2

Instrumentation Framework

The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong, it usually turns out to be impossible to get at or repair.

–Douglas Adams, Mostly Harmless

This chapter describes the process of going from unmodified application source code to native executables with sampled instrumentation. This process is managed by the *instrumentor*: a software tool whose external behavior mimics that of the native compiler, but that internally applies the instrumentation injection and sampling transformation steps depicted at the top center of Fig. 1.1. Our instrumentor, `sampler-cc`, is implemented as a source-to-source transformation for C using the CIL C front end [48]. Transformed code then proceeds to GCC for native compilation. From the developer's perspective, the `sampler-cc` command behaves exactly like the `gcc` command with a few extra instrumentation-related command line flags.

Section 2.1 presents the basic strategy for managing fair, randomly sampled instrumentation. This sampling transformation is quite general, with potential applications beyond bug hunting. However, bug hunting is the focus of this book, and Sect. 2.2 describes several instrumentation schemes that may be used with the sampling transformation and that we have found to be helpful for bug isolation. Section 2.3 considers performance issues and examines several optimizations that may be applied atop the basic sampling transformation. Section 2.4 describes an adaptive, non-uniformly sampled generalization of the core random sampling model, while Sect. 2.5 closes the chapter with an informal discussion of realistic sampling rates in truly large scale deployments.

2.1 Basic Instrumentation Strategy

This section describes our sampling framework. We begin with sampling of basic blocks and gradually add features until we can describe how to perform sampling for entire programs. Suppose we start with the following C code:

```
{
    check(p != NULL);
    p = p->next;
```

B. Liblit: Cooperative Bug Isolation, LNCS 4440, pp. 7–38, 2007.

```
    check(i < max);
    total += sizes[i];
}
```

Our sampling framework can be configured to sample arbitrary pieces of code, which may be either portions of the original program or instrumentation predicates added separately. Section 2.2 describes several instrumentation schemes that we have found useful. For the remainder of this section, assume that each italicized `check` call is an instrumentation site that has been tagged for sampling. The precise behavior of an instrumentation site is of no concern to the sampling transformation itself. We require only that each such site be removable. That is, performing some `check` calls and skipping others must not affect the user-visible behavior of the program.

2.1.1 Sampling the Bernoulli Way

Suppose that we wish to sample one hundredth of these checks. Maintaining a global counter modulo one hundred is simple, but has the disadvantage of being trivially periodic. If the above fragment were in a loop, for example, one of the checks would execute on every fiftieth iteration while the other would never execute. To avoid this temporal aliasing we wish sampling to be fair and uniformly random: each check should independently have a $1/100$ chance of being sampled each time it occurs. This property is characteristic of a so-called *Bernoulli process*, the most common example of which is repeatedly tossing a coin. We wish to sample based on the outcome of tossing a coin that is biased to come up heads only one time in a hundred.

A naïve approach would be to use a simple random number generator. Suppose `rnd(`n`)` yields a random integer uniformly distributed between 0 and $n-1$. Then the following code gives us fair random sampling at the desired density:

```
{
    if (rnd(100) == 0) check(p != NULL);
    p = p->next;

    if (rnd(100) == 0) check(i < max);
    total += sizes[i];
}
```

This strategy has some practical problems. Random number generation is not free: tossing the coin may be slower than simply doing the check unconditionally. Furthermore, what was previously straight-line code is now dense with branches and joins, which may impede other optimizations.

Sampling is sparse. Each of the conditionals has a $99/100 = 99\%$ chance of not sampling. On any run through this block, there is a $(99/100)^2 \approx 98\%$ chance that both instrumentation sites are skipped. If we determine, upon reaching the top of a basic block, that no site in that block is sampled, then we can branch into fast-path code with all conditionally-guarded checks removed. This design requires two versions of the code: one with sampled instrumentation, one without. It also requires that we can