STABILIZER: Enforcing Predictable and Analyzable Performance

Charlie Curtsinger Emery D. Berger

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

Abstract

Modern architectures have made program behavior brittle and unpredictable, making software performance highly dependent on its execution environment. Even apparently innocuous changes, such as changing the size of an unused environment variable, can—by altering memory layout and alignment—degrade performance by 33% to 300%. This unpredictability makes it difficult for programmers to debug or understand application performance. It also greatly complicates the evaluation of performance optimizations, since slight changes in the execution environment can have a greater impact on performance than a typical optimization.

We present STABILIZER, a compiler and runtime system that enforces reliable performance. By comprehensively randomizing the placement of functions, stack frames, and heap objects in memory, STABILIZER provides predictable performance with high probability. Random placement makes bad layouts unlikely, and re-randomization ensures they are short-lived when they do occur. We demonstrate that STABILIZER effectively eliminates measurement bias with minimal overhead, allowing it to be used in deployment to ensure predictable performance. In addition, STABILIZER enables statistically rigorous performance evaluation. We demonstrate its use by testing the effectiveness of standard optimizations used in the LLVM compiler; we find that, across the SPEC CPU2000 and CPU2006 benchmark suites, the effect of the `-O3` optimization level is indistinguishable from noise.

1. Introduction

Modern architectures have made program behavior brittle and unpredictable. Multi-level cache hierarchies and deeply pipelined architectures can cause execution times of individual instructions to vary over two orders of magnitude. Application performance is greatly affected by subtle details of individual chips, such as the size or implementation of caches and branch predictors. Even apparently innocuous changes, such as changing the size of an unused environment variable or the link order of object files, can dramatically alter application performance. Mytkowicz et al. demonstrate that such changes can degrade performance by 33% to 300% [14].

This sensitivity of application performance to its environment has numerous serious consequences:

- **Impairs performance understanding:** Environmental sensitivity makes it difficult for programmers to understand the performance of their applications. Even inserting a single *non-executed* `printf` statement can, by changing program layout, unexpectedly alter application performance.
- **Risks performance pathologies in deployed code:** Because subtle changes in input, compiler, libraries, and architecture can significantly degrade performance, deployed code may perform drastically worse than in testing.
- **Undermines performance analysis and research:** Since even a slight change in the environment can have a greater impact on performance than a typical optimization, it is difficult for developers or researchers to judge the effectiveness of performance optimizations with any degree of confidence.

Contributions

This paper presents STABILIZER, a system that eliminates performance pathologies, provides predictable performance, and enables rigorous performance analysis of programs:

Comprehensive Layout Randomization. STABILIZER consists of a compiler and runtime library that repeatedly randomize the placement of globals, functions, stack frames, and heap objects during execution. Intuitively, STABILIZER makes it unlikely that object and code layouts will be "unlucky". By periodically re-randomizing, STABILIZER further reduces these odds.

Predictable Performance. We show analytically and empirically that STABILIZER's use of re-randomization imposes a normal distribution on execution time, enabling predictability using standard statistical approaches. STABILIZER makes performance outliers quantifiably unlikely.

STABILIZER makes program execution independent of the execution environment. It prevents performance pathologies due to a wide range of factors, ranging from library versions down to architectural details. STABILIZER often operates with sufficiently low overhead that it can be used in deployment (below 5% for a third of benchmarks, and a median of 13.3%).

Sound Performance Analysis. By generating a normal distribution of execution times, STABILIZER makes it possible to perform rigorous and statistically sound performance analyses. STABILIZER allows researchers to answer the question: does a given change to a program truly improve its performance, or is it indistinguishable from noise? We use STABILIZER to assess the effectiveness of compiler optimizations in the LLVM compiler [10]. Across both the

SPEC CPU2000 and SPEC CPU2006 benchmark suites, 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.

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 low overhead and avoidance of measurement bias. Section 7 leverages STABILIZER to rigorously evaluate the effectiveness of LLVM's standard optimizations, 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 rigorous performance evaluation because of a lack of independence. Any change to a program's code or execution environment can lead to a different memory layout. Prior work has shown that small changes in memory layout can degrade performance by as much as 300% [14], making it impossible to evaluate any particular change in isolation.

2.1 Comprehensive Layout Randomization

By randomizing program layout dynamically, STABILIZER makes layout independent of changes in code or execution environment. STABILIZER performs extensive randomization, dynamically randomizing the placement of a program's functions, stack frames, heap objects, and globals. Code is randomized at a function granularity, and each function executes on a randomly-placed stack frame. STABILIZER also periodically *re-randomizes* code at runtime.

2.2 Normally-Distributed Execution Time

STABILIZER's 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 randomization strategy leads to normally-executed distributions as follows. Each random layout contributes to the total execution time. Total execution time is thus proportional to the average over many different layouts. The *central limit theorem* states that "the mean of a sufficiently large number of independent random variables ... will be approximately normally distributed" [6]. As long as STABILIZER re-randomizes layout a sufficient number of times, and each layout is chosen independently, then execution time will be normally distributed. Section 5 provides a more detailed analysis.

2.3 Predictable Performance

Ensuring that execution time conforms to the normal distribution enables predictable performance, bounding the likelihood of outliers; the chance of a normally-distributed random value (here, execution time) falling within two standard deviations of the mean is 95%.

2.4 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 false hypotheses than more general (non-parametric) tests that make no assumptions about distribution.

2.5 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 [6] to determine whether performance varies across 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 specified confidence (typically 5%), we say that the null hypothesis has been rejected—the distributions are not the same, so the treatment had a significant effect.

2.6 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, and the effect of each treatment [6]. Section 7 presents the use of STABILIZER and ANOVA to evaluate the effectiveness of compiler optimizations in LLVM.

3. Related Work

Randomization for Security. Most 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, 12]. 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) [21, 9]. 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 [3, 20, 4]. 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 fine granularity, and re-randomizes during execution.

Heap Randomization. DieHard uses heap randomization to prevent memory errors [2]. 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 [15]. Heap randomization alone is not sufficient to make performance predictable, but STABILIZER uses DieHard as the basis for all of its randomizations.

Predictable Performance. Quicksort is a classic example of using randomization for predictable performance [8]. Random pivot selection eliminates the possibility of a worst-case input, and bounds the probability of observing quicksort's $O(n^2)$ worst-case time complexity.

Base Randomization	ASLR	TRR	ASLP	Address Obfuscation	Dynamic Offset	B.S.DV [4]	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.

Randomization has also been applied to probabilistically analyzable real-time systems. Quiñones et. al showed that a random cache replacement policy enables probabilistic worst-case execution time analysis, while still providing good performance. This is a significant improvement over conventional hard real-time systems, where analysis of cache behavior relies on complete information. STABILIZER is a more general approach, but relies on some of the same probabilistic arguments for predictable performance.

Rigorous Performance Evaluation. Mytkowicz et al. observe that environmental sensitivities can degrade program performance by as much as 300% [14]. They propose experimental setup randomization to account for these sensitivities. 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" [18, 19]. Georges et al. propose statistically rigorous techniques for Java performance evaluation [7]. While prior techniques for rigorous performance evaluation require many runs over a wide range of environmental factors, STABILIZER enables efficient, rigorous performance evaluation by eliminating the dependence between experimental setup and program layout.

4. STABILIZER Implementation

STABILIZER fully randomizes the layout of its host application. This randomization dynamically randomizes the layout of heap objects, code, stack frames, and globals. Each randomization consists of a compiler transformation and runtime support. Figure 1 shows the process for building a program using STABILIZER. Each source file is first compiled to LLVM bytecode using the `llvmc` compiler driver. The resulting bytecode files are linked and processed with LLVM's `opt` tool running the STABILIZER compiler pass. The resulting executable is then linked with the STABILIZER runtime library, which performs the dynamic layout randomization. The following sections describe the implementation of each randomization in detail.

4.1 Heap Randomization

STABILIZER applies heap randomization using the DieHard memory allocator [2, 16], a bitmap-based allocator that fully randomizes individual object placement across a heap that is some factor M larger than required (in Stabilizer, we set M to $4/3$). Figure 2, taken from Novark et al. [16], presents an overview of DieHard's internals. The following two paragraphs are adapted from that paper:

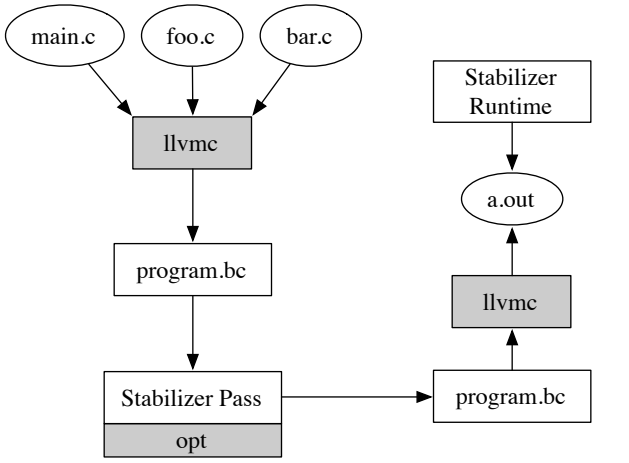


Figure 1. The process for building an application with STABILIZER.

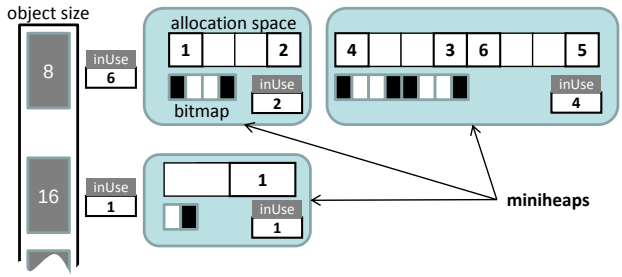


Figure 2. The DieHard memory allocator's heap layout (diagram from Novark et al. [16]); STABILIZER uses DieHard as a source of random objects for the heap, code, and stack frames.

DieHard allocates memory from increasingly large chunks that we call *miniheaps*. Each miniheap contains objects of exactly one size. DieHard allocates new miniheaps to ensure that, for each size, the ratio of allocated objects to total objects is never more than $1/M$. Each new miniheap is twice as large, and thus holds twice as many objects, as the previous largest miniheap.

Allocation randomly probes a miniheap's bitmap for the given size class for a 0 bit, indicating a free object available for recla-

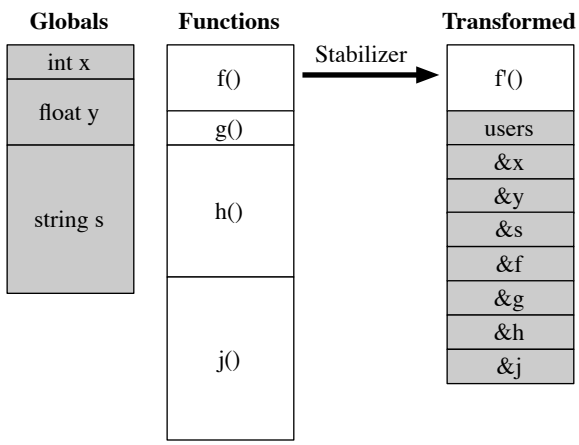


Figure 3. STABILIZER adds a relocation table to the end of each function, making every function independently relocatable. White boxes contain code and shaded boxes contain data.

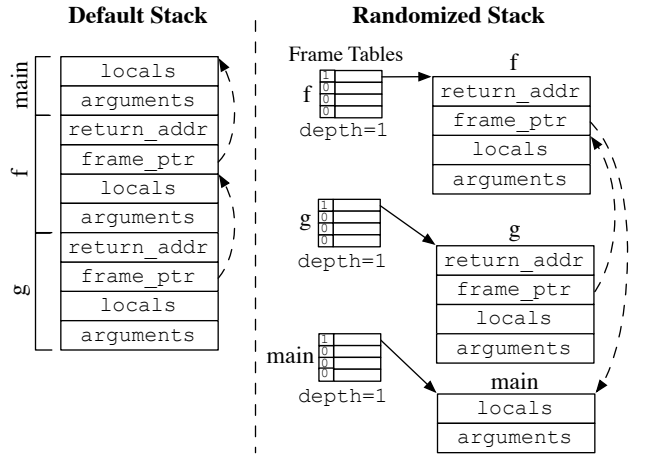


Figure 4. STABILIZER makes the stack non-contiguous. Each function has a frame table, which stores a frame for each recursion depth.

mation, and sets it to 1. This operation takes $O(1)$ expected time. Freeing a valid object resets the appropriate bit, which is also a constant-time operation.

Unlike conventional allocators, DieHard does not cache and reuse recently freed heap memory, but instead selects from the full range of available heap memory on every allocation, making each allocation's placement independent of the last.

STABILIZER's compiler pass rewrites calls to `malloc` and `free` (exposed in LLVM IR) to target the DieHard heap. Note that STABILIZER cannot move heap-allocated objects during execution because this is not permitted by C/C++.

4.2 Code Randomization

STABILIZER randomizes code at the function granularity. Every transformed function has a *relocation table* (see Figure 3), which is placed immediately following the code for the function. The relocation table contains a `users` counter that tracks the number of active users of the function, followed by the addresses of all globals and functions referenced by the relocated function.

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.

Pointers to entries in the relocation table actually point into the following function. Each function refers to its own adjacent relocation table using relative addressing modes, so 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 DieHard 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.

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 Randomization of Globals

STABILIZER randomizes the locations of global objects by allocating them on the DieHard heap at startup. If code randomization is also enabled, globals are already accessed indirectly through the function relocation table. In this case, the new random address for the global replaces the default location in the relocation table. If code randomization is disabled, STABILIZER rewrites accesses to globals to be indirected through a pointer global variable that holds the random address of the global. As with heap objects, STABILIZER does not relocate globals after startup.

4.4 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 via DieHard, 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 `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.

4.5 Architecture-Specific Implementation Details

STABILIZER runs on the x86, x86_64 and PowerPC architectures. Most of the implementation details are the same, but some specific modifications were required for portability.

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 (notably, Mac OS X), this memory is extremely limited.

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 if 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.

4.6 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 were called in 3 consecutive randomization periods are marked as persistent. The STABILIZER runtime preemptively relocates persistent functions at 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, globals, 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 demonstrates that, for programs that meet several basic assumptions described below, STABILIZER's randomization results in normally-distributed execution times. Section 6 empirically verifies this analysis.

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

Base case: a single loop. Consider a small program that runs repeatedly in a loop. 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:

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

When this same program is run with STABILIZER, every iteration is run with a different layout l_i with execution time e_i. Running this program with STABILIZER for 1000 iterations will 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. We have 1000). 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. It should be easy to see that 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 is *not* coupled to iterations in programs, so this is clearly not true. However, it is easy to see that if STABILIZER re-randomizes every n iterations, we can simply redefine an "iteration" to be n passes over the same code.

Programs with phase behavior. The second condition is that the program is simply a loop repeating the same code over and over again. 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.

This program can be decomposed into subprograms, each 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 a weighted sum of all the subprograms. The sum of two normally distributed random variables is also normally distributed, so 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.

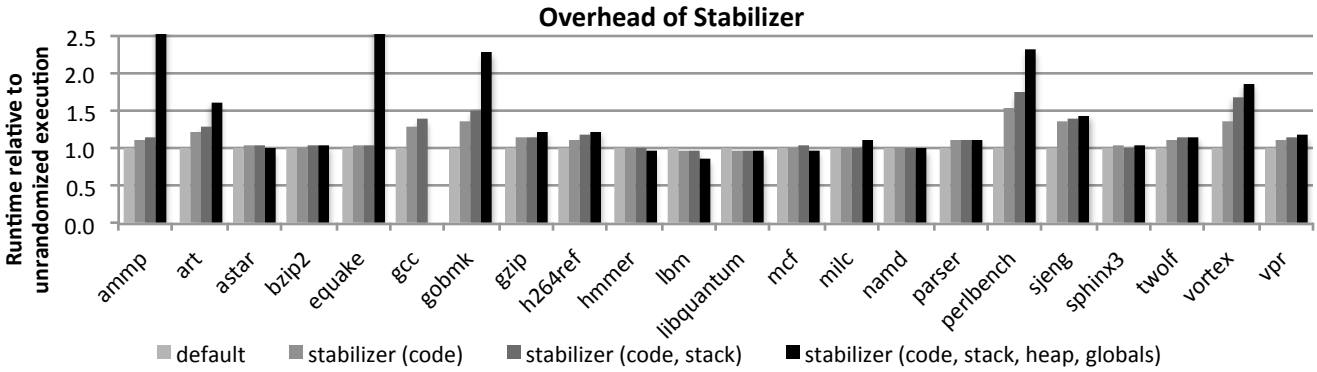


Figure 5. Overhead of STABILIZER relative to unrandomized execution. The values for `ammp` and `equake` with all randomizations enabled are 2.78 and 4.5 respectively. For a majority of benchmarks, STABILIZER imposes below 14% overhead, and in four cases actually improves performance.

5.1 Assumptions

STABILIZER can only guarantee normality when a program is randomized a sufficient number of times. Code layout randomization is performed at function granularity, so a program with a single function will not be re-randomized. This situation could arise in large programs if aggressive inlining eliminates most of the program's function calls. Most programs have a large number of functions, which allows STABILIZER to re-randomize code frequently enough to guarantee normality.

STABILIZER supports unmanaged languages, so live heap objects are not relocated. Every allocation returns a randomly selected heap address, so programs with a sufficiently large number of short-lived heap objects will be effectively re-randomized. This requirement corresponds to the generational hypothesis for garbage collection, which has also been shown to be true in unmanaged environments [5, 13].

Searching for Optimal Layouts

STABILIZER makes it possible to change the layout of a program dynamically. This raises the question: instead of randomizing layout, why not search for an optimal layout? Unfortunately, the space of possible layouts is prohibitively large. A program with just three functions and no globals or heap objects has $2^{47^6} \approx 7.8x10^{84}$ possible layouts for code and stack in a 64-bit address space (48 bits sign-extended, reserving the higher half for the kernel). As a comparison, there are an estimated 10^{80} atoms in the observable universe. Only considering cache ways, and not absolute addresses, a program with 35 cache line-sized functions will have a similar number of mappings onto a 16-way cache (not accounting for offset within the cache line).

Even if we disregard the gigantic state space for layouts, we are left with a problem of generality. The impact of layout is machine- and input-dependent. A good layout for one input may be the worst case for another. This means the search for an optimal layout would need to be run on every execution.

A deterministic memory layout opens the door for degenerate execution environments or inputs, which consistently lead to performance outliers. This is similar to quicksort with deterministic pivot selection [8]. Introducing randomization makes it impossible for any input to consistently produce the worst-case behavior. Unlike a fixed, "optimal" layout, a randomized layout can be generated efficiently, will have no degenerate cases, and bounds the probability of observing performance outliers for any input.

6. STABILIZER Evaluation

We evaluate STABILIZER in two dimensions. First, we test the claim that STABILIZER eliminates the impact of execution environment on program performance and leads to normally distributed execution times. Next, we quantify the overhead of running programs with STABILIZER relative to unrandomized execution.

All evaluations were performed on an dual-socket 6-core Intel Xeon X5650 running at 2.67GHz equipped with 24GB of RAM. Each core has 32KB of data L1 cache, 32KB of instruction L1 cache, and 256KB of unified L2 cache. Each socket has a single 12MB L3 cache shared by all cores. The system runs version 2.6.32 of the Linux kernel (unmodified). All programs (with and without STABILIZER) were built using version 2.9 of the LLVM compiler with the GCC 4.2 front-end using `-O2` optimizations unless otherwise specified.

Benchmarks. We evaluate STABILIZER on the SPEC CPU2006 and CPU2000 benchmark suites. From SPEC CPU 2006, we ran `astar`, `bzip2`, `gcc`, `gobmk`, `h264ref`, `hmmer`, `lbm`, `libquantum`, `mcf`, `milc`, `namd`, `perlbench`, `sjeng`, and `sphinx3`. We were unable to run `omnetpp`, `xalancbmk`, `dealII`, `soplex`, `povray`, and all the Fortran benchmarks. LLVM does not support the Fortran front-end, and STABILIZER currently does not support C++ exceptions. All SPEC CPU2006 benchmarks were run with train inputs.

We also ran the `ammp`, `art`, `crafty`, `equake`, `gzip`, `parser`, `twolf`, `vortex`, and `vpr` benchmarks from SPEC CPU2000. We excluded benchmarks that have more recent versions in SPEC CPU2006 (`gcc`, `mcf`, and `perlbmk`). We were unable to run `gap` and `mesa` because they would not build on our 64-bit machine. `eon` uses exceptions, so it is not yet supported by STABILIZER. All Fortran benchmarks from SPEC CPU2000 were also included. SPEC CPU 2000 benchmarks were run with ref inputs.

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 10 times, adding a random number of bytes (between 0 and 4096) to the shell environment variables on each run.

Without STABILIZER, 10 benchmarks exhibit execution times that are not randomly distributed with 95% confidence: `ammp`, `astar`, `gzip`, `lbm`, `libquantum`, `mcf`, `milc`, `namd`, `vortex`,

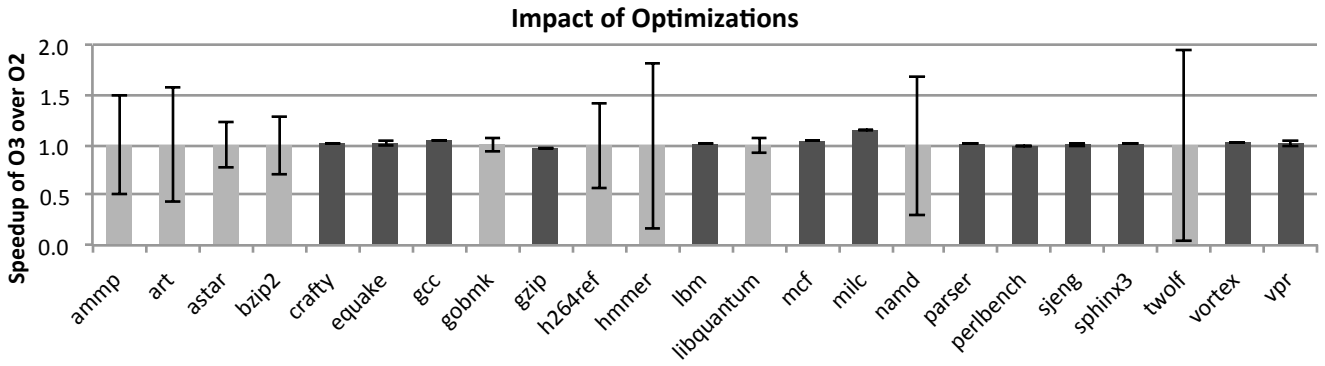


Figure 6. Speedup of `-O3` over the `-O2` optimization level in LLVM. Error bars indicate the p-values for the T-test comparing `-O2` and `-O3`.Benchmarks with dark bars showed a statistically significant change with `-O3` relative to `-O2`. However, despite these individual results of statistical significance, the data do not indicate significance across the entire suite of benchmarks (see Section 7.1).

and `vpr`. Running each of these benchmarks with STABILIZER leads to normally distributed execution times.

Figure 7 shows the distributions of four benchmarks 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). The quantile of every sample is computed. Each data point is placed at the intersection of the sample and reference distributions' quantiles. If the samples come from the reference distribution (modulo differences in mean and variance), the points will fall along a straight line in the diagonal.

Result: These figures demonstrate that STABILIZER imposes normally distributed execution times. This normality holds even for programs with execution times that were not originally normally distributed (that is, without STABILIZER).

6.2 Efficiency

Figure 5 shows the overhead of STABILIZER relative to unrandomized execution. Each benchmark was run 10 times for each configuration. The results show that for most benchmarks, code and stack randomization ad under 13% overhead. With all randomizations enabled, STABILIZER adds a median overhead of 14.5%.

When does STABILIZER hurt performance?

The overhead added by STABILIZER is mostly due to the reduced locality of a randomized program. Code and stack randomization both add additional logic to function invocation, but in practice this extra work does not significantly degrade performance. Programs run with STABILIZER use a larger portion of the virtual address space, putting additional pressure on the TLB. Randomly placed code and data are sparse across this increased virtual memory range, reducing cache utilization. In most cases the added overhead is modest, but for larger programs (`gcc`, `gobmk`, `perlbench`, `sjeng`, and `vortex`) it can measurably degrade performance.

With all randomizations enabled, STABILIZER adds significant overhead for `ammp`, `art`, `equake`, `gobmk`, `perlbench` and `vortex`. The majority of this overhead is due to startup costs with global randomization and the increased cost of heap allocations. Global randomization is not performed lazily, so for some short running benchmarks with many globals (`art`, `gobmk`, `perlbench`, and `vortex`) startup time contributes a large fraction of the overhead. This overhead could be reduced by randomizing globals lazily, which we leave for future work.

When does STABILIZER improve performance?

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 does slightly improve the performance of `lbm` and `libquantum`, likely due to elimination of branch aliasing.

7. Sound Performance Analysis

STABILIZER enables a sound method for performance evaluation, which we use to evaluate the effectiveness of LLVM's `-O3` optimization level. Figure 6 shows the speedup of `-O3` over `-O2` for all benchmarks. Running benchmarks with STABILIZER guarantees normally distributed execution times, so we can apply rigorous statistical methods to determine the effect of O3.

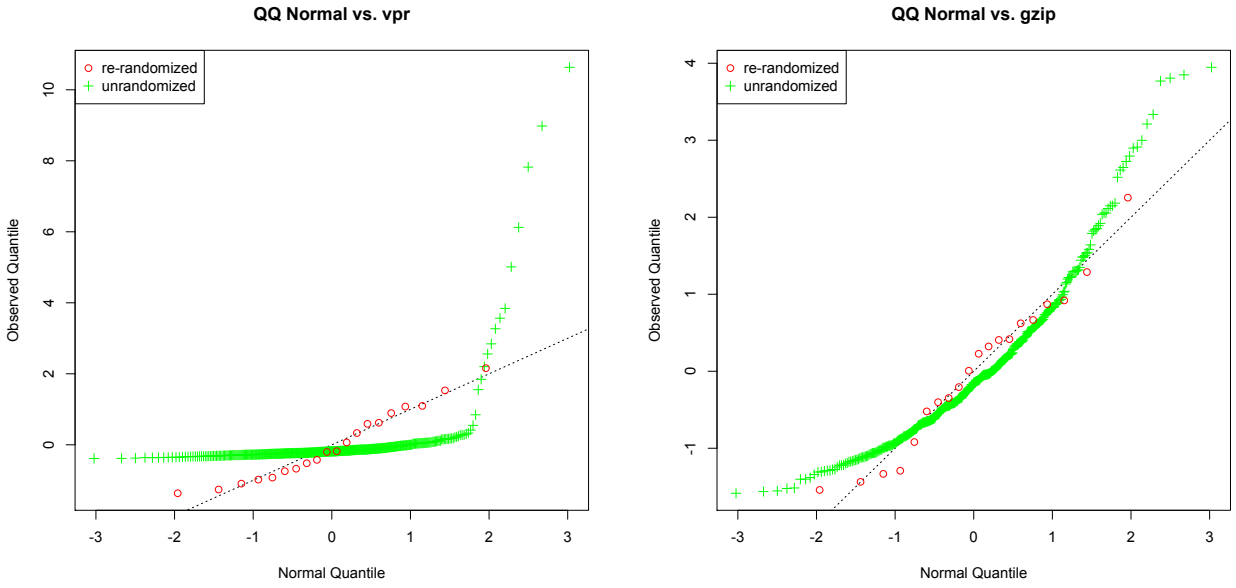
We first apply the two-sample T-test to determine whether `-O3` provides a statistically significant performance improvement over `-O2`. With a 95% confidence level, we determined that there is a statistically significant difference between `-O2` and `-O3` for 13 of 23 benchmarks. While this result may suggest that `-O3` does have an impact, there is a serious caveat: `gzip` and `perlbench` 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.

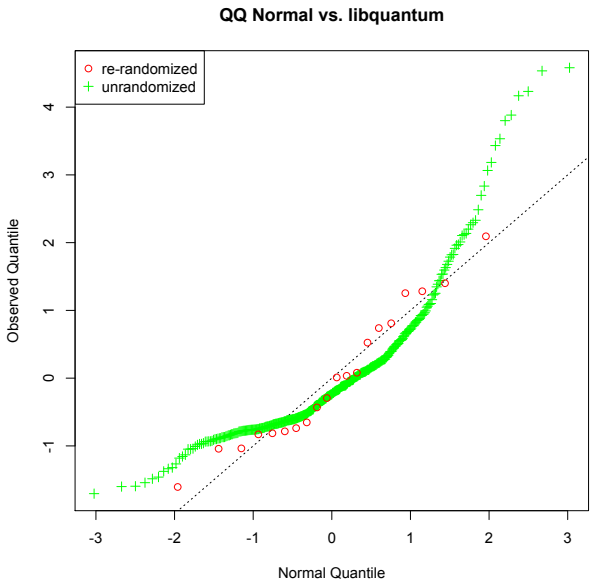
We run ANOVA with the complete set of benchmark runs at both `-O2` and `-O3` optimization levels. For this configuration, the optimization level and benchmarks are the independent factors (specified by the experimenter), and the execution time is the dependent factor.

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. Not surprisingly, 99.9% of the variance in our experiment is due to differences between benchmarks. Of the remaining variance, 46.5% is due to

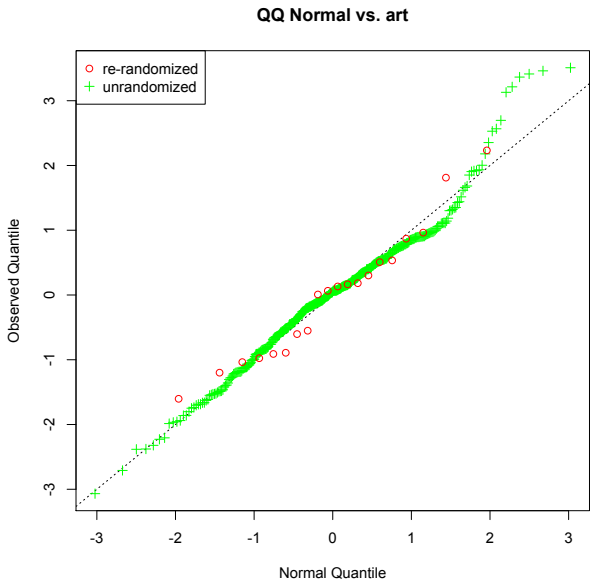


(a) vpr

(b) gzip



(c) libquantum



(d) art

Figure 7. Immunity from measurement bias: Quantile-quantile plots comparing the distribution of execution times for three benchmarks to the normal distribution. The solid line indicates where points drawn from a normal distribution will fall. In the first three cases, unrandomized execution times fall well outside of the range for normality, while runs with STABILIZER closely match the normal quantile line. The figure for `art` shows normally distributed execution times with and without randomization.

the interaction between specific benchmarks and `-O3`, 47.5% is due to random variation, and just 6.0% is due to the `-O3` optimizations.

Result: Using the F-test, we can determine if the variances are statistically significant [6]. We fail to reject the null hypothesis, and must conclude that `-O3` *optimizations are not statistically significant with 95% confidence*.

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 effectively randomizes the history portion of the branch predictor table, addressing another source of potential performance outliers.

Parallel programs suffer from the same sensitivity to layout as serial programs, but can suffer from more dramatic performance outliers due to false sharing [11]. False sharing occurs when two threads share different variables that happen to fall on the same cache line. This phenomenon incurs the same cache coherence penalty as true sharing. We believe that STABILIZER's randomization of heap objects and globals will make false sharing probabilistically unlikely. In its current form, STABILIZER does not efficiently support parallel programs. The cost of atomic operations performed at function call boundaries accounts for most of this overhead. Additionally, on some platforms, thread-local accesses are prohibitively expensive. We plan to address this overhead by reducing the amount of STABILIZER runtime data shared between threads.

Finally, DieHard may not be the best fit for the randomization of large, fixed-size functions and stack frames. Its power-of-two size classes lead to increased demand for virtual address space, placing unneeded pressure on the TLB. We plan to implement a specialized allocator that reduces the cost of STABILIZER's code and stack randomization.

9. Conclusion

Modern processor architectures are highly dependent on program layout. Layout can be affected by input, code changes, program link order, optimizations, shared library versions, and even shell environment variables. These dependencies lead to highly unpredictable performance, complicating performance evaluation and optimization.

This paper presents STABILIZER, a compiler and runtime system for comprehensive layout randomization. STABILIZER dynamically relocates functions, stack frames, heap objects, and globals on every execution, and repeatedly relocates code and stack during execution. STABILIZER makes performance outliers statistically unlikely, and makes execution times conform to a normal distribution. Normally distributed execution times enable a wide range of statistical techniques for performance evaluation. We use STABILIZER to rigorously evaluate the effectiveness of LLVM's `-O3` optimization level across the SPEC CPU2000 and CPU2006 benchmark suites, and found no statistically significant improvement.

We plan to make STABILIZER freely available by publication time, and encourage researchers to use it as a basis for sound performance evaluation.

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] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[11] T. Liu and E. D. Berger. Sheriff: precise detection and automatic mitigation of false sharing. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 3–18, New York, NY, USA, 2011. ACM.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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*, page 10 pp., april 2006.

[19] 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.

[20] 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.

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