runtime allocates a new frame.

External functions. Special handling is required when a stack-randomized function calls an external function. Because external functions have not been randomized with STABILIZER, they must run on the default stack to prevent overrunning the randomly located frame. STABILIZER returns the stack pointer to the default stack location just before the call instruction, and returns it to the random frame after the call returns. Calls to functions processed by STABILIZER do not require special handling because these functions will always switch to their randomly allocated frames.

Re-randomization. At regular intervals, STABILIZER invalidates saved stack frames by setting a bit in each entry of the frame table. When a function loads its frame from the frame table, it checks this bit. If the bit is set, the old frame is freed and a new one is allocated and stored in the table.

4.4 Architecture-Specific Implementation Details

STABILIZER runs on the x86, x86_64 and PowerPC architectures. Most implementation details are identical, but STABILIZER requires 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

PowerPC instructions use a fixed-width encoding of four bytes. Jump instructions use 6 bits to encode the type of jump to perform, so jumps can only target sign-extended 26 bit addresses (or offsets, in the case of relative jump). This limitation results in a memory hole that cannot be reached by a single jump instruction. To ensure that code is never placed in this hole, STABILIZER uses the `MAP_FIXED` flag when initializing the code heap to ensure that all functions are placed in reachable memory.

4.5 Optimizations

STABILIZER performs a number of optimizations that reduce the overhead of randomization. The first addresses the cost of software breakpoints. Frequently-called functions incur the cost of a software breakpoint after every function relocation. Functions that have been relocated in 10 different randomization periods are marked as persistent. The STABILIZER runtime preemptively relocates persistent functions at startup time, instead of on-demand with a software breakpoint. STABILIZER occasionally selects a persistent function at random and resets it to on-demand relocation to ensure that only actively used functions are eagerly relocated.

The second optimization addresses inadvertent instruction cache invalidations. If relocated functions are allocated near randomly placed frames or heap objects, this could lead to

unnecessary instruction cache invalidations. To avoid this, functions are relocated using a separate randomized heap. For x86_64, this approach has the added benefit of preserving low-address memory, which is more efficient to reach by jumps. Function relocation tables pose a similar problem: every call updates the `users` counter, which could invalidate the cached copy of the relocated function. To prevent this, the relocation table is located at least one cache line away from the end of the function body.

5. STABILIZER Statistical Analysis

This section presents an analysis that explains how STABILIZER's randomization results in normally-distributed execution times for most programs. Section 6 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 e. The population of all iteration execution times is E. Clearly, running the program with layout l for 1000 iterations will take time:

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

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

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

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

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

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

The central limit theorem tells us that $\bar{x}$ must be normally distributed (30 samples is sufficient for normality). Interestingly, the value of $\bar{x}$ is only different 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 [12].

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 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++ 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 [7, 18].

STABILIZER cannot break apart large heap allocations, and cannot add randomization to custom allocators. Programs that use custom allocators or allocate small objects from a single large array may not have normally distributed execution times because STABILIZER cannot sufficiently randomize their layout.

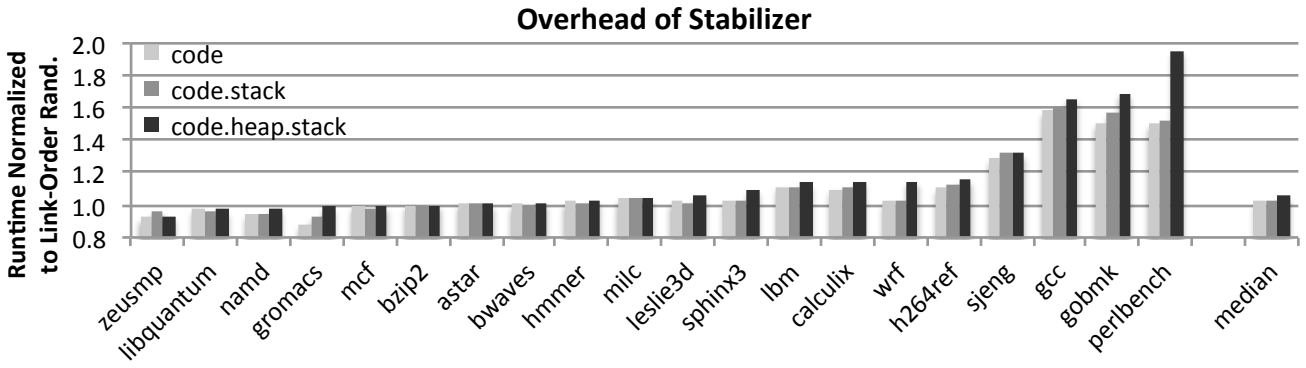


Figure 3. Overhead of STABILIZER relative to runs with randomized link order. With all randomizations enabled, STABILIZER adds a median overhead of 5.5%, below 20% overhead for 16 of 20 benchmarks, and in five cases slightly improves performance.

6. 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 were performed on a quad-socket 16-core AMD Opteron 6272 equipped with 64GB of RAM. Each core has a 16KB 4-way set associative L1 data cache, a 64KB 2-way set associative instruction cache, a 2MB 16-way set associative L2 cache, and a 6MB 64-way set-associative L3 cache shared by eight cores. The system runs version 2.6.32-5 of Linux kernel (unmodified) built for `x86_64`. All programs (with and without STABILIZER) were built using LLVM version 3.1. C and C++ benchmarks were built with the Clang frontend (also version 3.1). Fortran benchmarks were built with gfortran 4.6.3 using LLVM's GCC plugin to emit bytecode prior to optimization.

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 are not yet supported by STABILIZER. We plan add support for exceptions by rewriting LLVM's exception handling intrinsics to invoke STABILIZER-specific runtime support for exceptions. As an interim step, we plan to build these programs with exceptions disabled to gather preliminary results. STABILIZER is also evaluated on all Fortran benchmarks, except for `gamess`, `cactusADM`, `GemsFDTD`, and `tonto`. These benchmarks fail to build on our system when using gfortran with the LLVM plugin.

6.1 Performance Isolation

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 was run 20 times, with a different random link order for every run.

Without STABILIZER, 7 benchmarks exhibit execution times that are not normally distributed with 95% confidence: `astar`, `calculix`, `gromacs`, `leslie3d`, `milc`, `wrf`, and `zeusmp`. For all these benchmarks, except `wrf`, execution times with STABILIZER follow a Gaussian distribution.

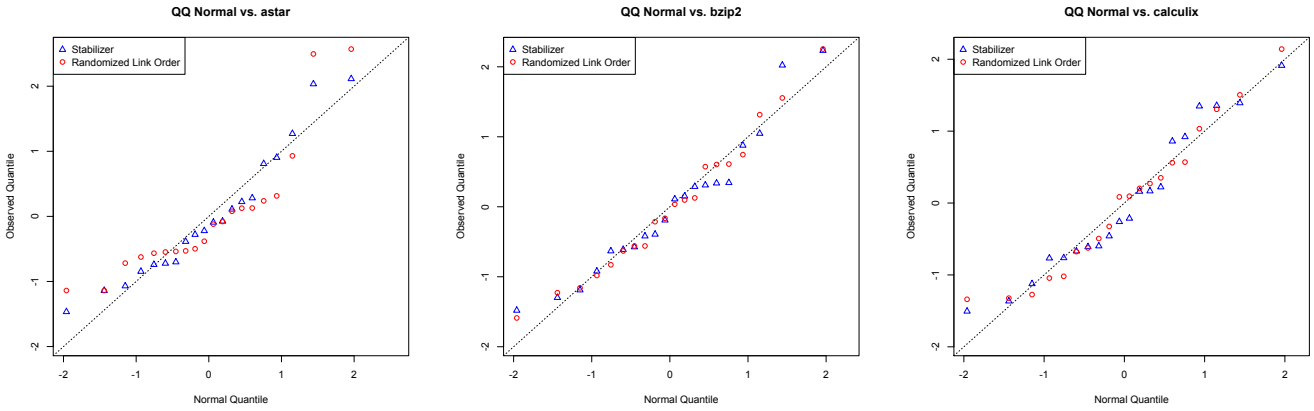
The `namd` benchmark has normally distributed execution time, but not with STABILIZER. `namd` uses floating point operations, which require 16-byte alignment for good performance. Unfortunately, we learned too late that the TLSF allocator guarantees only 8-byte alignment. STABILIZER is unable to impose a Gaussian distribution on execution times for `wrf`. This is because `wrf` has large stack frames which, like large heap objects, STABILIZER cannot break into smaller pieces to fully randomize.

Figure 4 shows the distributions of the eight benchmarks with non-Gaussian execution times using quantile-quantile (QQ) plots. QQ plots are useful for visualizing how close a set of samples is to a distribution (or another set of samples). These plots compare the observed quantiles for execution times to the Gaussian distribution. Each data point is placed at the intersection of the sample and reference distributions' quantiles. If the samples come from the same distribution family, the points will fall along the solid diagonal line shown on each plot. The figures for `astar`, `gromacs`, and `leslie3d` show distributions that differ significantly from Gaussian. Large jumps and changing slope in the QQ plot indicate the execution times have a bi-modal distribution.

Result: These figures demonstrate that STABILIZER nearly always imposes a Gaussian distribution on execution time. This holds even for programs with execution times that did not exhibit a Gaussian distribution without STABILIZER.

6.2 Efficiency

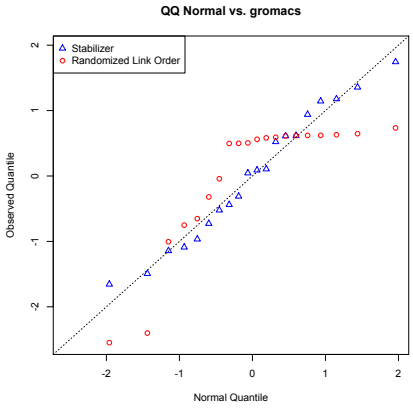
Figure 3 shows the overhead of STABILIZER relative to unrandomized execution. Each benchmark is run 20 times in each



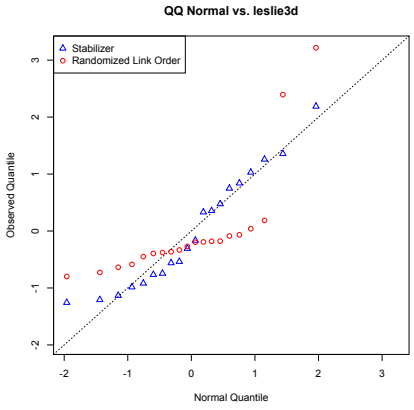
(a) astar

(b) bzip2

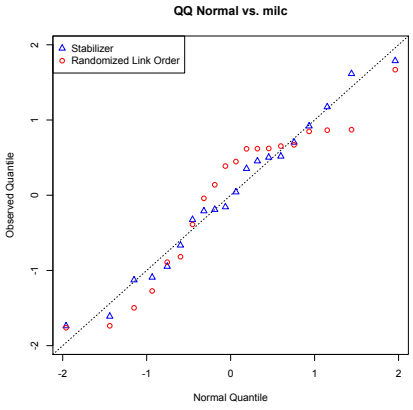
(c) calculix



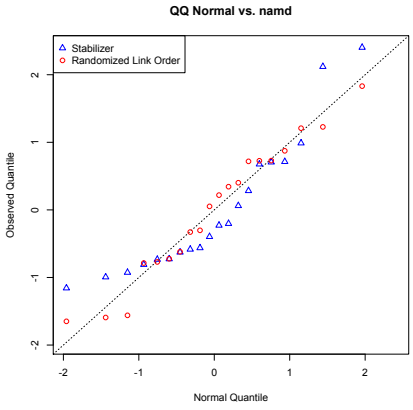
(d) gromacs



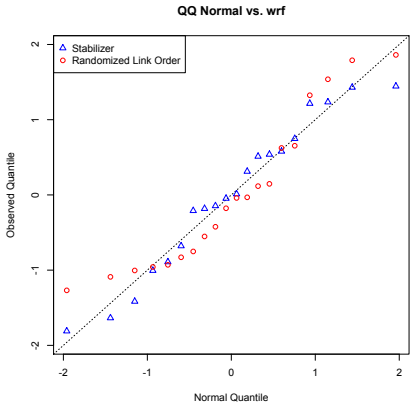
(e) leslie3d



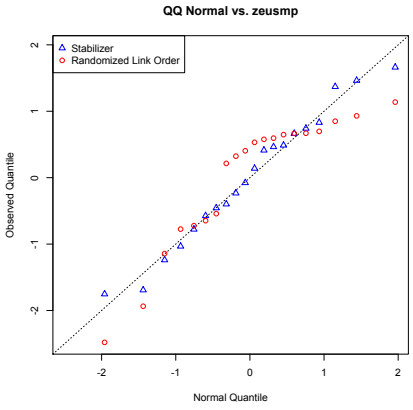
(f) milc



(g) namd



(h) wrf



(i) zeusmp

Figure 4. Gaussian distribution of execution time: Quantile-quantile plots comparing the distributions of execution times to the Gaussian distribution. The solid diagonal line shows where samples from a Gaussian distribution should fall. `astar`, `calculix`, `gromacs`, `leslie3d`, `milc`, and `wrf`, `zeusmp`, without STABILIZER, have execution times that are *not* drawn from a Gaussian distribution. All except `wrf` do conform to a Gaussian distribution when run with STABILIZER. `bzip2` has a Gaussian distribution with and without STABILIZER. `namd` has normally distributed execution times, but not with STABILIZER enabled. See Section 6.1 for a discussion of these results.

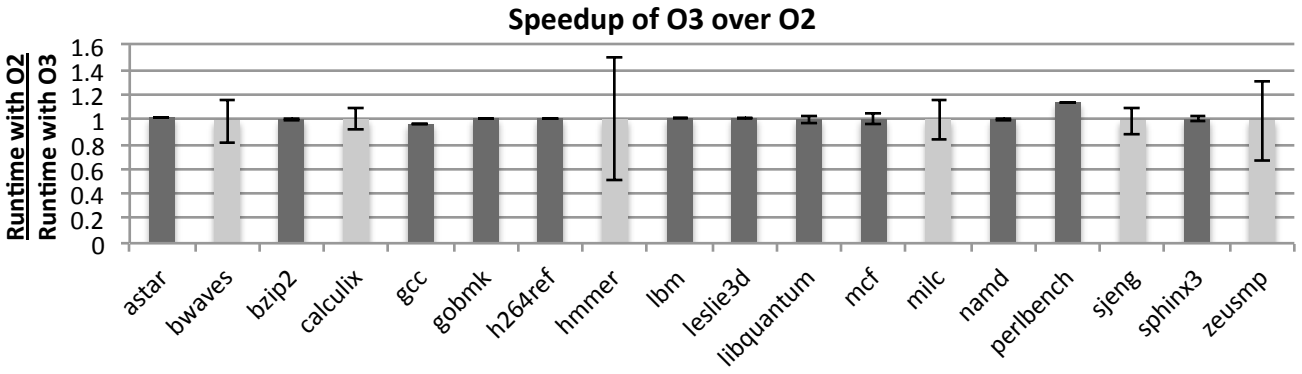


Figure 5. Speedup of `-O3` over the `-O2` optimization level in LLVM. Error bars indicate the p-values for the T-test comparing `-O2` and `-O3` (smaller is better). Benchmarks with dark bars showed a statistically significant change with `-O3` relative to `-O2`. While 12 of 18 benchmarks showed a statistically significant change, four of these cases were a performance *degradation*. Despite per-benchmark significance results, the data do not show significance across the entire suite of benchmarks (Section 7.1).

configuration. With all randomizations enabled, STABILIZER adds a median overhead of 5.5%.

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 significant overhead for four benchmarks: `sjeng`, `gcc`, `gobmk`, and `perlbench`. The high overhead for `perlbench` with heap randomization is actually due to TLSF's 8-byte alignment. Even without shuffling, `perlbench` runs substantially slower with TLSF than a heap that provides 16-byte aligned objects.

`gcc` is a particularly short-lived benchmark, running for just over 3 seconds on the train input. STABILIZER re-randomizes more frequently, backing off as execution continues. Because `gcc` spends its whole execution time during the frequent re-randomization period, it will never fully warm its caches or branch predictor tables.

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 some cases, STABILIZER improves the performance of benchmarks. Benchmarks are unlikely to exhibit cache conflicts and branch aliasing for repeated random layouts. Two programs (`mcf` and `hmmer`) show improved performance only when global and heap randomization are enabled. Stack randomization improves the performance of two more benchmarks (`lbm` and `libquantum`). Code randomization slightly improves the performance of `lbm` and `libquantum`; we attribute this to the elimination of branch aliasing [14].

7. 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` optimization level. Two benchmarks, `gromacs` and `wrf`, failed to build at the `-O3` optimization level. The impact of optimizations was evaluated on the remaining 18 benchmarks. Figure 5 shows the speedup of `-O3` over `-O2` for all benchmarks. Running benchmarks with STABILIZER guarantees normally distributed execution times, so we can apply statistical methods to determine the effect of `-O3` versus `-O2`.

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.

We first apply the two-sample t-test to determine whether `-O3` provides a statistically significant performance improvement over `-O2` for each benchmark. With a 95% confidence level, we find that there is a statistically significant difference between `-O2` and `-O3` for 12 of 18 benchmarks. While this result may suggest that `-O3` does have an impact, this result comes with a caveat: `bzip2`, `gcc`, `libquantum`, and `mcf` show a statistically significant *increase* in execution time with the added optimizations.

7.1 Analysis of Variance

Evaluating optimizations with pairwise t-tests is error prone. This methodology runs a high risk of erroneously rejecting the null hypothesis. In this case, the null hypothesis is that `-O2` and `-O3` optimization levels produce execution times with

the same distributions. Using analysis of variance, we can determine if `-O3` has a significant effect over all the samples.

We run ANOVA with the same 16 benchmarks, at both `-O2` and `-O3` optimization levels. For this configuration, the optimization level and benchmarks are the independent factors (specified by the experimenter), and the execution time is the dependent factor.

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.

The results show an F-value of 1.185 for one degree of freedom (the choice between `-O2` and `-O3`). The F-value is drawn from the F distribution [9]. The cumulative probability of observing any value from $F(1) > 1.185$ is called the p-value. For this experiment, we find a p-value of 0.291. Because this p-value is not less than our significance level $\alpha = 0.05$, we fail to reject the null hypothesis and must conclude that compared to `-O2`, `-O3` *optimizations are not statistically significant at a 95% confidence level.*

8. Future Work

We plan to extend STABILIZER to randomize code at finer granularity. Instead of relocating whole functions, STABILIZER can 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 can 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 is intractable: the 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. Upon detecting these problems, STABILIZER could trigger a complete or partial re-randomization of layout in an attempt 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) repeatedly re-randomizing layouts 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 statistical tests like ANOVA. We demonstrate STABILIZER's efficiency ($< 5\%$ 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`.

References

[1] A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 7 –18, feb. 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. Technical report, Gaithersburg, MD, United States, 2010.

[3] E. D. Berger and B. G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[4] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, June 2001.

[5] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 8–8, Berkeley, CA, USA, 2003. USENIX Association.

[6] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In

SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium, pages 17–17, Berkeley, CA, USA, 2005. USENIX Association.

[7] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining generational and conservative garbage collection: framework and implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '90, pages 261–269, New York, NY, USA, 1990. ACM.

[8] R. Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420, 1964.

[9] W. Feller. *An Introduction to Probability Theory and Applications*, volume 1. John Wiley & Sons Publishers, 3rd edition, 1968.

[10] T. M. Foundation. Buildbot/talos. `https://wiki.mozilla.org/Buildbot/Talos`.

[11] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[12] G. Hamerly, E. Perelman, J. Lau, B. Calder, and T. Sherwood. Using machine learning to guide architecture simulation. *J. Mach. Learn. Res.*, 7:343–378, Dec. 2006.

[13] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[14] D. A. Jiménez. Code placement for improving dynamic branch prediction accuracy. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 107–116, New York, NY, USA, 2005. ACM.

[15] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 339–348, Washington, DC, USA, 2006. IEEE Computer Society.

[16] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A new dynamic memory allocator for real-time systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, ECRTS '04, pages 79–86, Washington, DC, USA, 2004. IEEE Computer Society.

[17] I. Molnar. Exec-shield. `http://people.redhat.com/mingo/exec-shield/`.

[18] D. A. Moon. Garbage collection in a large LISP system. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 235–246, New York, NY, USA, 1984. ACM.

[19] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In M. L. Soffa and M. J. Irwin, editors, *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-XIV, Washington, DC, USA*, pages 265–276. ACM, Mar. 2009.

[20] G. Novark and E. D. Berger. DieHarder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 573–584, New York, NY, USA, 2010. ACM.

[21] 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.

[22] T. C. Project. Performance dashboard. `http://build.chromium.org/f/chromium/perf/dashboard/overview.html`.

[23] The PaX Team. The PaX project. `http://pax.grsecurity.net`, 2001.

[24] D. Tsafrir and D. Feitelson. Instability in parallel job scheduling simulation: the role of workload flurries. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, Apr. 2006.

[25] D. Tsafrir, K. Ouaknine, and D. G. Feitelson. Reducing performance evaluation sensitivity and variability by input shaking. In *Proceedings of the 2007 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 231–237, Washington, DC, USA, 2007. IEEE Computer Society.

[26] H. Xu and S. J. Chapin. Improving address space randomization with a dynamic offset randomization technique. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 384–391, New York, NY, USA, 2006. ACM.

[27] J. Xu, Z. Kalbarczyk, and R. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, pages 260–269, Oct. 2003.