Efficient Tracing of Cold Code via Bias-Free Sampling

Baris Kasikci [1*], Thomas Ball [2†], George Candea [1‡], John Erickson [2§], and Madanlal Musuvathi [2¶]

[1]School of Computer and Communication Sciences, EPFL
[2]Microsoft

Abstract

Bugs often lurk in code that is infrequently executed (i.e., cold code), so testing and debugging requires tracing such code. Alas, the location of cold code is generally not known a priori and, by definition, cold code is elusive during execution. Thus, programs either incur unnecessary runtime overhead to "catch" cold code, or they must employ sampling, in which case many executions are required to sample the cold code even once.

We introduce a technique called *bias-free sampling* (BfS), in which the machine instructions of a dynamic execution are sampled independently of their execution frequency by using breakpoints. The BfS overhead is therefore independent of a program's runtime behavior and is fully predictable: it is merely a function of program size. BfS operates directly on binaries.

We present the theory and implementation of BfS for both managed and unmanaged code, as well as both kernel and user mode. We ran BfS on a total of 679 programs (all Windows system binaries, Z3, SPECint suite, and on several C# benchmarks), and BfS incurred performance overheads of just 1–6%.

1 Introduction

Monitoring a program's control-flow is a fundamental way to gain insight into program behavior [5]. At one extreme, we can record a bit per basic block that measures whether or not a block executed over an entire execution (coverage) [29]. At another extreme, we can record the dynamic sequence of basic blocks executed (tracing) [28]. In between these two extremes there is a wide range of monitoring strategies that trade off runtime overhead for precision. For example, record-replay systems [12, 15] that record most execution events in a program incur a large overhead, whereas sampling strategies that collect fewer runtime events for both profiling and tracing [16] incur less overhead.

In testing and debugging, there is a need to sample infrequently executed (i.e., cold) instructions at runtime, because bugs often lurk in cold code [9, 23]. However, we don't know a priori which basic blocks will be cold vs. hot at runtime, therefore we cannot instrument just the cold ones. To make matters worse, traditional temporal sampling techniques [21, 24] that trade off sampling rate for sampling coverage can miss cold instructions when the sampling rate is low, requiring many executions to gain acceptable coverage. As a result, developers do not have effective and efficient tools for sampling cold code.

In this paper, we present a non-temporal approach to sampling that we call *bias-free sampling* (BfS). BfS is guaranteed to sample cold instructions without oversampling hot instructions, thereby reducing the overhead typically associated with temporal sampling.

The basic idea is to sample any instruction of interest *the next time* it executes and *without* imposing any overhead on any other instructions in the program.

We do this using code breakpoints (a facility present in all modern CPUs) dynamically. We created *lightweight code breakpoint* (LCB) monitors for both the kernel and user mode of Windows for both native (with direct support in the kernel) and managed applications (with a user-space monitor) on both Intel and ARM architectures.

To ensure that none of the cold instructions are missed, the bias-free sampler inserts a breakpoint at *every* basic block in the program, both at the beginning of the program execution and periodically during the execution. This ensures at least one sample per period of every cold instruction. We also show how to sample without bias hot instructions independently of their execution frequency at a low rate.

Devising an efficient solution that works well in prac-

*baris.kasikci@epfl.ch
†tball@microsoft.com
‡george.candea@epfl.ch
§jerick@microsoft.com
¶madanm@microsoft.com

tice on a large set of programs requires solving multiple challenges: (a) processing a large number of breakpoints, in the worst case simultaneously on every instruction in the program (existing debugging frameworks are unable to handle such high volumes because their design is not optimized for a large number of breakpoints that must be processed quickly); (b) handling breakpoints correctly in the presence of a managed code interpreter and JIT optimizations (managed code gets optimized during execution, therefore it cannot be handled the same way as native code); and (c) preserving the correct semantics of programs and associated services, such as debuggers.

A particular instance of LCB that we built is the lightweight code coverage (LCC) tool. We have successfully run LCC at scale to simultaneously measure code coverage on all processes and kernel drivers in a standard Windows 8 machine with imperceptible overheads. We also have extended LCC with the ability to record periodic code coverage logs. LCC is now being used internally at Microsoft to measure code coverage.

Using breakpoints overcomes many of the pitfalls of code instrumentation. CPU support for breakpoints allows setting (a) a breakpoint on *any* instruction, (b) an arbitrary number of breakpoints, and (c) setting or clearing a breakpoint without synchronizing with other threads (with the exception of managed code) that could potentially execute the same instruction.

The contributions and organization of this paper are:

- We analyze and dissect common approaches to cold code monitoring, showing that there is need for improvement (§2);
- We present our BfS design (§3) and its efficient and comprehensive implementation using breakpoints for both the kernel and user mode of Windows for both native and managed applications (§4);
- We show on a total of 679 programs that with our implementation of LCB, our coverage tool LCC, which places a breakpoint on every basic block in an executable and removes it when fired, has an overhead of 1-2% on a variety of native C benchmarks and an overhead of 1-6% on a variety of managed C# benchmarks (§5);
- We show how to use periodic BfS to extend LCC to quickly build interprocedural traces with overheads in the range of 3-6% (§6).

§7 discusses related work and §8 concludes with a discussion of applications for BfS.

2 From Rewriting to Bias-Free Sampling

In this section, we provide background on the approaches used to monitor program behavior, and outline the conceptual path that leads to our proposed technique.

2.1 Program Rewriting

A traditional approach to monitoring program behavior is *static program rewriting* as done by Gcov [13], which takes as input an executable E and outputs a new executable E' that is functionally the same as E except that it monitors the behavior of E. At Microsoft, many such monitoring tools have been built on top of the Vulcan binary rewriting framework [27], such as the code coverage tool `bbcover`. Vulcan provides a number of program abstractions, such as the program control-flow graph, and the tool user can leverage these abstractions to then use the Vulcan APIs to add instructions at specific points in the binary. Vulcan ensures that the branches of the program are adjusted to reflect this addition of code.

Another approach to monitoring is *dynamic program rewriting*, as done by DynInst [7] and Pin [22], as well as Microsoft's Nirvana and iDNA framework [6]. Many of the tools built with rewriting-based approaches, both static and dynamic, use "always-on" instrumentation (they keep the dynamically-added instrumentation until the program terminates), even when for goals that should be much less demanding, like measuring code coverage.

2.2 Efficient Sampling

Static or dynamic program rewriting approaches that are always-on incur prohibitive overheads, and they cannot sample cold code in a bias-free manner.

In 2001, Arnold et al. introduced a framework for reducing the cost of instrumented code that combines instrumentation and counter-based sampling of loops [24]. In this approach, there are two copies of each procedure: The "counting" version of the procedure increments a counter on procedure entry and a counter for each loop back edge, ensuring that there is no unbounded portion of execution without some counter being incremented. When a user-specified limit is reached, control transfers from the counting version to a more heavily instrumented version of the procedure, which (after recording the sample) transfers control via the loop back edges back to the counting version. In this way, the technique can record more detailed information about acyclic intraprocedural paths on a periodic basis.

Hirzel et al. extended this method to reduce overhead further and to trace interprocedural paths [17]. They implemented "bursty tracing" using Vulcan, and report runtime overheads in the range of 3-18%. In further work [16] they sample code at a rate inversely proportional to its frequency, so that less frequently executed code is sampled more often. This approach is based on the premise that bugs reside mainly on cold paths.

Around the same time, Liblit et al. [21] proposed "Sampling the Bernoulli Way" in their paper on what

later was termed "cooperative bug isolation." The motivation for their approach was that classic sampling for measuring program performance "searches for the 'elephant in the haystack': it looks for the biggest consumers of time" [21]. In contrast, the goal is to look for needles (bugs) that may occur rarely, and the sampling rates may be very low to maintain client performance. This leads to the requirement that the sampling be statistically fair, so that the reported frequencies of rare events be reliable. The essence of their approach is to perform fair and uniform sampling from a dynamic sequence of events. To obtain sufficient samples of rare events, their approach relies on collecting a large number of executions.

2.3 Bias-Free Code Sampling

There's a fundamental tension between the desire to look for needles in a haystack (cold code), the use of bursty tracing, and Bernoulli sampling to achieve efficiency. Bursty tracing can trace cold code at a high cost; sampling is efficient, but it requires many runs before cold paths are sampled, and thus may incur a large overhead.

Consider the simple example of a hot loop containing an `if-then-else` statement where the `else` branch is very infrequently executed compared to the loop head—say the `else` branch executes once every million iterations of the loop. The desire to keep the sampling rate low for efficiency means it's unlikely that Bernoulli sampling or the bursty tracing approach will hit upon the one execution of the `else` branch in a million iterations.

Furthermore, we generally do not know a priori which code blocks will be cold during the execution of interest. Thus, we need a way to sample all code but not let the different frequencies of execution of the different code blocks influence the runtime performance overhead. In other words, the sampling rate of a code block should be (mostly) independent of how often it is executed. We say "mostly" because there still is a dependency: the block must be executed at least once for it to be sampled.

The basic idea behind our approach is (using the example above) that placing a breakpoint on the first instruction in the `else` branch guarantees that we will sample the next (albeit rare) execution of the `else` branch with no cost for the many loop iterations before that point. By refreshing this breakpoint periodically, we can obtain several samples of this rare event.

Looking at it from the other side, Bernoulli sampling gives equal likelihood that any of the million loop iterations of the loop's execution will be sampled. This may be fair to all the loop iterations, but it doesn't help identify the cold code. Cooperative bug isolation makes up for the fact that a single execution may not uncover cold code by the law of large numbers (of executions) to increase the confidence that a rare event will be sampled.

Bias-free code sampling is a way to sample cold events just as efficiently as Bernoulli sampling with far fewer executions.

3 Design

The core idea of BfS is to use breakpoints to: (a) sample cold instructions that execute only a few times during an execution, without over-sampling hot instructions; (b) sample the remaining instructions independently of their execution frequency. Algorithm 1 presents the BfS algorithm. We discuss the algorithm in its full generality before discussing particular instantiations.

3.1 Inputs

The algorithm takes as input three parameters. The parameter `K` ensures that the first `K` executions of any instruction are always sampled. Assuming a nonzero `K`, this ensures that rare instructions, such as those in exception paths, are always sampled when executed.

The second parameter `P` is the sampling distribution of the instructions. For instance, a memory leak detector [16] might only chose to sample memory access instructions, and accordingly `P` will indicate a zero probability for non-memory-accesses. Similarly, a data-race detector [18] might only choose memory accesses that are not statically provable as data-race-free. Among the instructions with non-zero probability, `P` might either dictate a uniform sampling, or bias towards some instructions, based on application needs. For instance, additional runtime profile information could be used to increase the bias towards hot instructions or towards instructions that are likely to be buggy.

The final parameter `R` determines the desired sampling rate, i.e., the number of samples generated per second. This indirectly determines the overhead of the algorithm. In the special case when `R` is infinity, the algorithm periodically refreshes breakpoints on all instructions selected according to `P`.

3.2 Cold Instruction Sampling

The algorithm maintains a map `BPCount` that determines the number of logical breakpoints set at a particular instruction. The algorithm ensures that a breakpoint is set at a particular instruction whenever its `BPCount` is greater than zero. When a breakpoint fires, this count is decremented and the breakpoint is removed only when this count is zero. Setting all entries of this array to `K` ensures that the first `K` executions of the instructions with nonzero probability in `P` are sampled.

Algorithm 1: Bias-free Sampling Algorithm

```
Input: int K, Dist P, int R

// BPCount[pc] > 0 implies pc has a breakpoint
Map < PC, int > BPCount
Map < PC, int > SampleCount
Set < PC > FreqInst

function Init
  For all pc with nonzero probability in P
    BPCount[pc] = K

function OnBreakpoint(pc)
  BPCount[pc]--
  SampleCount[pc]++
  SampleInstruction(pc)
  if SampleCount[pc] >= K then
    FreqInst.Add(pc)
  if SampleCount[pc] > K then
    ChooseRandomInst()

function Periodically()
  hitNum = NumBPInLastPeriod()
  if R is infinity then
    BPCount[pc]++ for all pc in FreqInst
    return
  while hitNum++ < R*Period do
    ChoseRandomInst()

function ChooseRandomInst()
  pc = Choose(P, FreqInst)
  BPCount[pc]++
  FreqInst.Remove(pc)
```

At one extreme,With `K` 1, `P` choosing only the first instruction in every basic block, and `R` as 0, we obtain an efficient mechanism for code coverage, described as LCC in Section 5. On the other extreme, when `K` is set to infinity, one gets full execution tracing.

The algorithm maintains `FreqInst`, a set of instructions that have executed `K` or more times. Periodically, the algorithm adds breakpoints to instructions selected from this set based on the distribution `P`. When one such breakpoint fires, another breakpoint is inserted on an instruction chosen from this set, again based on `P`. One can consider this as a single logical breakpoint moving from the sampled instruction to a new instruction. To maintain the number of pending logical breakpoints, the algorithm uses the `BPCount` map to distinguish the initial `K` breakpoints from new breakpoints.

3.3 Bias-Free Sampling

Perhaps surprisingly, the algorithm described above is sufficient to sample instructions from `FreqInst` based on `P` irrespective of whether these instructions are hot or cold. Since an instruction is sampled only when a breakpoint fires and these breakpoints are inserted based on `P`, we meet the desired sampling distribution [18].

However, a single breakpoint set at a cold instruction may take a long time to fire. This can arbitrarily reduce the sampling rate achieved by this logical breakpoint. In the worst case, a breakpoint set in dead code will reduce the sampling rate to zero.

The algorithm has several mechanisms to avoid this pitfall and maintain an acceptable sampling rate. First, the algorithm starts by setting `K` logical breakpoints at every instruction. This helps in identifying only those instructions that have executed a few times. In particular, dead code will not be added to `FreqInst`. Second, once a breakpoint is set at an instruction, it will be removed from `FreqInst` till it fires (at which point it is added back to `FreqInst`). This mechanism automatically prunes cold instructions from the set to periodically replenish the number of logical breakpoints. This is similar to DataCollider [18], however, rather than maintaining a constant number of pending logical breakpoints, our algorithm increases the number of logical breakpoints in every period that has a lower number of breakpoint firings than expected by the sampling rate. As these logical breakpoints get "stuck" on cold instructions, the continuous replenishing helps maintain the sampling rate.

4 Implementation

Now, we describe the implementation of LCB in detail. We start by reviewing hardware and operating system support for breakpoints.

4.1 Breakpoint Mechanism

Modern hardware contain a special breakpoint instruction that tells the processor to trap into the operating system. For instance, the x86 architecture provides an `int 3` instruction for this purpose. To set a breakpoint on an instruction, one overwrites the instruction with the breakpoint instruction. The breakpoint instruction is no larger than other instructions in the ISA (in x86, the breakpoint instruction is a single byte), making it possible to set a breakpoint without overwriting other instructions in the binary. When a breakpoint fires, the operating system forwards the interrupt to the process or to the debugger if one is attached. Processing the breakpoint involves removing the breakpoint by writing back the original instruction at the instruction pointer and resuming the program. The breakpoint instruction is designed so that setting and removing a breakpoint can be done atomically in the presence of other threads that might be executing the same instructions. For example, in architectures (such as ARM) that support two-byte breakpoint instructions, all instructions are always two-byte aligned.

4.2 Kernel Support

One of the key goals of LCB is to provide a general capability to set and remove a large number of breakpoints as efficiently as possible. Equally important is to do so without changing the semantics of the monitored programs and associated services such as debuggers. LCB relies on kernel processing for efficient and transparent processing of breakpoints. While most of the functionality of LCB can be implemented as a kernel driver that is loaded early in the boot sequence, we relied on some modifications to the Windows kernel. Another advantage of kernel support is that we can use LCB to sample kernel-mode drivers as well.

4.3 Efficient Processing of Breakpoints

4.3.1 Bypassing the Debugger

When a breakpoint fires, the default behavior of the kernel is to notify the debugger (if attached) or send the interrupt to the process. LCB driver registers itself as a debugger so that it gets a first chance to process the interrupt. Bypassing the regular debugger is crucial for efficiency, as debuggers do not handle well frequent firing of any breakpoints. The LCB driver forwards the interrupt to the debugger or to the process if the breakpoint is not one inserted by LCB.

4.3.2 Handling Shared Modules

Another key design decision of LCB is how to handle shared modules. The code section of modules that are frequently loaded by many processes, such as the C libraries, are loaded in memory once and shared across many processes through appropriate virtual memory mapping. Setting a breakpoint at an instruction in such a shared module can be implemented in one of two ways. The first option is to make the breakpoint common to *all* processes. Thereby, the sampling of the instruction is triggered when any of the processes executes the instruction. Another option is to create a per-process copy of the memory page containing the instruction, causing the loss of memory savings achieved by sharing the module.

The current design of LCB uses the first option for efficiency. In many of our usage scenarios, LCB is turned on for many processes, and the memory bloat that would occur as a result of choosing the second option is unacceptable (as LCB sets breakpoints on all code pages). Moreover, this allows us to extend LCB-based sampling for multiprocess programs. For instance, when measuring code coverage, any of the processes executing a particular C library function is sufficient to cover that function.

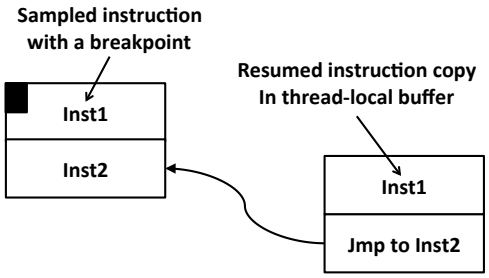

Figure 1: Implementation of multi-shot breakpoints.

4.3.3 Handling Multi-Shot Breakpoints

The functionality LCB provides may require resuming the currently sampled instruction *without* removing the breakpoint. Such multi-shot breakpoints are required, for instance, for sampling the first `K` executions of a basic block. This goes against the default processing of breakpoints, where the breakpoint needs to be removed before resuming the sampled instruction. Once LCB has resumed the execution, it would not get back control unless another breakpoint fires. In the interim, the sampled instruction could have executed many times.

Another option is to use the single-stepping capability of modern architectures. For instance, setting the Trap Flag in the EFLAGS register causes the x86 processor to generate an interrupt after executing a single instruction. Debuggers use this facility to single-step an instruction after removing its breakpoint and on the subsequent interrupt (caused by single-stepping) set the breakpoint on that instruction again. This is safe as debuggers usually block all other threads during this process, however, this generally has unacceptable overhead.

To handle multi-shot breakpoints in native code, LCB creates a copy of the currently sampled instruction in a thread-local buffer, as shown in Figure 1. Immediately after the copy, LCB inserts a jump instruction to transfer control to the instruction after the sampled instruction. When returning from the breakpoint handler, LCB sets the current instruction pointer to the copy of the instruction. This allows the current thread to resume execution without removing the breakpoint. The jump after the copy ensures that control returns to the original program. Note that, this design works even if the sampled instruction is a jump or branch instruction, in which cases the jump instruction of the copy is not executed.

When creating a copy of the instruction, one has to carefully handle instructions that refer to the instruction pointer. For instance, *relative* jump instructions calculate their destination based on the current instruction pointer. Such instructions need to be appropriately modified to retain their semantics when creating a copy. While LCB handles many common cases, it defaults to single-stepping (with all other threads blocked) for other

instructions that refer to the instruction pointer.

The instruction copy in the thread-local buffer is reclaimed by the thread when it ensures that its current instruction pointer and the return values in its stack trace do not point to the copy. For kernel-mode drivers, LCB allocates a processor-local buffer, rather than a thread-local one. This buffer is shared by all contexts that execute on a particular process, including interrupt handlers.

4.4 BfS for Managed Code

Supporting managed code in LCB (such as code written in .NET languages and encoded into the Common Intermediate Language (CIL)) required overcoming several challenges: integration with the Common Language Runtime [2], making sure that the just-in-time (JIT) optimizations do not remove certain breakpoints, and finding and fixing issues in CLR that prohibited setting a large number of breakpoints. In this section, we detail how we overcame these challenges[1].

Initially, we attempted to place breakpoints on every basic block without going through the CLR debugging APIs. However, this did not work, because CLR introspects the managed binary during JITing, and if it finds that the binary has been modified (in this case to include a breakpoint per basic block), it throws an exception and causes the program to crash.

Consequently, we used the CLR debugging APIs to support managed programs in LCB. To do this, we implemented a special debugger within LCB that intercepts the load of each managed module when a program is run and places a breakpoint in each of the program's basic blocks. This debugger's core responsibility is to place breakpoints and track their firing. A program need not be launched using this debugger for LCB to be operational: LCB can be automatically attached to a program at load time.

The second challenge was that the CLR JIT optimizations were modifying the programs by eliminating some basic blocks (e.g., through dead-code elimination) or by moving them around (e.g., through loop-invariant code motion), causing the correspondence between the removed breakpoints and source code to be lost.

To overcome this challenge, we added an option to LCB to disable JIT optimizations and obtain perfect correspondence between the source code and the basic blocks. We are looking into recovering the lost correspondence through program analysis as part of future work, thereby not forcing users of LCB to disable JIT optimizations.

[1] In the process of implementing LCB for managed programs, we discovered and fixed performance bottlenecks and bugs in the CLR. CLR debugging APIs had such issues, because they were not built to be used by a client such as LCB that places a breakpoint in each basic block of a program. The first bug we fixed was a performance issue that caused threads to unnecessarily stall while LCB was removing a breakpoint, due to an incorrect spinlock implementation. The second bug was a subtle correctness issue that occurred only when the number of breakpoints was above 10,000, and JIT optimizations were enabled. We also fixed this issue that was causing the CLR to crash.

4.5 Transparent Breakpoint Processing

For a facility that is commonly used, such as breakpoints, one would not expect the use of breakpoints to change the semantics of programs. While this is generally true, we had to handle several corner cases in order to apply LCB to a large number of programs.

4.5.1 Code Page Permissions

Setting a breakpoint requires write permission to modify the code pages. However, for security purposes, all code pages in Windows are read-only. A straightforward approach is to change the permission to enable writes, then set/clear the breakpoint, and then reset the permission to *readonly*. However, this leaves a window in which another (misbehaving) thread could potentially write to the code page. Under such conditions, the original program would have received an access violation exception while the same program running with LCB would not.

To avoid this, LCB creates an alternate virtual mapping to the same code page with write permissions and uses this mapping to set and clear breakpoints. This mapping is created at a random virtual address to reduce the chances of a wild memory access matching the address. The virtual mapping is cached to amortize the cost of creating the mapping across multiple breakpoints—due to code locality, breakpoints in the same page are likely to fire together.

When sampling kernel-mode drivers, LCB sometimes has the need to process breakpoints at interrupt levels during which it is unable to call virtual-memory-related functions to create/tear down virtual mappings. In such scenarios, LCB uses the copy mechanism for dealing with multi-shot breakpoints described above (§ 4.3.3) to temporarily resume execution without removing a breakpoint. At the same time, LCB queues a deferred procedure call that is later invoked at a lower interrupt level to remove the breakpoint.

Finally, LCB does not set or clear breakpoints on code pages that are writable in order to avoid conflicts with self-generated code.

4.5.2 Making Breakpoints Invisible to the Debugger

Many programs with LCB enabled run with a debugger attached. As described above, LCB hides its breakpoints from the debugger by processing them before the debugger. However, debuggers need to read the code pages,

say in order to disassemble code to display to the user. LCB traps such read requests and provides an alternate view with all its breakpoints removed.

5 LCC Evaluation

In this section, we measure the cost of placing "one-shot" breakpoints on every basic block in an executable using LCB monitors in order to measure code coverage. The resulting code coverage tool is called LCC. LCC represents the leanest instance of LCBẆe first perform a case study on the Z3 automated theorem prover [10] (§5.1), followed by a broader investigation on the SPEC 2006 CPU integer benchmarks (§5.2), then three managed benchmarks from the CLR performance benchmarks (§5.3), and a large scale evaluation on Windows binaries (§5.4).

The code coverage evaluations were performed on an HP Desktop with a 4-core Intel Xeon W3520 and 8 GB of RAM running Windows 8. In our study, we consider three configurations for each application: no code coverage (*base*), the application statically rewritten by the `bbcover` tool (*bbcover*), and the application breakpoint-instrumented by LCC (*lcc*). In order to make the comparison between the tools as fair possible, we use the same basic blocks for LCC breakpoints as identified by the Vulcan framework for the `bbcover` tool. We instruct LCC to insert a breakpoint at the address of the first instruction in each basic block. On the firing of a breakpoint, a bit (in a bitvector) is set to indicate that the basic block has been covered.

5.1 Z3

Z3 is an automated theorem prover written in C++ consisting of 439,927 basic blocks (as measured by Vulcan). Z3 is computationally and memory intensive, having a SAT solver at its core, which is solving an NP-complete problem. We run Z3 on a set of 66 input files that take Z3 anywhere from under 1 second to 150 seconds to process (and many points in between). Each file contains a logic formula over the theory of bit vectors (generated automatically by the SAGE tool [14]) that Z3 attempts to prove satisfiable or unsatisfiable. Z3 reads the input file, performs its computation, and outputs "sat" or "unsat". We test the 64-bit version of the Release build of Z3. For each test file, we run each configuration five times.[2]

We added timers to LCC to measure the cost of setting breakpoints, which comes to about 100 milliseconds to set all 439,927 breakpoints.

[2]We validated that the output of Z3 is the same when run under each code coverage configuration as in the base run and that the coverage computed by LCC is the same as that computed by `bbcover`.

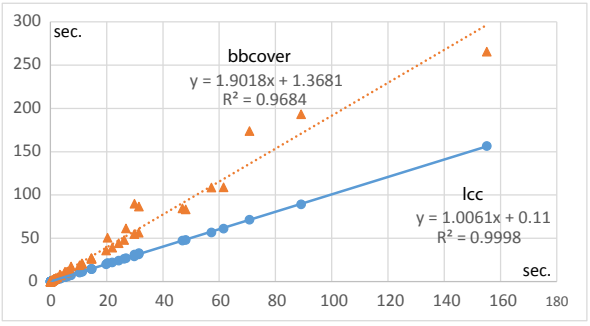

Figure 2: Plot comparing the absolute run-times of coverage tools `bbcover` (triangles, upper line) and LCC (circles, lower line) on the Z3 program (y-axis) against the base configuration (x-axis). Both times are seconds.

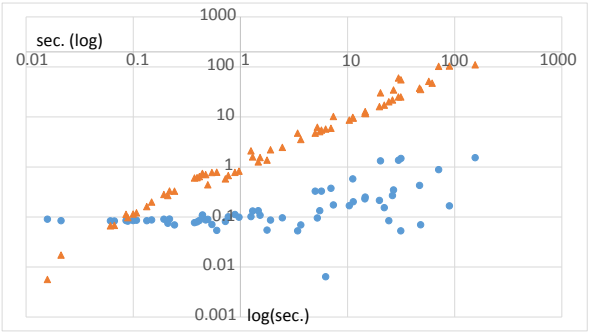

Figure 3: Log-log plot comparing the overhead of `bbcover` (orange triangles) and LCC (blue circles), in seconds over base (y-axis), as a function of base (x-axis).

Figure 2 plots the absolute run time of each test t for the base configuration (the median of 5 runs) against each of the two code coverage configurations and shows the best linear fit for each configuration. We see that the overhead for LCC is less than 1%, with much less perturbation than the overhead of `bbcover`, while the overhead for `bbcover` is around 90% and has outliers.

We would expect that the overhead for LCC be a small constant, independent of the running time of the base execution. In the log-log plot of Figure 3, the x-axis is the run-time in seconds of the base configuration on a test t, while the y-axis represents the overhead (in seconds) of each of the code coverage configurations (over the base configuration) on the same test t. We see the expected linear relationship of the cost of code coverage with respect to execution time for `bbcover`. The plot for LCC shows that the overhead for LCC appears to increase slightly with the base time, although its overhead never exceeds 1.5 seconds.

Figure 4 shows the number of basic blocks (y-axis) covered as a function of run-time (x-axis, log scale). The first thing to notice is that most of the tests cover somewhere between 17,000 and 29,000 basic blocks, a

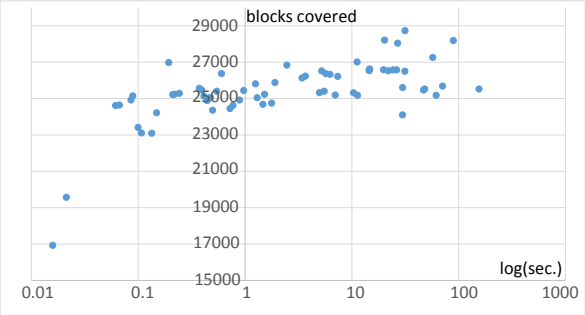

Figure 4: Run-time of base configuration (x-axis, in seconds on log scale) versus number of basic blocks covered, for each of the 66 tests.

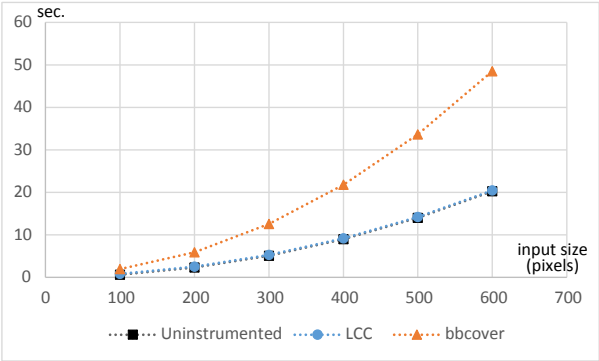

Figure 5: Plot comparing absolute runtimes in seconds (y-axis) of `bbcover` (triangles), LCC (circles), and uninstrumented execution (squares) on the RayTracer program as a function of input size (x-axis).

small fraction of all the basic blocks in Z3. This is not a surprise, as the 66 tests were selected from a suite that exercises just a part of Z3 (the bit vector theory and SAT solver). The two tests that cover less than 21,000 blocks also have the shortest runtimes. Block coverage increases slightly as execution time increases, correlated with the observed increase in runtime overhead for LCC.

5.2 SPEC CPU2006 Integer Benchmarks

To understand the cost of code coverage on a wider set of programs, we integrated both `bbcover` and LCC into the SPEC CPU2006 Monitoring Facility and performed experiments on the SPEC 2006 CPU Integer benchmarks (except for 462.libquantum and 483.xalancbmk, which did not compile successfully on Windows 8).

Table 1 presents the results of the experiments, with one row per benchmark. We ran each benchmark for five iterations using base tuning. The second and third column show the number of basic blocks in a benchmark and the number of tests for that benchmark (each iteration runs all tests once and sums the results). We call out the number of tests because each test is a separate execution of the benchmark, which starts collection of code coverage afresh. Thus, for example, the 403.gcc benchmark has 9 tests and so will result in setting breakpoints 9 times on all 198719 blocks (for one iteration). The columns labeled *base*, *lcc*, and *bbcover* are the median times reported by the `runspec` script (in seconds) of the five runs, for each configuration, respectively, as well as the standard deviation. The overhead of the *lcc* and *bbcover* configurations to the base configuration is reported in the remaining two columns.

The overhead of `bbcover` ranges from a low of 18.67% (429.mcf) to a high of 176.22% (400.perlbench). In general, the slowdown varies quite a bit depending on the benchmark. Our experience with static instrumentation is that the number of the frequently executed basic blocks in the executable is the main determiner of overhead. The overhead of LCC, on the other hand, ranges from 1.4% to 2.18%, showing that LCC achieves low overhead across a range of benchmarks, despite the high cost of breakpoints.

5.3 Managed Code

We evaluated LCB's managed code support using three programs used internally at Microsoft for CLR performance benchmarking: RayTracer, a program that performs ray tracing; BizTalk, a server application used in business process automation; and ClientSimulator, a web client simulation program. We measured the uninstrumented runtimes and coverage measurement overheads for `bbcover` and LCC. All results are averages of five runs.

In Figure 5, we vary the size of the input object RayTracer processes from 100 to 600 pixels to see how the overhead changes with input size. The y-axis shows the absolute runtime. The runtime overhead of LCC is a steady 0.2 seconds corresponding to a maximum of 6% overhead irrespective of input size, whereas the runtime overhead of `bbcover` is proportional to the runtime of RayTracer with a maximum absolute time of 28 seconds and a maximum overhead factor of 3×.

Similar to the native Z3 binary, this experiment shows that for the managed RayTracer binary, LCC's overhead is less than that of `bbcover` and it is independent of the program's runtime behavior.

For BizTalk and ClientSimulator, we used standard workloads of the benchmarks. For Biztalk, LCC incurs 1.1% runtime overhead versus `bbcover`'s 2.0% overhead; for ClientSimulator, LCC incurs 5.8% runtime overhead versus `bbcover`'s 34.7% overhead.

RayTracer has several loops that execute many times, therefore, for this case, the runtime overhead of `bbcover` (which instruments the code) is two orders of magni-

Benchmark	num. of blocks	num. of tests	*base* (sec.)	std. dev.	*lcc* (sec.)	std. dev.	*bbcover* (sec.)	std. dev.	*lcc* overhead	*bbcover* overhead
400.perlbench	68224	3	473.71	0.98	481.98	1.33	1308.49	12.31	1.75%	176.22%
401.bzip2	6667	6	575.02	0.77	584.31	2.57	1108.96	5.73	1.62%	92.86%
403.gcc	198719	9	402.27	0.81	410.55	2.75	765.55	1.32	2.06%	90.31%
429.mcf	5363	1	366.49	0.66	373.00	5.50	434.93	0.99	1.78%	18.67%
445.gobmk	43714	5	530.79	0.74	541.47	0.72	1162.91	0.63	2.01%	119.09%
456.hmmer	15563	2	350.59	1.31	357.65	0.17	446.69	1.78	2.01%	27.41%
458.sjeng	10502	1	629.40	3.04	638.24	1.02	1496.96	3.06	1.40%	137.84%
464.h264ref	24189	3	604.54	0.74	613.95	0.93	1008.73	3.57	1.56%	66.86%
471.omnetpp	47069	1	342.99	0.64	350.47	0.12	641.45	1.97	2.18%	87.01%
473.astar	6534	1	439.59	0.77	446.95	0.59	670.12	4.81	1.67%	52.44%

Table 1: Results of running coverage tools on the SPEC 2006 CPU Integer benchmarks. See text for details.

tude more than LCC's. LCC also has lower overhead for BizTalk and ClientSimulator. We conclude that the managed code support for LCC is efficient.

5.4 Windows Native Binaries

To evaluate the robustness of LCB, we applied LCC to all the native binaries from an internal release of Windows 8. We integrated and ran LCC on a subset of system tests in the standard Windows test environment. The goal of this experiment was to check if LCC can robustly handle a variety of executables, including kernel-mode drivers that are loaded during the operating system boot up. Another goal of this experiment was to ensure that LCC does not introduce test failures either due to implementation bugs or due to the timing perturbation introduced by the firing of breakpoints.

The system tests ran for a total of 4 hours on 17 machines. We repeated the test for different system builds: 32-bit and 64-bit x86 binaries, and ARM binaries. The size of the binaries covered ranges from 70 basic blocks to ~1,000,000. All tests completed successfully with no spurious failures or performance regressions.

To compare coverage, we ran the same tests with the `bbcover` tool. Figure 6 shows the difference in coverage achieved by the two tools. Of the 665 binaries, `bbcover` didn't produce coverage for 45 binaries because its overhead caused those tests to time out, thereby failing them. Therefore, the figure reports the coverage for the remaining 620 binaries. The binaries in the x-axis are ordered by the coverage achieved with `bbcover`.

As the tests are highly timing dependent and involve several boot-reboot cycles, there can be up to 20% difference in coverage across runs. Despite this nondeterminism, Figure 6 shows a clear trend. For all but 40 binaries, LCC reports more coverage than `bbcover`. This increased coverage is due to the fact that tests that time out or fail under `bbcover`, due to problems in relocation

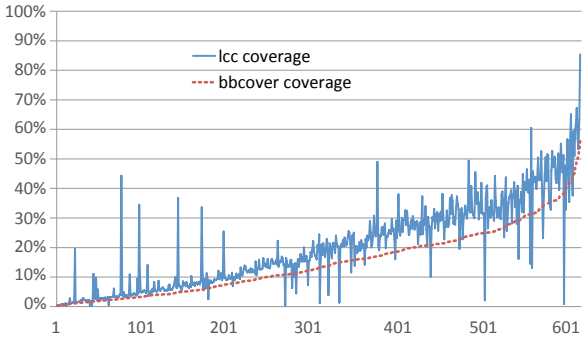

Figure 6: Difference between the coverage reported by LCC vs. `bbcover` (y-axis) for 620 Windows binaries (x-axis).

or excessive runtime, run to completion with LCC. For a small number of cases, LCC reports less coverage than `bbcover` due to test non-determinism.

6 Cold Block Tracing

In this section, we extend LCC to create a simple tracing/logging facility for cold basic blocks, using two different strategies. First, we store the order in which breakpoints fire in a log. This reflects a compressed form of an execution trace where all instances of a basic block beyond its first execution are dropped. The size of this log is bounded by the size of the program. We call this "single-shot logging", since a basic block identifier will appear at most once in the log. Second, we set the `R` parameter to infinity in the BfS algorithm (§3), to periodically refresh breakpoints on all basic blocks. With this option enabled, the size of the log file is proportional to the length of program execution rather than program size. Next, we discuss these two strategies in detail.

Test #	*base* (sec.)	*lcc* (sec.)	*per0.5* (sec.)	*per0.5* overhead	*per5.0* (sec.)	*per5.0* overhead	*lcc* blocks	*per0.5* blocks	*per0.5* Growth	*per5.0* blocks	*per5.0* Growth
21	31.46	32.94	36.29	15.33%	33.10	5.20%	28731	85459	2.97	46356	1.61
52	31.41	31.47	35.55	13.17%	32.72	4.15%	26506	72886	2.75	42359	1.60
12	46.95	47.38	54.39	15.84%	49.02	4.40%	25472	113730	4.46	45437	1.78
57	48.07	48.14	54.76	13.91%	49.69	3.37%	25525	114658	4.49	45882	1.80
43	57.22	56.89	65.44	14.36%	60.13	5.07%	27264	129836	4.76	51316	1.88
65	61.53	61.19	70.74	14.96%	63.90	3.84%	25175	138819	5.51	48581	1.93
14	70.74	71.63	81.27	14.88%	74.22	4.91%	25690	149329	5.81	59058	2.30
62	89.14	89.31	101.96	14.38%	94.07	5.54%	28191	185079	6.57	67408	2.39
29	155.09	156.63	175.82	13.37%	164.88	6.31%	25522	269179	10.55	72864	2.85

Table 2: Periodic logging of Z3 on tests that execute 30 seconds or more.

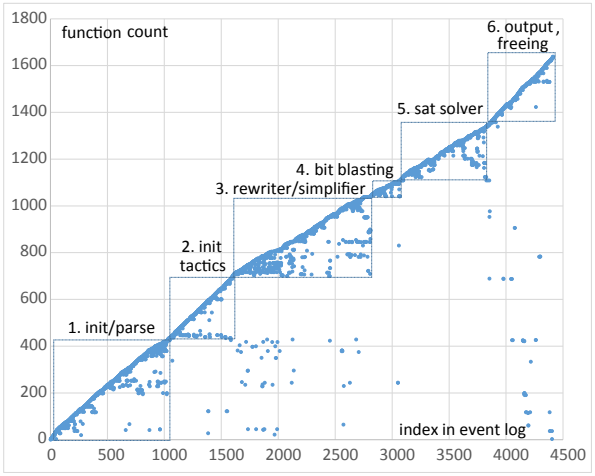

Figure 7: Scatterplot of the coverage log for Z3 test 15 with single-shot code coverage. The base execution of Z3 on test 15 took 20 seconds.

6.1 Single-Shot Code Coverage Logs

The additional cost to log basic block identifiers to a file is negligible for long executions and can be ameliorated by writing the log to a memory buffer, which in the case of single-shot logging is bounded by the size of the program. We give details on the overhead of logging when we consider periodic logging.

We can view a code coverage log as a sequence of events $(i, b(i))$, where i is the index of the event in the log and $b(i)$ is the block id. Such information about the relative ordering of the first execution of each basic block can be useful for identifying phases in a program's execution. Each basic block b has associated symbol information, including an identifier of the function $f(b)$ in which it resides. We assign to each function f a count $c(f)$ which corresponds to the number of unique functions that appear before it in the log.

Figure 7 shows, for a single execution of Z3, a scatter plot that contains one point for each log entry (executed basic block) with index i (x-axis), where the y-axis is the count of the function containing block, namely $c(f(b(i)))$. The scatterplot shows there are about 4500 events in the log. Phases are identified naturally by the pattern of "lower triangles" in which the blocks of a set of functions execute close together temporally. In the plot of the Z3 execution, we have highlighted six phases: (1) initialization of basic Z3 data structures and parsing of the input formula; (2) initialization of Z3's tactics that determine which decision procedures it will use to solve the given formula; (3) general rewriting and simplification; (4) bit blasting of bit-vector formula to propositional logic; (5) the SAT solver; (6) output of information and freeing of data structures.

This simple analysis shows that a one-shot log can be used to naturally identify sets of related functions since it provides an interprocedural view of control-flow behavior. We intend to use this information to identify program portions with performance bottlenecks and to improve job scheduling inside a datacenter [26, 11].

6.2 Periodic Logs

While one-shot code coverage logs are cheap to collect, there are many other (cold) traces the program will execute that will not be observed with the one-shot approach. To collect such information, we can periodically refresh the breakpoints on all basic blocks, as supported by the LCB framework.

Figure 8 shows the scatterplot of the execution log of Z3 run on the same test as in Figure 7, but with breakpoints refreshed every half second. From this plot, we can see that the SAT solver accounts for most of the log. Furthermore, notice that compared to the one-shot log in Figure 7, we see the interplay between the code of the SAT solver in phase 5 and the code of functions executed early on (during phase 1), which represent various commonly used data structures. We also observe more

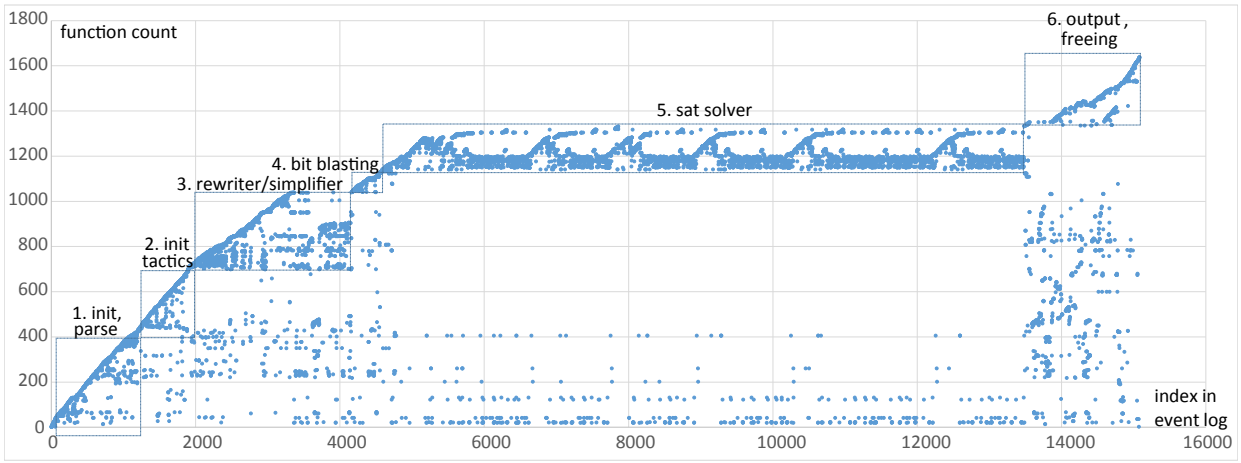

Figure 8: Scatterplot of the coverage log for Z3 test 15 with breakpoints are refreshed every .5 seconds.

activity between the functions in the final phase and the rest of the code (except for the SAT solver).

To evaluate the time and space costs of periodic logging, we selected all the 9 Z3 tests that execute 30 seconds or more in the base configuration. In our first experiment, we refresh all 439,927 breakpoints every half second (configuration *per0.5*), measure the overhead as well as the number of breakpoint firings. Note that the time to set all breakpoints is around 0.1 seconds, expected overheads range from 6 (for a 30 second test) to 30 seconds (for the longest test at around 150 seconds).

Table 2 shows the 9 tests ordered in increasing order of base execution time. As expected, we see that the execution times for *per0.5* increase execution overhead by 4 seconds on the low end (test 21) compared to *base* and 20 seconds on the high end (test 29). While refreshing breakpoints twice a second significantly increases the overhead compared to the single-shot logging of the *lcc* configuration, it is still less expensive than the `bbcover` tool (which doesn't log). Not surprisingly, the size of the periodic log (column "*per0.5* blocks") compared to that of one-shot logging ("*lcc* blocks") is substantial (ranging from a growth of 2.97× to 10.55×).

In the second experiment, we refresh the breakpoints every 5 seconds (configuration *per5.0*), resulting in runtimes closer to that of *lcc* than *per0.5*, and reducing the growth rate of the periodic log substantially.

7 Related Work

Once debuggers gave programmers the ability to set and remove breakpoints on instructions [19], the idea of using a one-shot breakpoint to prove that an instruction was executed (or covered) by a test was born. The rest is just a "simple matter of coding". The first tool we found that uses one-shot breakpoints to collect code coverage is the Coverage Assistant in the IBM Application Testing Collection [4] which mentions "minimal" overheads but does not provide implementation specifics.

DataCollider [18] uses hardware breakpoints to sample memory accesses and detect data races, therefore, it uses a small number of breakpoints at a time (e.g. 4 in x86 processors). Conversely, bias-free sampling requires a large number of breakpoints—linear in the size of the program—to be handled efficiently, which LCB does.

Residual test coverage [25] places coverage probes for deployed software only on statements that have not been covered by pre-deployment testing, but these probes are not removed during execution when they have fired.

Tikir et al.'s work on efficient code coverage [29] uses the dynamic instrumentation capabilities of the DynInst system [7] to add and remove code coverage probes at run-time. While efficient, such approaches suffer from the problem that basic blocks that are smaller than the jump instruction (5 bytes on x86) cannot be monitored without sophisticated fixup of code that branches to the code following the basic block. In addition, special care has to be taken to safely perform the dynamic rewriting of the branch instruction in the presence of concurrent threads. For instance, DynInst temporarily blocks all the threads in the program before removing the instrumentation to ensure that all threads either see the instruction before or after the instrumentation.

The Pin framework [22] provides a virtual machine and trace-based JIT to dynamically rewrite code as it executes, with a code cache to avoid rewriting the same code multiple times. The overhead of Pin without any probes added is around 60% for integer benchmarks. The code cache already provides a form of code coverage as the presence of code in the cache means it has been executed.

Static instrumentation tools like PureCoverage [3], BullseyeCoverage [1], and Gcov [13] statically modify program source code to insert instrumentation that will

be present in the program throughout its lifetime. These tools can also be used to determine infrequently executed code, albeit at the expense of always triggering the instrumentation for frequently-executed code.

THeME [30] is a coverage measurement tool that leverages hardware monitors and static analysis to measure coverage. THeME's average overhead is 5% (with a maximum overhead of up to 30%), however it can determine only up to 90% of the actual coverage. LCB has similar average overhead as THeME, but it fully accurately determines the actual coverage. Furthermore, LCB can be used to obtain multi-shot periodic logs.

Symbolic execution [20, 8] can be used to achieve high coverage in the face of cold code paths. In particular, symbolic execution can explore program paths that remain unexplored after regular testing, to increase coverage. However, symbolic execution is typically costly, and therefore, it is more suited to be used as an in-house testing method. Developers can employ symbolic execution in conjunction with BfS; the latter can be used in the field thanks to its low overhead.

8 Conclusion

Bias-free sampling of basic blocks provides a low overhead way to quickly identify and trace cold code at runtime. Its efficient implementation via breakpoints has numerous advantages over instrumentation-based approaches to monitoring. We demonstrated the application of bias-free sampling to code coverage and its extension to periodic logging, with reasonable overheads and little in the way of optimization.

Acknowledgments

We would like to thank our anonymous reviewers, and Wolfram Schulte, Chandra Prasad, Danny van Velzen, Edouard Bugnion, Jonas Wagner, and Silviu Andrica for their insightful feedback and generous help in building LCB and improving this paper. Baris Kasikci was supported in part by ERC Starting Grant No. 278656.

References

[1] BullseyeCoverage. http://www.bullseye.com/productInfo.html.

[2] Common Language Runtime. http://msdn.microsoft.com/en-us/library/8bs2ecf4%28v=vs.110%29.aspx.

[3] IBM Rational PureCoverage. ftp://ftp.software.ibm.com/software/rational/docs/v2003/purecov.

[4] Application testing collection for mvs/esa and os/390 user's guide. http://publibfp.dhe.ibm.com/cgi-bin/bookmgr/Shelves/atgsh001, January 1997.

[5] T. Ball. The concept of dynamic analysis. In *FSE*, 1999.

[6] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE*, 2006.

[7] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *HPCA*, 14, 2000.

[8] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[9] F. Cristian. Exception handling. In *Dependability of Resilient Computers*, pages 68–97, 1989.

[10] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[11] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. ASPLOS '13, 2013.

[12] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.

[13] GCC coverage testing tool, 2010. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[14] P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3), 2012.

[15] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *OSDI*, 2008.

[16] M. Hauswirth and T. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS*, 2004.

[17] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *FFDO: Feedback-Directed and Dynamic Optimization*, December 2001.

[18] S. B. John Erickson, Madanlal Musuvathi and K. Olynyk. Effective data-race detection for the kernel. In *OSDI*, 2010.

[19] W. H. Josephs. An on-line machine language debugger for os/360. In *Proceedings of the Fall Joint Computer Conference (AFIPS'69)*, 1969.

[20] J. C. King. Symbolic execution and program testing. *Comm. ACM*, 1976.

[21] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. PIN: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[23] P. D. Marinescu and G. Candea. LFI: A practical and general library-level fault injector. In *DSN*, 2009.

[24] B. G. R. Matthew Arnold. A framework for reducing the cost of instrumented code. In *PLDI*. ACM, 2001.

[25] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *ICSE*, 1999.

[26] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. EuroSys '13, 2013.

[27] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[28] J. R. L. Thomas Ball. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4), July 1994.

[29] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *ISSTA*, 2002.

[30] K. Walcott-Justice, J. Mars, and M. L. Soffa. Theme: A system for testing by hardware monitoring events. ISSTA 2012, 2012.