NIST Technical Note 1826

Estimation of uncertainty in application profiles

David Flater

http://dx.doi.org/10.6028/NIST.TN.1826

National Institute of
Standards and Technology
U.S. Department of Commerce

NIST Technical Note 1826

Estimation of uncertainty in application profiles

David Flater
Software and Systems Division
Information Technology Laboratory

http://dx.doi.org/10.6028/NIST.TN.1826

January 2014
Reprinted with corrections April 2014

U.S. Department of Commerce
Penny Pritzker, Secretary of Commerce

National Institute of Standards and Technology
Patrick D. Gallagher, Under Secretary of Commerce for Standards and Technology and Director

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

National Institute of Standards and Technology Technical Note 1826
Natl. Inst. Stand. Technol. Tech. Note 1826, 39 pages (January 2014)
CODEN: NTNOEF
http://dx.doi.org/10.6028/NIST.TN.1826

Estimation of uncertainty in application profiles

David Flater

January 2014
Reprinted with corrections April 2014

Abstract

Application profiling tools are the instruments used to measure software performance at the function and application levels. The most powerful measurement method available in application profiling tools today is sampling-based profiling, where a potentially unmodified application is interrupted based on some event to collect data on what it was doing when the interrupt occurred. It is well known that sampling introduces statistical uncertainty that must be taken into account when interpreting results; however, factors affecting the variability have not been well-studied. In attempting to validate two previously published analytical estimates, we obtained negative results. Furthermore, we found that the variability is strongly influenced by at least one factor, self-time fragmentation, that cannot be determined from the data yielded by sampling alone. We investigate this and other factors and conclude with recommendations for obtaining valid estimates of uncertainty under the conditions that exist.

1 Introduction

> "The hope is that the progress in hardware will cure all software ills. However, a critical observer may observe that software manages to outgrow hardware in size and sluggishness." [1]

This report is intended for the audience that wishes to measure software performance, whether as a necessary step for improving it or for making performance comparisons in general, rather than relying on newer hardware and more power to cure all performance ills.

The simplest measurements of application performance are nothing more than the total elapsed time or CPU time that is required for an application to run to completion for a given input. At this highest level of granularity, the only difference between hardware and software performance measurements is which factor (the hardware platform or the software application) is controlled and which one is varied. Such measurements can be obtained without recourse to sophisticated tools, but the results are correspondingly less useful if it is necessary to improve the software's performance.

Measuring software performance at a finer granularity is called *profiling*. With measurements scoped to the function level or below, it becomes possible to identify and remedy previously unknown performance bottlenecks in the software with optimal precedence. However, every measurement has an associated uncertainty. If the uncertainty of profile results is estimated incorrectly, the conclusions drawn can be incorrect and the actions taken may be inefficient or do more harm than good.

We performed a series of experiments to better understand this uncertainty and made some interesting findings, including:

- Our results do not support either of two previously published estimates of uncertainty;
- Our results show a pattern of the actual variability being strongly influenced by a different factor that was not previously identified;
- Larger sources of noise that are specific to a particular environment or measuring instrument exist;
- Results can be skewed by seemingly extraneous factors and potentially by sampling bias; and

- Projections made in the process of matching observed results with expected results can introduce uncertainties that exceed those of the original measurements.

The remainder of this paper is organized as follows. Section 2 provides background about profiling and profiling tools. Section 3 provides background about applicable standard uncertainty methods and available estimates of the uncertainty of function-level measurements. Section 4 characterizes the observed non-normalities of results (meaning, they do not adhere to a Gaussian, a.k.a. normal, distribution). Section 5 presents a series of experiments focusing on the effects of several factors on results from different profiling tools. Section 6 presents additional experiments investigating other factors. Section 7 provides a worked example illustrating methods for multiple comparisons and uncertainty resulting from the extrapolation of expected results. Section 8 looks at possible biases. Finally, Section 9 lists findings, recommendations, and future work.

2 Profiling and profiling tools

There are a great many tools for application profiling. However, the selection of a particular platform, programming language, compiler or interpreter can quickly limit the choices. The examples in this report use the following tools and platforms:[1]

- On x86_64 Linux:
  - Perf [2] is included in the Linux kernel source tree and therefore inherits its versioning from the kernel.
  - OProfile [3] includes a new data collection tool, Operf, that is built on the same kernel subsystem as Perf, as well as a "legacy" tool, Opcontrol, that uses a different method to collect data.
  - GNU Gprof [4] is an older profiling tool that is integrated with the GNU Compiler Collection (GCC) [5]. GNU Gprof is very similar to the earlier Berkeley Gprof [6, 7]. Unlike Perf and OProfile, Gprof inserts data collection instrumentation into applications when they are compiled. Data are collected only for objects that were compiled with that instrumentation, which typically excludes the kernel and requires the use of Gprof-enabled versions of standard libraries if complete results are to be obtained.
  - The compiler used under Linux was GCC. Versions were as specified for each experiment.
- On x86_64 Windows 7:
  - VTune Amplifier XE 2013 [8] ("AXE") is included in Intel's developer suites for the Windows and Linux platforms and provides several different data collection drivers with differing characteristics.
  - The compiler used under Windows was Intel Composer XE 2013 [9] ("ICL"), which operated in conjunction with Visual Studio 2012 Express.
- On Android 2.3.7:
  - Android's Debug class [10] is included in its Java library and offers tracing-based profiling of Java apps that support it.

Although application profiling tools can provide a range of different kinds of measurements, the most powerful measurement method available and the focus of this report is sampling-based profiling, where an application is interrupted based on some event to collect data on what it was doing when the interrupt occurred. Using hardware interrupts that are enabled by a separate profiling tool and then serviced by a kernel subsystem, it is possible to collect data about the inner workings of an application without modifying its source code or executables.

Perf, OProfile, Gprof, and AXE all support sampling-based profiling; however, Gprof is limited to user space and a fixed sampling frequency. For AXE, the capabilities vary by data collection driver.

[1] Specific computer hardware and software products are identified in this report to support reproducibility of results. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the products identified are necessarily the best available for the purpose.

The simplest form of analysis for the resulting data simply ranks the functions of an application by the proportion of samples in which they were found to be currently executing. This proportion is called the *self time* of a function because it represents only the time spent executing the function itself, not the time spent executing any subfunctions that it invokes. Those functions that directly consume the most CPU time are readily identified by self time.

Profiling tools support an additional level of data collection to gather information on why those functions were called. From call chains, the *total time* of functions, which includes both the self time and the time spent executing any subfunctions, can be determined. Since the uncertainty issues for total time are not materially different, the examples in this report focus on the self-time results that are simpler to collect.[2]

Perf, OProfile, and certain AXE drivers allow the user to choose numerous different events to drive event-based sampling and to adjust the frequency with which samples are taken. When the chosen event is that a particular hardware timer reaches a specified count, the sampling frequency is derived from the count and the timer's frequency. If a different sort of event is chosen, such as a tracepoint for a specific kernel operation, sampling need not occur at any regular interval and may occur in bursts. For the purposes of this report, it is assumed that the sampling is periodic.

If the sampling frequency at any time becomes too high, data collection tools may lose samples or the overhead of profiling may skew or completely overwhelm the application under test. If it is too low, too few samples will be taken and the results will not be useful for decision-making. The frequency must be tuned to account for the capabilities of the system and the needs of the measurement.

When samples are taken at regular intervals, functions that do not account for much of an application's CPU time may be entirely missed by data collection or may be sampled only occasionally, appearing and disappearing at random from the results of repeated data collection runs. To prevent reports from becoming cluttered with such ephemeral functions, some reporting tools implement adjustable thresholds to filter the reported results. The suppressed results are not necessarily invalid; they are merely deemed insignificant.

Gprof counts function calls with embedded instrumentation, so even functions that do not account for much of an application's CPU time are reliably detected. However, Gprof determines self time by sampling at a fixed frequency, so the functions in question won't necessarily have significant values for self time.

The uncertainties resulting from a variety of factors are potentially important for any application profile. The following sections describe ways of determining that uncertainty.

3 Standard uncertainty methods

Without disrespect to the comprehensive and authoritative references on standard uncertainty that are readily available [12, 13, 14, 15], this section provides a shortened and simplified how-to that is tailored to the audience and the kind of measurement being discussed.

3.1 Empirical estimation of uncertainty from repeated measurements

Estimating uncertainty based on statistical analysis of repeated measurements is known as Type A evaluation of uncertainty [12, §4.2][16]. This empirical approach has the advantage that it requires little advance knowledge about the underlying distribution of measurement results along with the corresponding disadvantage that any such knowledge that one may have is not used to improve the estimate. Holistic benchmarks and self-time measurements are both amenable to this approach if the measurements can be repeated a sufficient number of times.

[2] If one calculates total time simply using the proportion of samples in which a given function either is executing or is found on the stack, then there is no material difference from the uncertainty perspective. Graph-based approaches for estimating total time are less straightforward and may produce divergent results when cycles are present in the call graph. See Ref. [11] for a simple example.

The most frequently used method, which we will refer to as the "original" method, is as follows. It assumes that the samples are independent and identically distributed (i.i.d.), that the distribution has a finite mean and variance, and that the desired summary statistic is the sample mean. Letting X_k represent the n samples (individual, independent measurement results) and $\overline{X}$ the sample mean of those n samples, the standard deviation of the mean is estimated as

$$u = \left(\frac{1}{n\,(n-1)} \sum_{k=1}^{n} \left(X_k - \overline{X} \right)^2 \right)^{\frac{1}{2}}$$

3.2 Analytical estimation of uncertainty

Estimating uncertainty using an analytical model is a case of Type B evaluation of uncertainty [12, §4.3][17].

An analytical model of the intrinsic measurement uncertainty caused by sampling would be useful to support a conclusion that observed variations in performance are actual and not measurement artifacts. We are aware of only two previously published models, which are the documented uncertainty for GNU Gprof and the multinomial model that is applied in Ref. [18]. These two models are presented in the following subsections, followed by a third model that characterizes quantization error.

3.2.1 Gprof result

Gprof's author documented the following result for the "expected error" of self times:

> The actual amount of error can be predicted. For N samples, the *expected* error is the square-root of N. For example, if the sampling period is 0.01 seconds and 'foo''s run-time is 1 second, N is 100 samples (1 second/0.01 seconds), sqrt(N) is 10 samples, so the expected error in 'foo''s run-time is 0.1 seconds (10 × 0.01 seconds), or ten percent of the observed value. Again, if the sampling period is 0.01 seconds and 'bar''s run-time is 100 seconds, N is 10000 samples, sqrt(N) is 100 samples, so the expected error in 'bar''s run-time is 1 second, or one percent of the observed value. It is likely to vary this much *on the average* from one profiling run to the next. (*Sometimes* it will vary more.) [19]

Gprof reports self time in seconds, but the sampling period of 0.01 s, be it actual or derived, is stated in every report:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 78.07      1.27     1.27        8   159.07   159.07  fn2
 20.37      1.60     0.33        2   165.99   165.99  fn1
```

The derivation or source of the result was not cited. Lacking further details, we would assume that it characterizes the behavior of sampling-based profilers in general, not just Gprof. But in Section 5, we will present empirical results that argue for a different analytical model (or none at all) for Gprof as well as other sampling-based profilers.

3.2.2 Multinomial model

Another approach is to use a multinomial distribution as the analytical model as was done in Ref. [18].[3] In this model, each sample is an independent trial, each function is a multinomial outcome, and the probability of each outcome is the self-time proportion of the corresponding function.

If function i has a self-time proportion of p_i and the total number of samples taken in the entire profiling run was n, then according to the definition of the multinomial distribution, the standard deviations of the sample counts (the random variables indicating the number of times outcome i would be observed in n trials) should be

$$\sigma_i = \sqrt{n\, p_i\,(1-p_i)}$$

This analytical result comes with a logical derivation, but the empirical results that we will present in Section 5 do not support this model either.

3.2.3 Quantization error

A sample count is a discrete measure, but the "true" or "ideal" measure of a function's self time generally must be continuous to be practical. Each sample-based measurement of self time therefore is affected by quantization error. Assuming that there is sufficient variation in the data, an empirical estimate of uncertainty from repeated measurements typically would already include the additional uncertainty contributed by this error. However, in an analytical evaluation, quantization error is a component of uncertainty that must either be accounted for or declared to be negligible in the context of the particular measurement.

Consider the idealized scenario in which a function runs for exactly the same amount of time each time it is executed, profiling samples occur with precise periodicity, and the time at which the first profiling sample is taken (call this the "offset") is a uniform random variable with a range from zero to one sampling period. In the simplest case, where all of a function's self time is spent in a single invocation of that function and no subfunctions are called, one can intuit that the quantization error is unbiased:

- Error in the negative direction occurs whenever a "fractional sample" is dropped.
- Error in the positive direction occurs whenever $n+1$ samples are distributed within a function execution of length $n+\epsilon$ sampling periods; for example, a function execution of length 2.2 sampling periods permits samples to be taken at $t = 0.1, 1.1, 2.1$ sampling periods from the time at which the function began executing.

Over a large number of independent measurements that follow this idealized model, the mean sample count for the function in question will converge to the ideal value. Since the error is unbiased, the estimate of its value in a measurement equation is zero. Letting $v \in \Re_{\geq 0}$ be the ideal or true value that is being approximated, for a function that is called only once, the discrete sample count can be described using a single draw from the binomial distribution (B) with $p = v - \lfloor v \rfloor$:

$$N = \lfloor v \rfloor + B(1, v - \lfloor v \rfloor)$$

The variance of N, and therefore of the quantization error, is $p\,(1-p) \leq \frac{1}{4}$ with the maximum occurring at $p = \frac{1}{2}$ (when the execution time of the function is exactly halfway between two multiples of the sampling period).

But consider now what happens when the self time of a function is divided among I separate calls of that function. Making additional simplifying assumptions that the execution time of each invocation is the same and that the offset *with respect to each invocation* is an independent draw from a uniform distribution with a range from zero to one sampling period, the equation for the sample count generalizes to

$$N = I\lfloor v' \rfloor + B(I, v' - \lfloor v' \rfloor)$$

[3] The cited reference refers to the binomial distribution instead of the multinomial distribution and scales results by the factor of n instead of using sample counts directly, but the underlying model is the same.

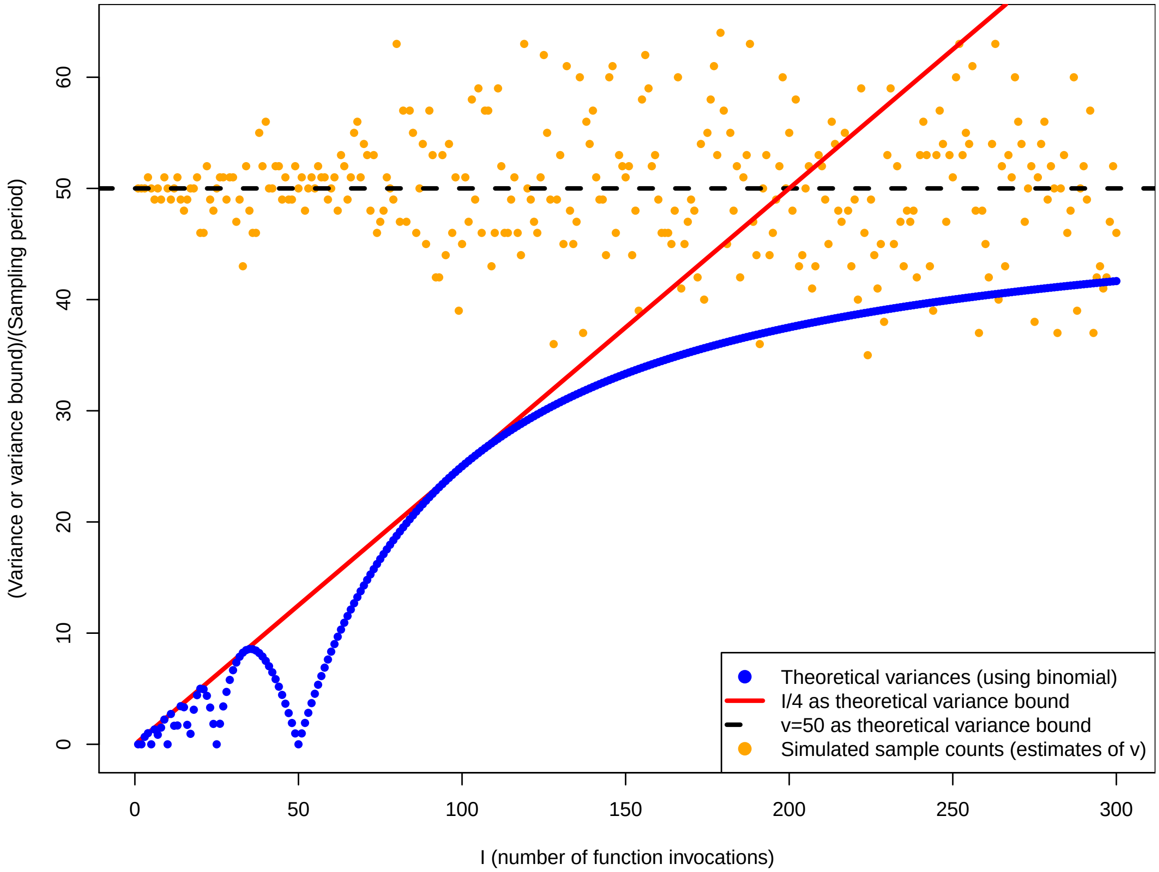

Figure 1: Comparison of theoretical bounds for the variance of quantization error with the theoretical variances and simulated sample estimates for function self time $v = 50$ sampling periods

with $v' = \frac{v}{I}$. If we again use the maximum at $p = \frac{1}{2}$, the upper bound on the variance of N given I is $\frac{I}{4}$.

Fortunately, there is an alternative bound that does not increase linearly with I. If we assume that $I > v$, $\lfloor v' \rfloor = 0$, and the variance of N can be simplified to

$$Iv'\left(1 - v'\right) = I\frac{v}{I}\left(1 - \frac{v}{I}\right) = v\left(1 - \frac{v}{I}\right)$$

which approaches v as I goes to infinity. In the event that $I \leq v$, trivially, $\frac{I}{4} \leq v$, so v also functions as an upper bound in all cases.

With N being our estimate of v, this second bound begins to look like the gprof estimate of $\sqrt{N}$ for the standard deviation, but only as I approaches infinity (i.e., when the self time of a function is highly fragmented). At the other end of the spectrum, the equally valid $\frac{I}{4}$ bound drives the variance to zero as I goes to zero. If the call count I is a known value, $\frac{I}{4}$ provides a tighter bound for sufficiently small I. Of course, if one has a good estimate of v and I is known, one can calculate the theoretical variance of N directly, but this theoretical variance begins fluctuating with increasing sensitivity to its input parameters as I becomes less than v.

Figure 1 shows the relationship between the two bounds, the theoretical variance, and a sample of simulated sample counts (estimates of v) for a range of I from 1 to $6v$.

The empirical results that we will present in Section 5 do show a trend of sample count variance increasing as I increases. However, the observed variances in most cases are much less than predicted by this model, despite the presumed presence of other sources of variability in addition to quantization error.

3.3 Expanded uncertainty and confidence intervals

The purpose of "expanded uncertainty" is to derive a confidence interval for the measurement given a standard deviation.[4] In the original method, this is done based on the assumption that the distribution of the mean of a given number of repeated measurements is approximately normal, a condition which is often asserted by virtue of the Central Limit Theorem [12, §G.2].

Having obtained an estimated standard uncertainty u (either using empirical, analytical, or a mixture of analysis methods to assess the uncertainty from each source), one then calculates the expanded uncertainty $U = ku$ from u and a coverage factor k, which is chosen as needed to produce the desired coverage probability (commonly 95 % or 99 %). The confidence interval is then simply the experimental mean plus or minus U.

In the absence of complications, k can be taken directly from the t-distribution with $n-1$ degrees of freedom [12, §G.3]. In Octave [21], that value of k for a level of confidence C (or $(100 \times C)$ % confidence) is returned by `tinv((1+C)/2, n-1)`. In R [22], the same value is returned by `qt((1+C)/2, n-1)`.

Potential complications:

- If the combined uncertainty is the sum of two or more estimated uncertainty components, one must adjust the degrees of freedom for the t-distribution using the Welch-Sattherthwaite formula [12, §G.4.1].
- When it is necessary to achieve a simultaneous level of confidence for every possible comparison that might be made between different measurements rather than a confidence interval for a single measurement in isolation, the Šidák inequality [23, 24] or one of the comparable methods should be used to determine the higher level of confidence that must be used for the individual measurements in order to obtain the desired confidence for the set of intervals.

An example use of the Šidák inequality appears in Section 7.3. An extended example that applies both the Welch-Sattherthwaite formula and the Šidák inequality to software performance measurements can be found in Ref. [25].

3.4 Numerical determination of confidence intervals

The use of the t-distribution in the previous section introduces an assumption of normality. Measurements of software can often be automated and repeated a sufficient number of times not only to validate an appeal to the Central Limit Theorem, but to make the uncertainty of the mean negligible according to any reasonable evaluation. But when the distribution of results is not normal and the sample size is limited by any factor, the validity of the normal-based estimate becomes difficult to establish, and there is no universally applicable recipe for doing so. For that reason, the "bootstrap" method of estimating uncertainty [26] is gaining currency.

The bootstrap method is more robust and can be used any time that the distribution of measurement values can be reasonably well characterized. In the cited article and in common practice, that distribution is simulated through random sampling with replacement from the original data. Alternative methods can be used, such as fitting a common probability distribution or a derived estimated distribution to the data.

The general purpose of the bootstrap is to obtain approximate confidence intervals. Unlike the original method, the bootstrap method can be applied in essentially the same way for summary statistics other than the mean and can accurately characterize confidence intervals that are asymmetrical around the estimated value. Since the data used to find a confidence interval are generated randomly rather than collected experimentally, there should be no problem getting sufficiently many samples for the estimate to converge even in complex cases, pathological distributions excepted.

[4] Ref. [12] avoids the term *confidence interval* for reasons that are described in [12, §6.2.2], but the term *coverage interval* that is used instead is unfamiliar to many readers and could be misinterpreted to suggest that we mean either a Bayesian *credible interval* or a *statistical tolerance interval.* We believe that the uncertainty intervals used in this document meet the conditions and use the specific definitions that [12, §6.2.2] indicates for *confidence intervals.* According to [12, §2.3.5, §6.2.2] and [20, §2.36], however, if an expanded uncertainty includes uncertainty from sources assessed using Type B methods, then the interval should be termed a *coverage interval* and not a *confidence interval.*

The uncertainty of the mean of n samples would be estimated by randomly generating "many" sets of n samples according to the supplied probability distribution, calculating the means of those sets, and then examining the resulting distribution of generated means. Although one can easily estimate the standard deviation of the mean at that point by taking the standard deviation of the set of generated means, this result is known to converge to [27, Eqn. 5.12]:

$$u' = \left(\frac{1}{n^2} \sum_{k=1}^{n} \left(X_k - \overline{X} \right)^2 \right)^{\frac{1}{2}}$$

which for large n is close to and not better than the original method's estimate. Instead, it is more productive to jump directly to the goal of finding a confidence interval. The location and width of the confidence interval can change even though the standard deviation does not. A probabilistically symmetric[5] confidence interval can be estimated by taking quantiles of the set of generated means (e.g., 2.5 % and 97.5 % for a 95 % confidence interval). For multiple comparisons, the higher level of confidence that was used in the k calculation for the original method is used analogously in choosing the quantiles for the confidence interval in the bootstrap method.

The confidence interval described above is commonly known as the percentile interval. It has greater coverage error than the BC_a and bootstrap-t intervals [28], but compared to those alternatives it is much simpler to explain and implement. The better (and worse) alternatives are explained in textbooks [27] and implemented in the R boot package [29].

The validity of the bootstrap method depends on one's ability to approximate the distribution of measurement values. For many fairly well-behaved distributions, the effort may not be onerous. However, it may happen that by the time an unusual distribution has been adequately characterized through repeated measurements, the uncertainty of the mean of those measurements will already have become negligible. On the assumption that future measurements of the same type will exhibit the same distribution of results, the effort might support a reduction in the amount of data collection for subsequent experiments.

4 Observed non-normalities of self-time results

The following generalizations are based on visual examination of the distributions of results for the self time of functions as reported by multiple application profiling tools.

4.1 Discretization

Trivially, any profiler that relies on sampling has a quantum corresponding to a single sample. Less trivially, any profiler that relies on tracing to determine self times has a quantum corresponding to the resolution of the clock that is used to timestamp tracing events. If this resolution is sufficiently coarse, in can be observed in scatter plots of results, which will stratify along quantum levels.[6]

By definition, discretization guarantees that the distribution of results is not a normal distribution, but it can be a close enough approximation for many purposes. Figure 2 shows an example of an otherwise well-behaved distribution.

[5] Meaning, that the probability of missing the interval is the same on both sides. An alternative is to find the shortest confidence interval, which is not necessarily symmetric [13, §5.3.4].

[6] Some computer systems in which the precision of timestamps significantly exceeded the resolution of the hardware clock have mitigated timing anomalies by artificially incrementing the software clock value each time it was read. Such a practice would add noise to results but would not obscure the strata entirely.

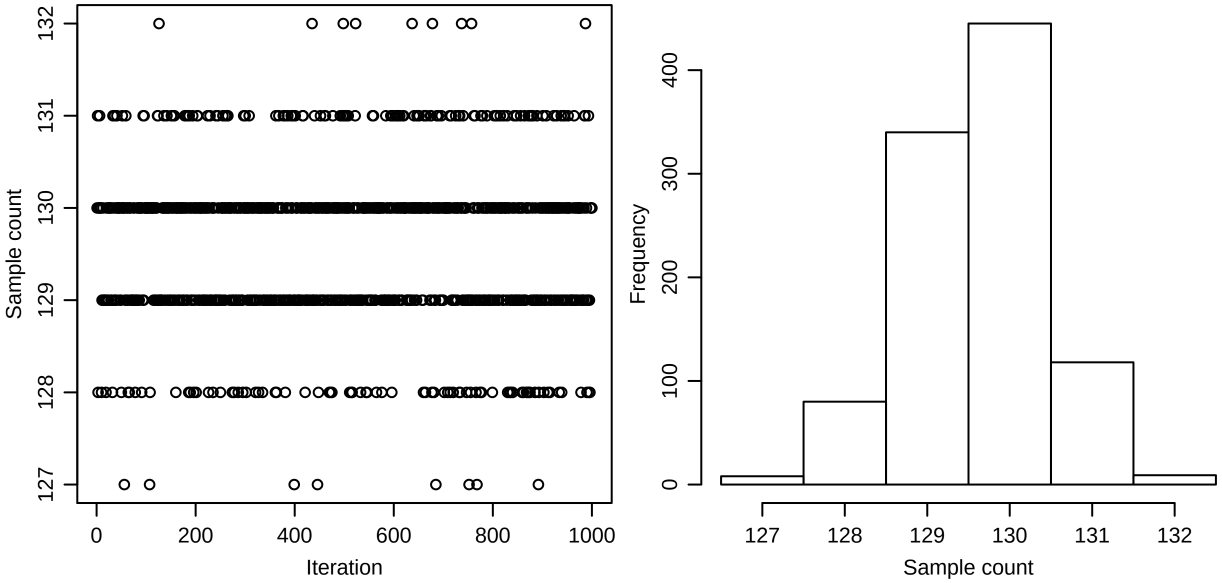

Figure 2: Example showing effect of discretization on bell-shaped distribution of results

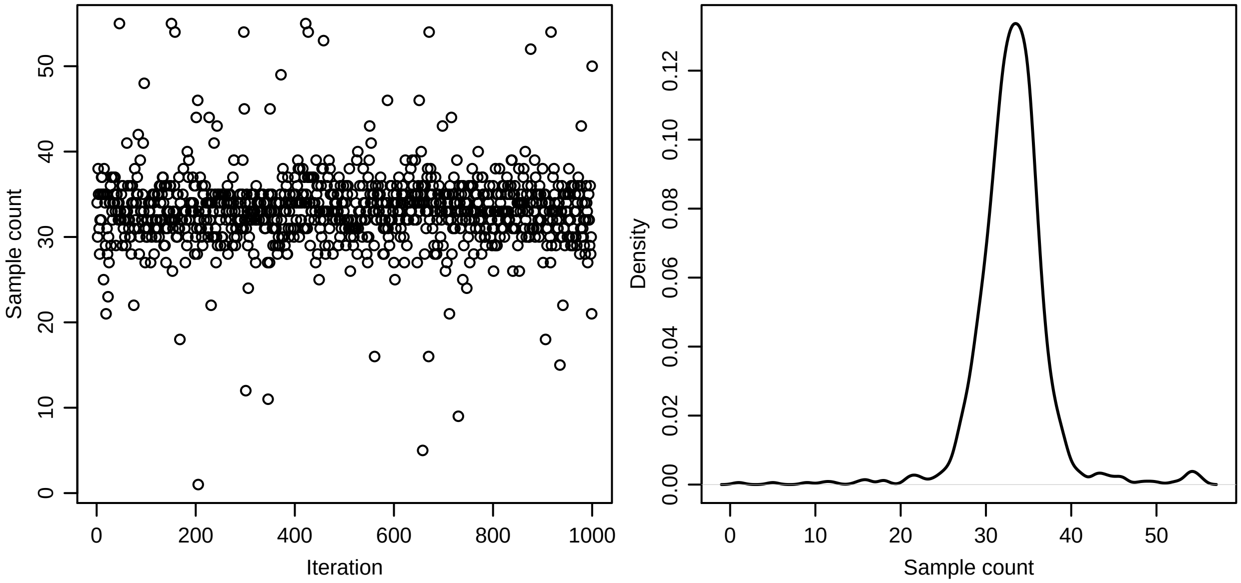

Figure 3: Example of long-tailed distribution of results

4.2 Asymmetry

It is not unusual for distributions of results to show a longer tail on one side. The component of variability that represents actual performance differences is bounded on one side by the performance that is optimal for the system, producing a longer tail going upward. The component of variability that represents sampling noise is bounded at the high end by the number of sampling periods that will fit within the program's execution and at the low end by zero, so it can produce an asymmetry in either direction depending on location. The degree to which these effects or others affect the distribution of results is tool- and test-case-dependent.

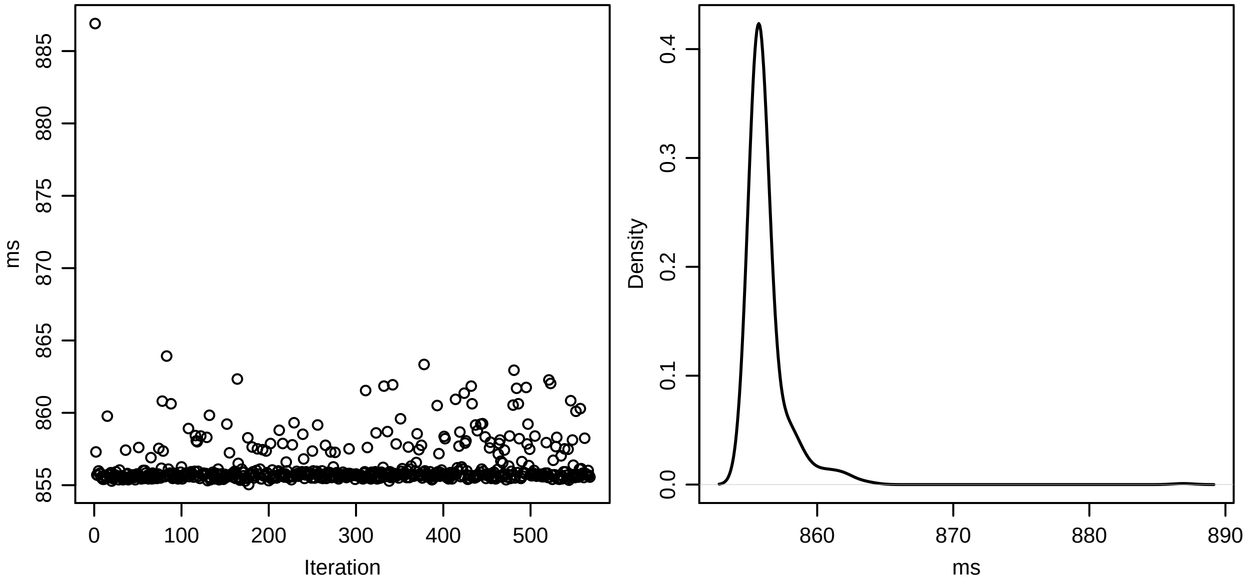

Figure 4: Example of an asymmetrical distribution with a solitary outlier

4.3 Long tails and outliers

When the results in one case display a relatively smooth change in the proportion of points from the center of the distribution to its extremes and a large proportion of points far from the center of the distribution, they may form a long-tailed distribution. Figure 3 shows an example.

However, it sometimes happens that a single, extreme outlier appears in the data, as in the example of Figure 4. If it occurs as the first measurement of a series, it might plausibly be attributed to start-up effects and optionally discarded if a steady-state result is desired. One or two extreme outliers appearing in the middle of data sets are more troubling, since they suggest the presence of a long tail or of one or more additional modes that were inadequately characterized by the sample.

4.4 Multi-modality

Having collected relatively large samples, we have observed so-called outliers clustering around secondary modes in a manner that transient failures would not explain. Figure 5 shows an example. There has been no evidence to support a conclusion that they are erroneous and should be discarded. An assumption that the performance of software in a controlled configuration should be characterized by a better-behaved distribution would be convenient but difficult to justify, given the complexity of the system.

When measurements behave in this manner, the maximum (worst-case) result might be of equal or greater value than the mean. The most informative presentation of results, if enough data have been collected to characterize it, is to plot the probability distribution.

5 Observed effects of sample count, self-time fragmentation, and kernel timer frequency

5.1 Uncertainty methods applied

This subsection describes the methods applied for the experiments in this section and similar ones in subsequent sections.

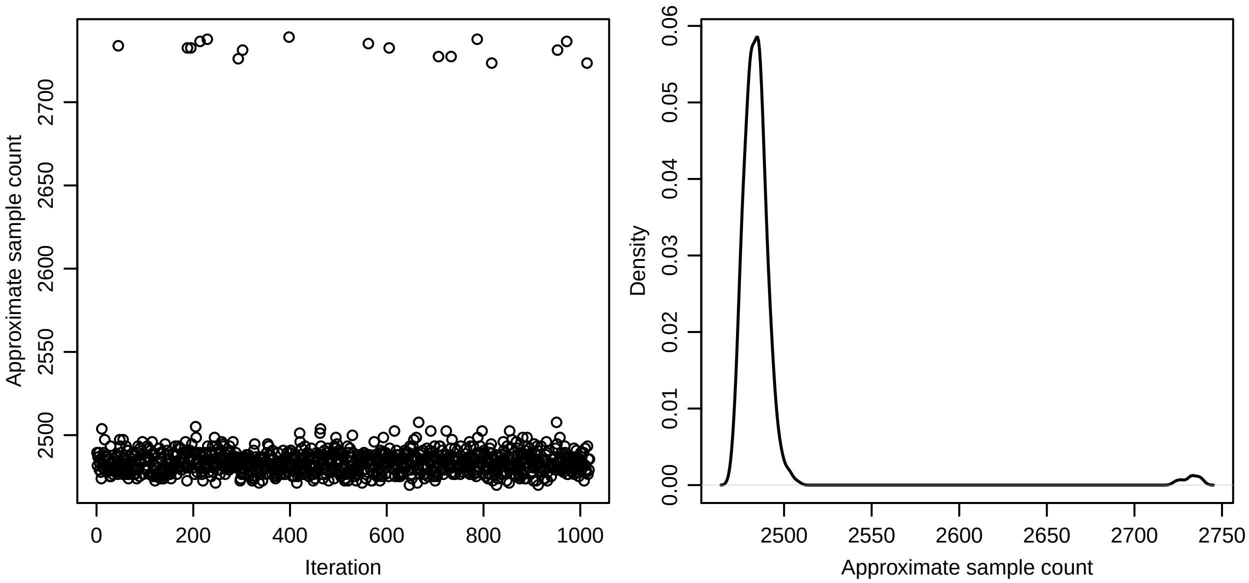

Figure 5: Example of distribution with a low-frequency secondary mode

Consistent with the methods discussed in Section 3, we assume that measurement results are independent and identically distributed (i.i.d.) samples from an unknown distribution.

The unknown distribution may have asymmetry, a long tail, or multiple modes. For the sake of tractability, we assume that any secondary modes in the distributions of results have a high enough frequency that they would appear in our samples with high probability. Without some assumption of this form, we could never be assured of having adequately characterized the distribution with any finite sample however large.

The distributions of measurement results from each experiment were visually inspected for pathologies such as time-dependency (a failure of i.i.d.) and significant outliers. Inspection was performed using scatter plots, kernel density plots, and autocorrelation plots. The kernel density plots were produced using the R density function with smoothing bandwidth of no less than the value of 0.5 samples (half the quantum of the input data). This minimum limit prevented the discretization of sample counts from producing oscillations in the density plots when the default bandwidth selector (based on Silverman's rule [30]) chose a very narrow bandwidth. Similarly, the autocorrelation plots were produced by the R acf function after dithering the input data to mask the discretization.

Next, confidence intervals for the means were calculated using the original method and subsequently validated using the bootstrap method (percentile interval). Despite noted outliers and non-normality, the two methods yielded significantly different confidence intervals for the mean only for certain cases in Section 7, where extreme solitary outliers in a relatively small sample (100) led to a questionable bootstrap.

To test the analytical estimates of uncertainty given in Section 3.2, we also needed confidence intervals for the standard deviations—the uncertainties of the standard uncertainties. The estimated variance of samples from a normal population is known to have a scaled chi-squared distribution; in the original method, the confidence intervals therefore would be derived using quantiles of *that* distribution: [31, Thm. 6.16][32]

$$\sigma^2 \in \left[\frac{(n-1)\, s^2}{\chi^2_{n-1,1-\alpha/2}}, \frac{(n-1)\, s^2}{\chi^2_{n-1,\alpha/2}} \right]$$

However, the confidence interval for the standard deviation is much more sensitive to non-normality than the confidence interval for the mean. In cases where the distributions of measurement results were clearly non-normal, we found that the bootstrap method gave significantly wider confidence intervals than the above equation. In well-behaved cases, the two methods were in material agreement. Therefore, we have consistently used 10^6 iterations of the bootstrap method (percentile interval) to find the confidence intervals of standard deviations.

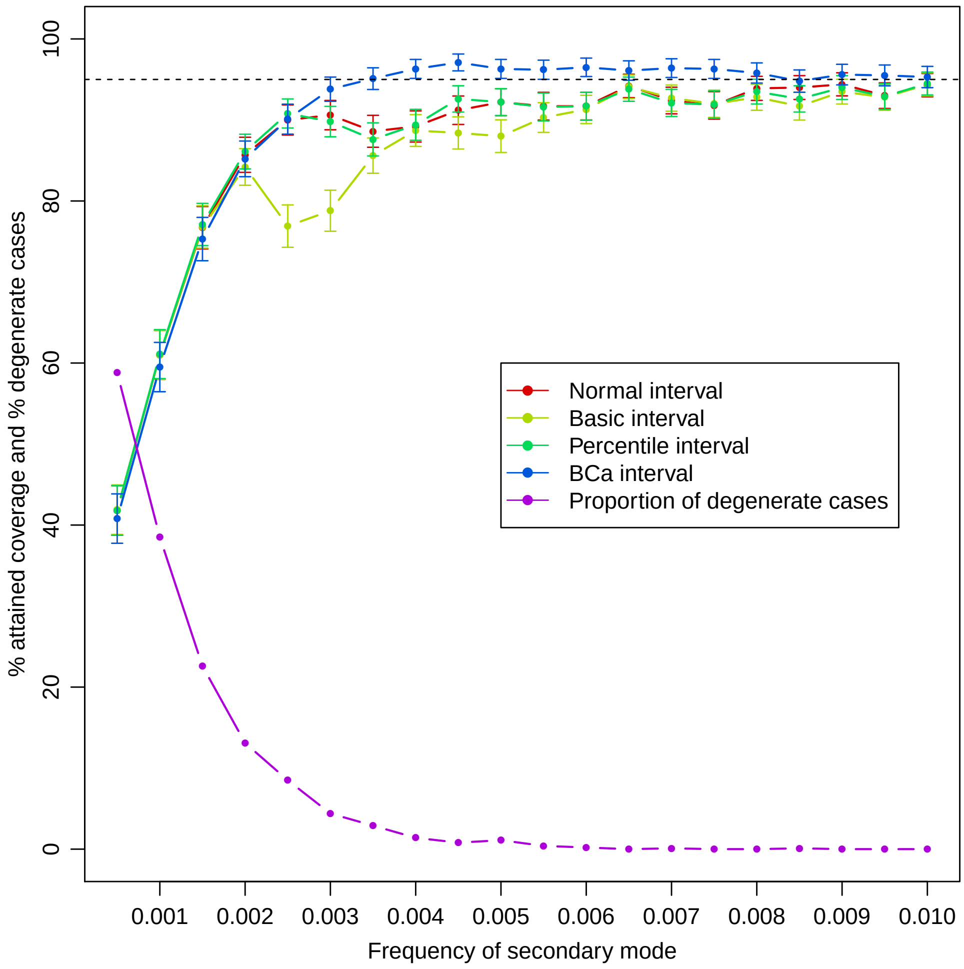

Figure 6: Attained coverage of several bootstrap confidence intervals for 95 % nominal coverage of the standard deviation of a mixture distribution in 10^3 trials with sample size 10^3 and 10^4 bootstrap replicates in each trial

Through simulation experiments on mixture distributions, we found that the coverage attained by the percentile interval for the true standard deviation drops below the specified level when a low-frequency secondary mode is present (see Figure 6). The critical parameter affecting the coverage error (i.e., the uncertainty of the uncertainty of the uncertainty...) is not the sample size but the expected number of "outliers" corresponding to the secondary mode. For a mixture of two normal distributions with standard deviations of 1 and means of 10 and 30, the attained coverage drops below a nominal 95 % as the expected count of outliers drops below 10. Under the same conditions, the BC_a interval [27, Ch. 14] runs on the high side of nominal until the expected count of outliers drops below 3. At that point, however, the attained coverage of both methods falls off a cliff.

When the sample contains not a single value belonging to the secondary mode, it is a degenerate case that ensures the failure of the bootstrap. In the simulation results, the precipitous drop in attained coverage as the expected count of outliers drops below 3 is clearly related to the rising prevalence of degenerate cases. However, a degenerate case would be undetectable in practice without an additional source of information—hence our assumption 6 paragraphs ago that this did not occur.

Although use of the BC_a interval would marginally improve the attained coverage when low-frequency secondary modes appear, we will instead observe that our samples were too small to adequately characterize the secondary modes and use due caution in interpreting the results in those cases.

5.2 Derivation of predicted standard deviations

The analytical models of Section 3.2 produce one result for each profiling run. But in our experiments, we performed many profiling runs to be able to determine the standard deviations empirically. There are consequently too many predictions to present alongside the empirical results.

For the Gprof and quantization error models, we used the mean sample counts rather than the individual sample counts to obtain a single set of predictions for each treatment. For the multinomial model, we pooled the results from the many profiling runs to obtain a single set of p_i values and used the mean of the total sample counts for n.

With respect to the quantization error predictions, since the value of I is known in these experiments and the mean sample counts provide a good estimate of v, the predictions are taken from the theoretical variance (the blue dots in Figure 1) rather than the bounds.

5.3 Gprof

To validate the $\sqrt{N}$ result quoted in Section 3.2.1, we performed the following experiment with Gprof:

- Controlled variables
  - Dell Precision T5400 PC as used in [25], fixed CPU frequency
  - Slackware Linux 14.0 booted in single-user mode
  - Linux kernel version 3.9.2
  - GCC version 4.7.3
  - Test program alternates between 2 calls to `fn1` and 8 calls to `fn2`, two functions with the same expected execution times
  - Program built with –O1 –pg
  - 1000 runs for each combination of independent variables
- Independent variables
  - 2 levels of the value N described above: `fn2` is expected to show approximately 4 times as many samples as `fn1`
  - 15 levels of self-time fragmentation: the test program is parameterized by a function call count to allow approximately the same total workload to be executed using longer functions called fewer times or with shorter functions called more times
  - 2 levels of the kernel configurable CONFIG_HZ: 100 vs. 1000 (CONFIG_NO_HZ was also set in each case)

Each run of the test program produced a value for both levels of N by dividing execution time between two functions, with the main program being overhead. The order of tests progressed upward through each level of self-time fragmentation before starting on the next of the 1000 iterations. The two levels of HZ were run separately after a reboot into the respective kernels.

The test program used the same workload for `fn1` and `fn2` and created the different levels of N by invoking `fn2` four times as frequently as `fn1`. This implies an expectation that any effect of self-time fragmentation would arise due to the shortening execution times of functions as opposed to the absolute number of times that functions were invoked. If the latter case were true, the N factor and the self-time fragmentation factor would not be independent of one another with the test program as implemented, and it would be better to make the same number of calls to functions with different workloads instead of vice-versa.

Regardless, the results for Gprof (Figure 7) indicate that the effect of self-time fragmentation on variability dominates the effects of N and HZ. Neither of the previously published analytical models responds to self-time fragmentation at all, so the observations do not follow their predictions. Although the model of

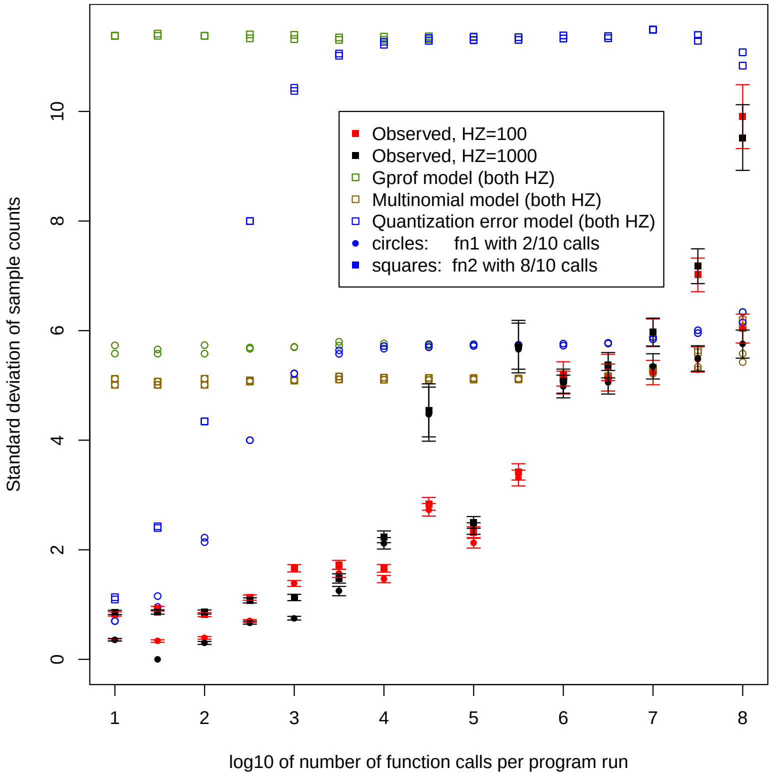

Figure 7: Gprof sample count standard deviations, predicted vs. actual, with 95 % confidence intervals (presumed less than 95 % for $x = 5.5$ with HZ=1000). Confidence intervals are not drawn when the range is vanishingly small.

quantization error in Section 3.2.3 does predict an effect from self-time fragmentation, the observed variability in most cases is still much less than predicted.

It is worth noting that the multinomial predictions would all be exactly the same in an idealized world. The total sample count n would be the same for every run, the observed p_i proportions would be exactly 0.2 and 0.8, and the predicted standard deviation for both functions of interest would always be $0.4\sqrt{n}$.

Figure 8 shows the effects of self-time fragmentation and CONFIG_HZ on the mean sample counts. Some impact from self-time fragmentation is unsurprising since the execution of the program is increasingly burdened with function call overhead. Similarly, CONFIG_HZ is expected to affect the number of interrupts that occur during program execution and may even impact the servicing of Gprof's profiling interrupts.[7]

The distributions of results were mostly well-behaved with discretization being the only divergence from the idealized model of normality that affected every result. Some cases showed asymmetry and/or long tails, but only the case of $10^{5.5}$ function calls with HZ=1000 was significantly affected by a solitary outlier (an anomalously high count for `fn1`, and a correspondingly low count for `fn2`, in iteration 564). An optimistic confidence interval in that case does not impact the findings.

In the case of $10^{4.5}$ function calls and HZ=1000, the distributions showed long tails on both sides. The distribution for `fn1` appeared earlier as the example in Figure 3. An especially interesting event was the occurrence of a sample count of 1 for `fn1`. This measurement differed from the mean by an enormous factor of 33, yet in the context of the long-tailed distribution of 1000 samples it does not stick out as an extreme outlier.

[7] In a message to the kernel-newbies mailing list, Greg Kroah-Hartman wrote, "From userspace's point of view, the kernel HZ value means NOTHING" [33]. But in this case, despite Gprof being limited to userspace profiling, the source of the profiling interrupts one way or another is in the kernel [34].

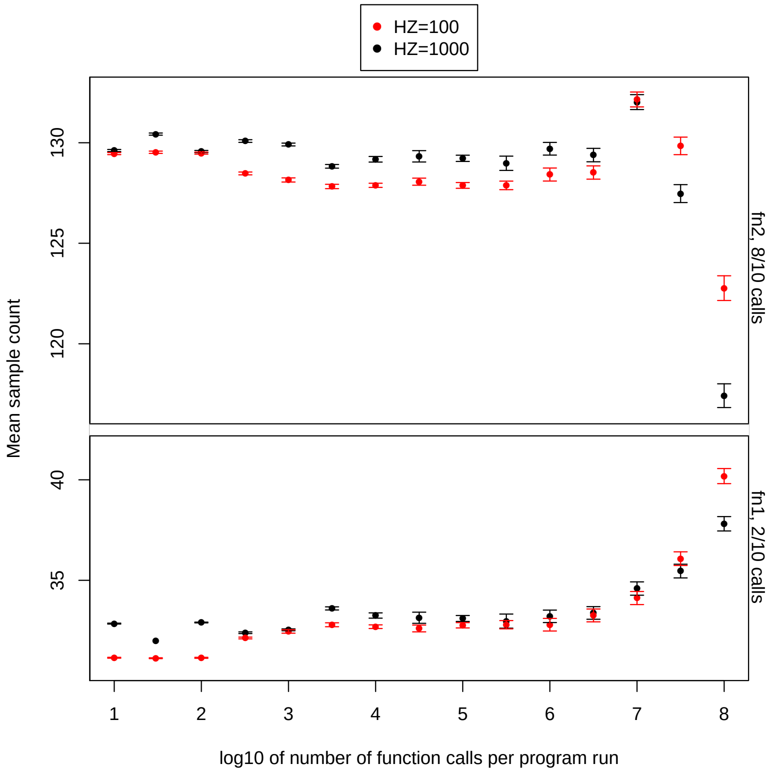

Figure 8: Gprof mean sample counts with 95 % confidence intervals

5.4 Linux Perf

The test of Section 5.3 was repeated with the following changes:

- The test program was compiled with –O2 for Perf instead of –O1 –pg for Gprof.[8]
- Data were collected using perf record –e cpu-cycles –c 1000000
- Call chains were not recorded.

1000 iterations with each combination of HZ and level of self-time fragmentation yielded the results shown in Figure 9 and Figure 10.

In Figure 9, the filled circle representing the observed standard deviation of the sample count for `fn1` is invisible for most combinations with fewer than $10^{7.5}$ function calls per run because the filled square representing the corresponding result for `fn2` is directly on top of it. Similarly, in Figure 10, the results for HZ=1000 are mostly invisible because the results for HZ=100 are directly on top of them, reflecting the negligible difference that HZ made for the mean sample counts.

Self-time fragmentation again had a significant impact on standard deviation, and the HZ value had some impact, although not consistently. The observations again failed to track any of the analytical models, although the multinomial predictions are in the right vicinity for the runs with maximum self-time fragmentation. It can also be said in the models' favor that they usually err on the side of caution for this test case.

The distributions of results were again not normal but were relatively well-behaved. The distributions corresponding to $10^{6.5}$ function calls did not stand out from the others. On the other hand, the results for $10^{1.5}$ and 10^3 at HZ=100 showed significant positive autocorrelation with a weaker pattern of positive autocorrelation appearing between them at 10^2 and $10^{2.5}$. It is far from obvious why such a pattern should appear in only this region of the data!

[8] Inlining of `fn1` and `fn2` was prevented by `noinline` function attributes.

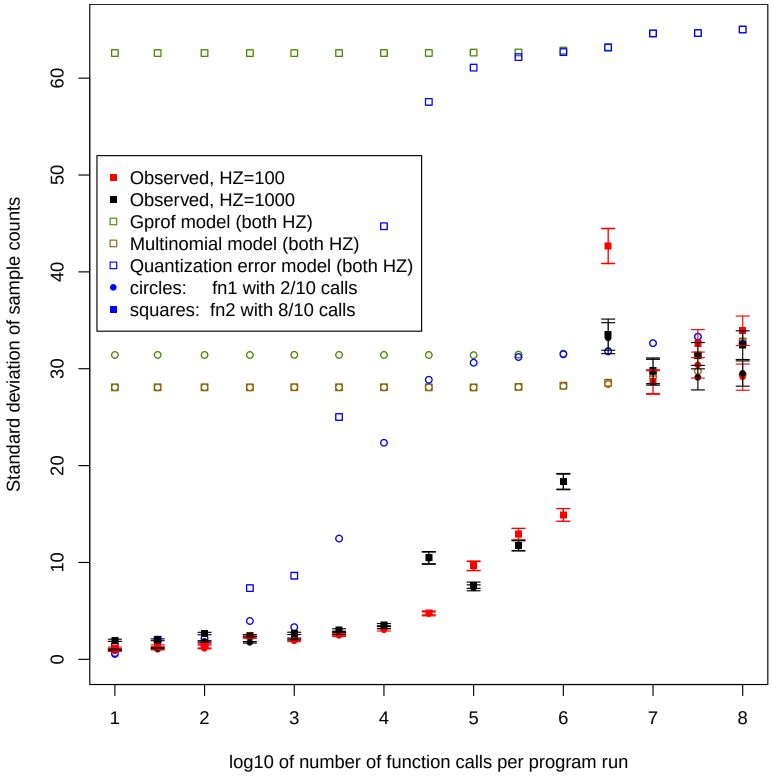

Figure 9: Perf sample count standard deviations, predicted vs. actual, with 95 % confidence intervals

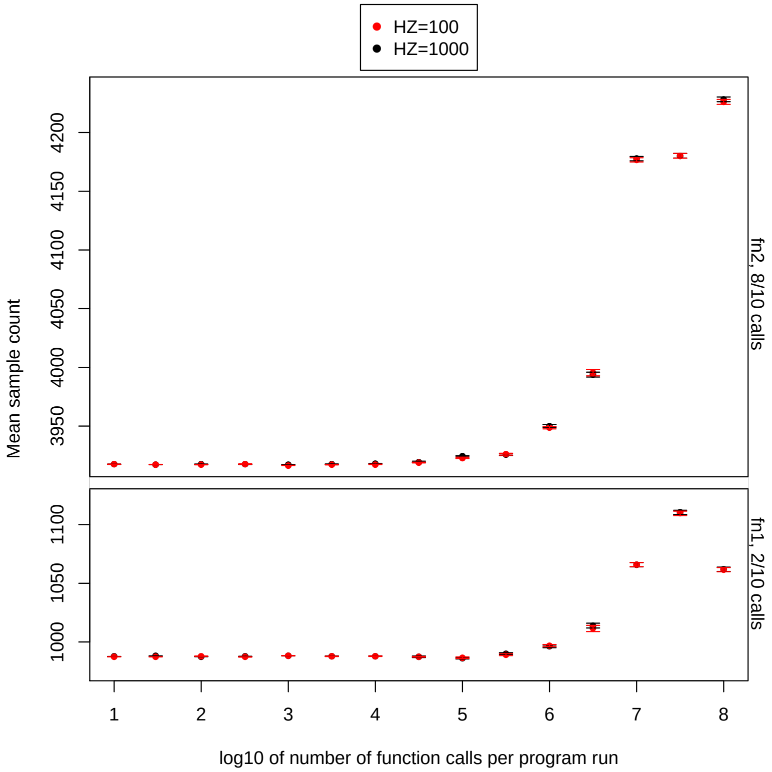

Figure 10: Perf mean sample counts with 95 % confidence intervals

5.5 Windows Amplifier XE

Initially, the test of Section 5.3 was repeated with the following changes:

- Different PC, with a Xeon W3530 4-core, variable-frequency CPU (nominal 2800 MHz, Turbo Boost to 2933 or 3067 MHz).
- The PC was running Windows 7 Enterprise Service Pack 1 64-bit under relatively uncontrolled, centrally-administered desktop conditions, with randomly occurring competing loads (e.g., security scans).
- The test program was compiled with Intel Composer XE 2013 ("ICL") Update 4 at /O2 with debugging enabled.[9] ICL depends on Visual Studio; Visual Studio 2012 Express Update 2 or earlier was installed.
- Data were collected using Intel VTune Amplifier XE 2013 ("AXE") Update 8, amplxe-cl –collect lightweight-hotspots.[10]
- Call chains were not recorded (–knob enable-stack-collection=false).[11]
- The CONFIG_HZ setting does not apply.

Operational issues and interference from administrative software prevented a complete and credible data collection run on this configuration, but as it was, the collected results showed some apparent time-dependency (Figure 11). This was attributed to the interaction of Turbo Boost with temperature—AXE reported the self times as "CPU Time:Self" in seconds, so presumably they would vary with frequency—and when the experiment was repeated (again incompletely) with Turbo Boost, SpeedStep, and C-states control disabled in BIOS setup, the pattern did not reappear. Nevertheless, when the experiment was migrated to a non-centrally-administered Windows 7 laptop to resolve the issues, a similar slowdown failed to reproduce even though Turbo Boost, SpeedStep, and C-states control all remained enabled (Figure 12).

The follow-up experiment on the laptop was different in the following ways:

- Dell Latitude E6330 with a Core i5-3320M CPU (2-core, nominal 2.6 GHz, Turbo Boost to 3.1 or 3.3 GHz)
- Windows 7 Professional Service Pack 1 64-bit with no unnecessary software
- Disconnected from network
- Minor revisions of software (AXE Update 10, ICL Update 5, VS 2012 Express Update 3)

Self times must be expressed in terms of samples counted in order to obtain a predicted result using the $\sqrt{N}$ rule of thumb, but no way was found to export the data from AXE in those terms. Instead, milliseconds were converted to approximate sample counts using the nominal values for CPU frequency and CPU cycles per sample ("Events Per Sample," as AXE reports, for the event CPU_CLK_UNHALTED.THREAD):

$$\begin{aligned} x \text{ samples} &\approx y \text{ ms} \times \frac{1 \text{ s}}{10^3 \text{ ms}} \times \frac{2.6 \times 10^9 \text{ cycles}}{1 \text{ s}} \times \frac{1 \text{ sample}}{2000003 \text{ cycles}} \\ &\approx y \text{ ms} \times \frac{1.3 \text{ samples}}{\text{ms}} \end{aligned}$$

Despite improved control over the follow-up environment, it remained the case that test executions would occasionally fail silently and produce no output. As a hedge, the number of iterations was increased to 1020. At the end of the experiment, 4 of the 15 logs (for 15 levels of self-time fragmentation) contained only 1019

[9] Inlining of `fn1` and `fn2` was prevented by an `auto_inline(off)` pragma.

[10] Operational problems with reporting prevented the use of the –collect-with runsa data collection driver, which would have been more comparable to Perf.

[11] In Update 9 of AXE, the syntax quoted here was deprecated. The profiling mode previously known as –collect lightweight-hotspots –knob enable-stack-collection=false was renamed to –collect advanced-hotspots –knob collection-detail=hotspots-sampling.

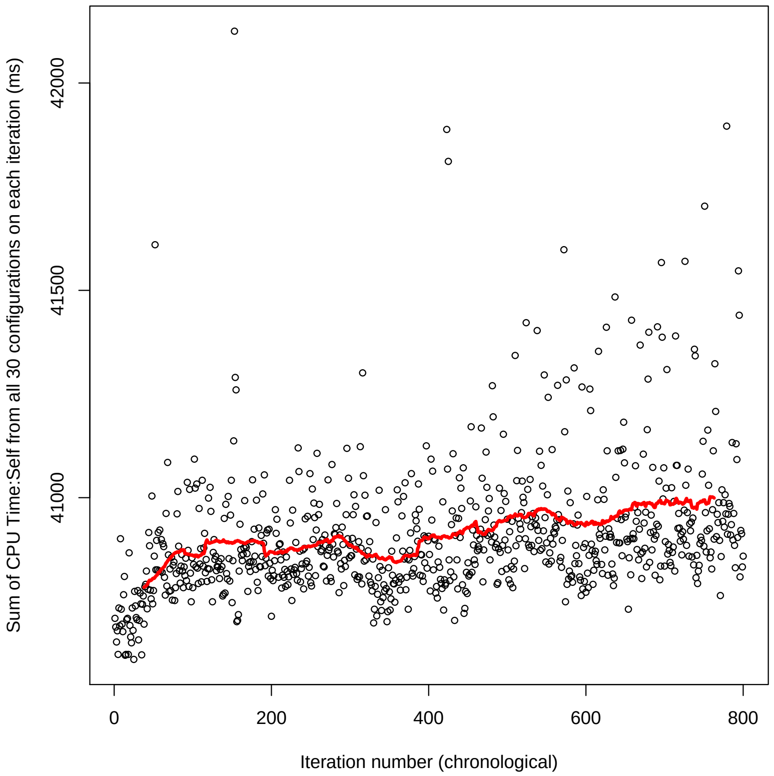

Figure 11: Sum of time series from original AXE experiment showing an apparent slowdown over time with 75-point moving mean

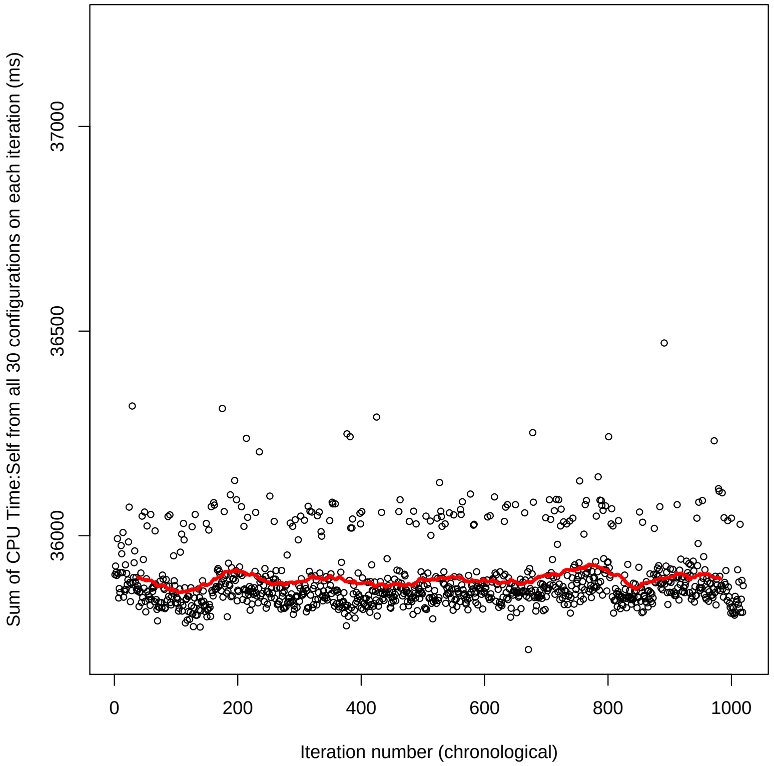

Figure 12: Sum of time series from follow-up AXE experiment showing minimal slowdown over time with 75-point moving mean, vertical axis scaled to match previous figure

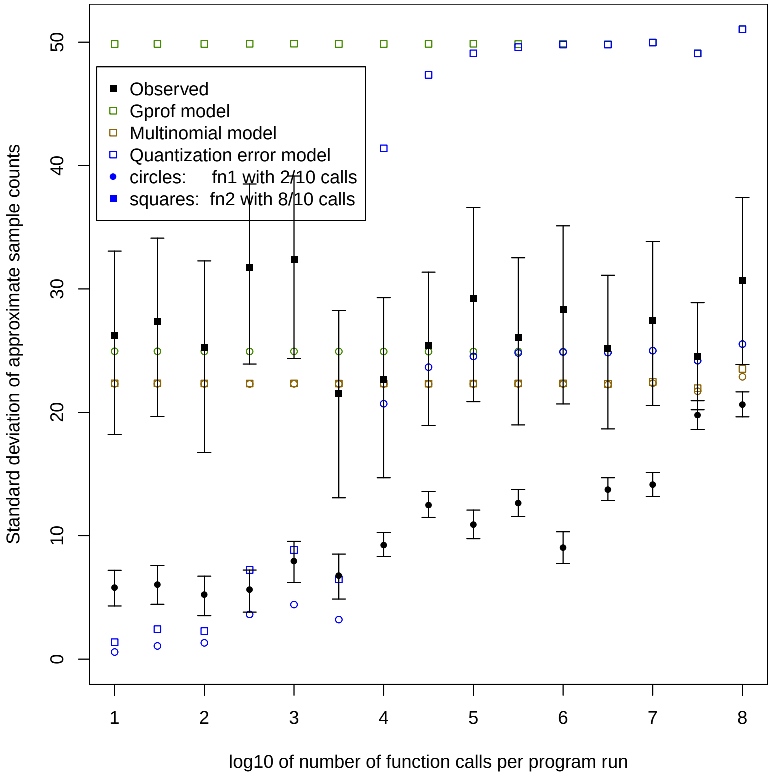

Figure 13: AXE approximated sample count standard deviations, predicted vs. actual, with presumed less than 95 % confidence intervals

samples. For the combined time series shown in Figure 12, therefore, some of the data are time-shifted by one iteration.

Many of the distributions of results this time exhibited a low-frequency secondary mode located above the main cluster at a ratio close to 1.1, which corresponds to no integral adjustment of the CPU multiplier from any of the three cited CPU frequencies. An example appeared in Figure 5. Some results showed a long tail going upward instead, and some had outliers below the main cluster. Because of the low-frequency secondary modes and outliers, the confidence intervals for standard deviations in this experiment, shown in Figure 13, are presumed invalid.

Nevertheless, the previously noted effect of self-time fragmentation on variability can be observed using a more robust measure, the interquartile range (IQR). In Figure 14, the standard deviations (SD) are shown in comparison with the interquartile ranges divided by the constant factor 1.349, which would approximate the SD if the results were normally distributed [35]. The effect of self-time fragmentation is obscured by the outlier-induced "noise floor" in the standard deviations for `fn2`, but remains apparent in the interquartile ranges.

The still-valid mean results are shown in Figure 15.

5.6 Android Debug class (Java)

For the last version of this experiment, we made radical changes:

- From desktop platforms to a mobile platform (Droid X phone running Android 2.3.7);
- From Intel CPUs to an ARM-based CPU (TI OMAP3630-1000, 1 GHz ARM Cortex-A8 [36]);
- From C language to Java for the test program;

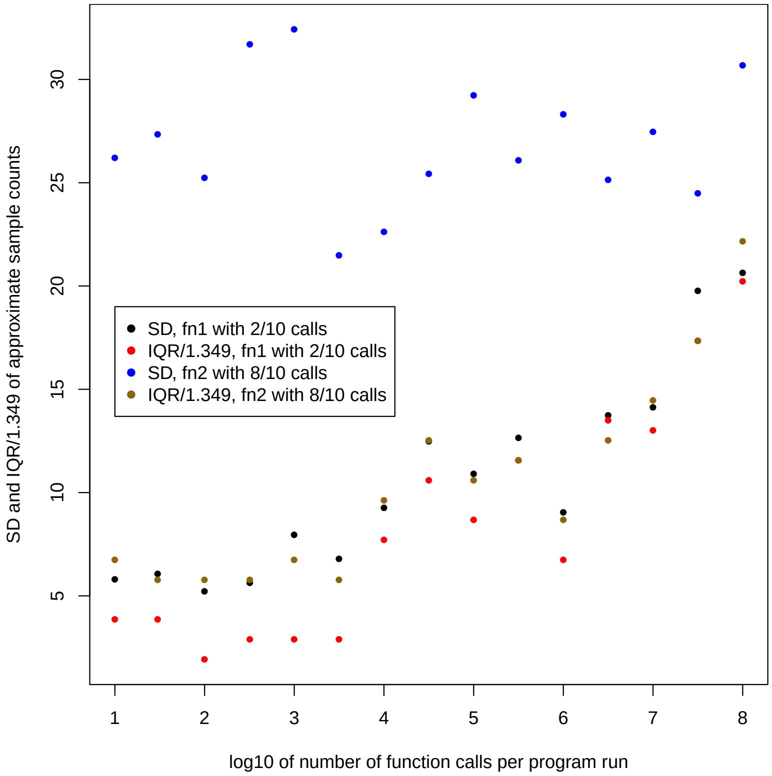

Figure 14: AXE approximated sample count standard deviations and interquartile ranges divided by 1.349

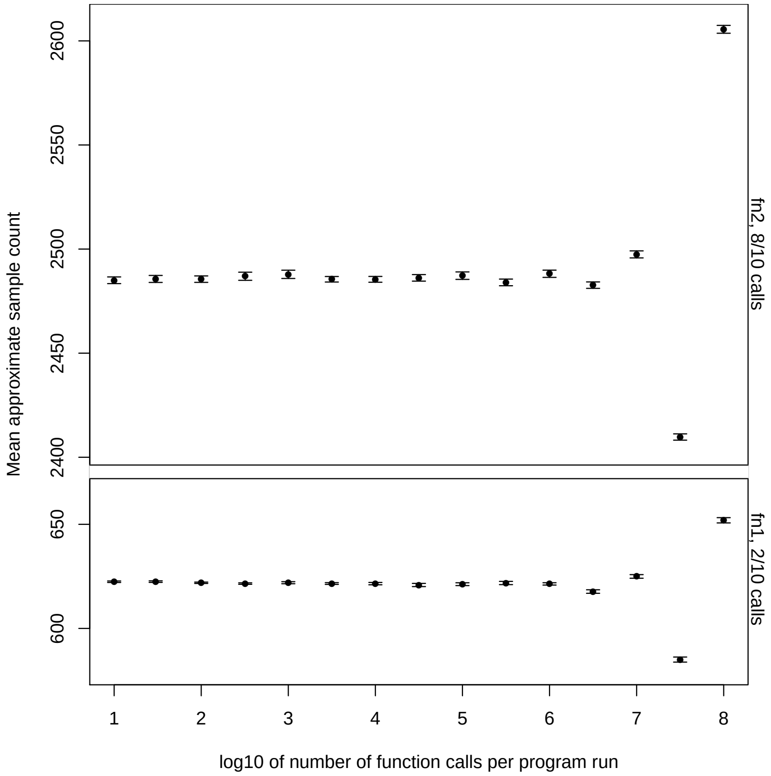

Figure 15: AXE mean approximate sample counts with 95 % confidence intervals

- From a sampling-based measurement to a tracing-based measurement using Android's Debug Java class [10] and the reporting tool dmtracedump [37]. Self time is determined from the timestamps of traced events for function entry and exit.

The experiment was run with the phone connected to AC power and in airplane mode. The test application was launched repeatedly by a version of RTD [38] that was customized for this experiment. All traces were accumulated on the phone's sdcard and offloaded for analysis after data collection was complete.

Both time and space constraints compelled us to reduce the scope of the test. The maximum number of function calls per run was reduced by a factor of 100, the total workload by a factor of 200. Since every function entry and exit is traced by Debug, the size of the tracing data expanded exponentially as the number of function calls did, reaching 18 MB for a run having 10^6 function calls. The number of completed iterations was limited to 569 by the amount of space available on the phone's 16 GB sdcard. One trace file from iteration 380 was unrecoverable because of an I/O error, so all of the traces for that iteration were skipped, leaving a final sample size of 568 for each level of self-time fragmentation.

The results, shown in Figure 16 and Figure 17, are dominated by an escalation of the reported self times that begins when self-time fragmentation reaches 10^5 function calls per program run. Presumably, this is related to the tracing overhead escalating in proportion to the number of function calls—arguably still an effect of self-time fragmentation, but an indirect one. The distributions of results in these cases also show a low-frequency mode located below the main cluster.

The results at the low end (10 function calls) showed an extreme outlier corresponding to the very first execution of the test application for `fn1` (shown in Figure 4) and a bimodal distribution and a solitary outlier in the middle of the run for `fn2`. Thus, the confidence intervals for standard deviations are presumed invalid for this case and the high-end cases.

For the intermediate range from $10^{1.5}$ to $10^{4.5}$ function calls per run, the distributions of results were well-behaved. Examining only this intermediate range, the standard deviation is consistently higher for `fn2` than for `fn1`. As self-time fragmentation increases, a relatively minor upward trend in standard deviations becomes evident before the escalation of the means begins to affect the standard deviation to a much greater extent. However, apart from that relative comparison, the significance of the upward trend in the intermediate range has no objective measure. Any comparison of the variability with that of sampling-based measurements could only be done in terms of time, for a given sampling frequency, and the analytical models for sampling-based measurements are not applicable.

6 Other factors

6.1 Sampling frequency

The sampling frequency is not adjustable for Gprof or for AXE's lightweight-hotspots driver as far as we can tell. The following results were obtained using Linux Perf. The experiment was similar to that of Section 5.4, with the following changes:

- CONFIG_HZ was fixed at 1000.
- The number of function calls was fixed at 1000.
- Iterations were reduced to 100 per configuration.
- The event count for profiling (which has a reciprocal relationship with sampling frequency) became an independent variable with 10 levels.

Figure 18 shows that measured self-times remained proportional throughout the range of feasible sampling frequencies, from extremely low up to just below the point where the event rate was apparently throttled.

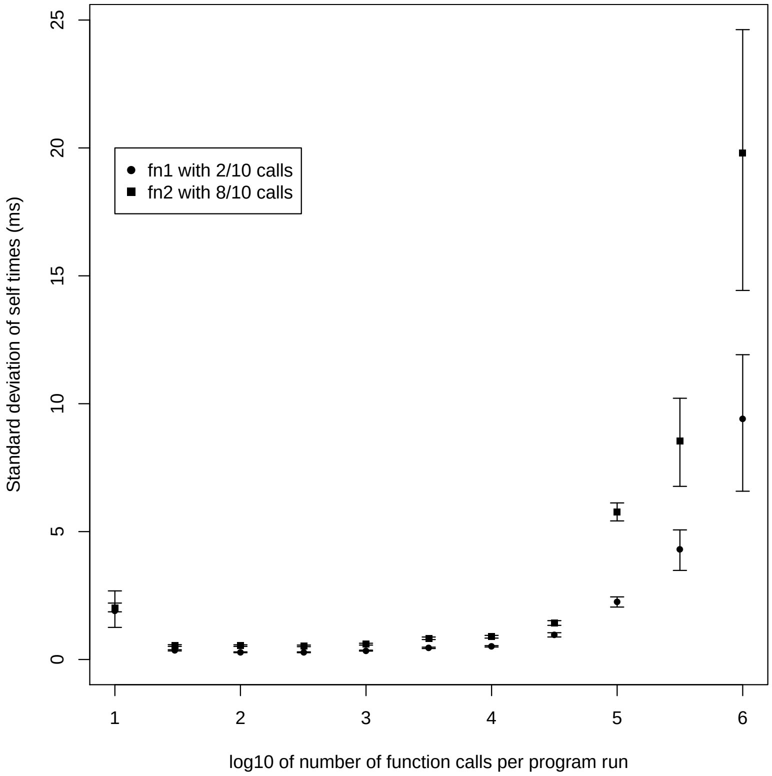

Figure 16: Standard deviations of Android reported self times with 95 % confidence intervals for $1 < x < 5$, presumed less than 95 % for the other cases

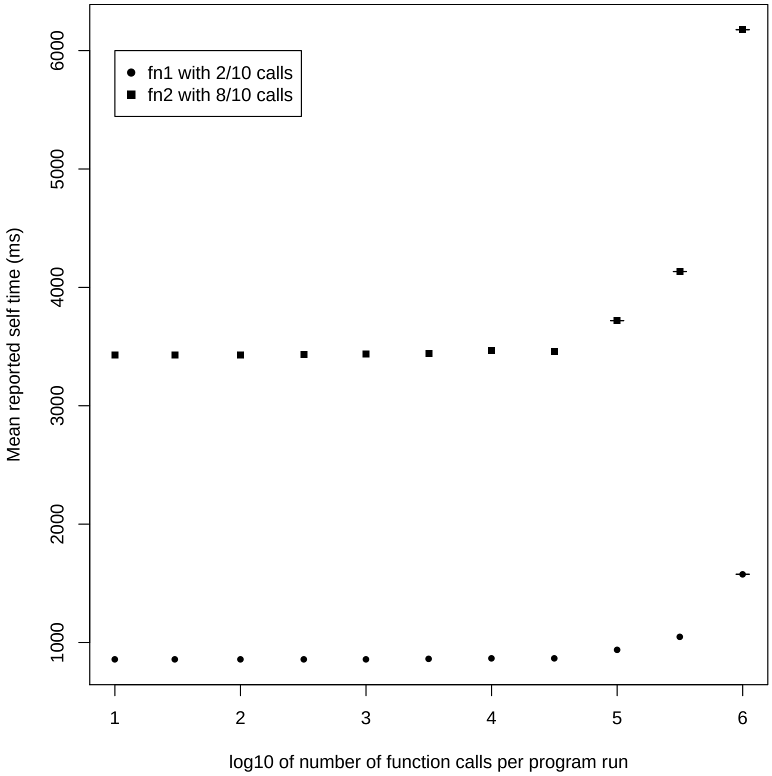

Figure 17: Android mean reported self times with 95 % confidence intervals (when not vanishingly small)

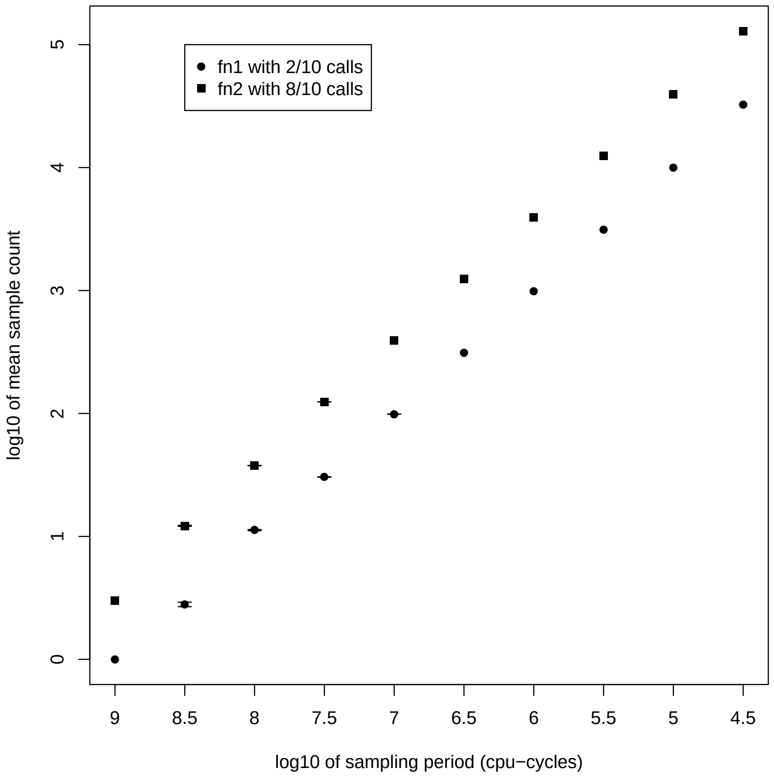

Figure 18: Effect of sampling frequency on mean sample count for Perf with 95 % confidence intervals for the means

Figure 19 and Figure 20 show the effect of sampling frequency on the standard deviation of the self-time measurements in absolute and relative terms respectively. Although the standard deviation does rise as sample counts become large, in relative terms it approaches zero. At low frequencies, the bounded quantization error (see Section 3.2.3) becomes relatively significant.

The distributions of results were well-behaved, though not normal. The smaller sample sizes led to more fluctuations in the density curves. Positive autocorrelation reared its ugly head again as the sampling period reached 10^8 cycles and the sampling frequency became very low. For `fn1` at $10^{8.5}$ cycles, there were only two distinct sample count values—3 and 1—and the 10 occurrences of 1 came in runs of length 7 and 3.

6.2 Competing loads

To test whether competing loads would impact the variability of results, experiments were performed similar to those of Section 5.3 and Section 5.4, with the following changes:

- CONFIG_HZ was fixed at 1000.
- Iterations were reduced to 100 per configuration.
- The number of function calls was reduced to 2 levels, at 10 and 10^6.
- Two new independent variables were added:
  - 2 levels of kernel support for symmetric multiprocessing (SMP off, SMP on). For SMP off, the kernel was booted with the `nosmp` parameter.
  - 2 levels of competing load (no load, with load). The command `dd if=/dev/urandom of=/dev/null` was used as the competing load.

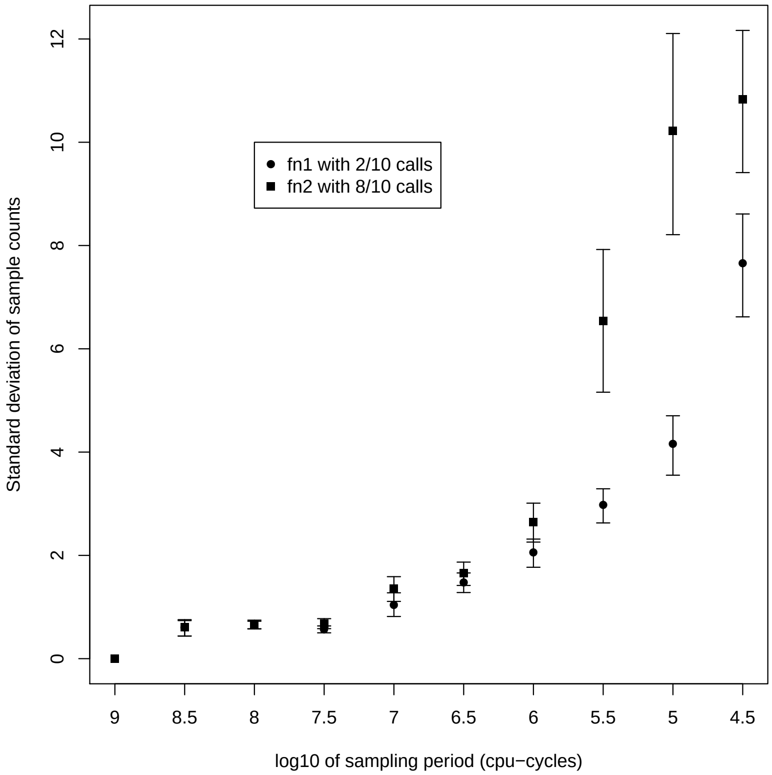

Figure 19: Effect of sampling frequency on sample count standard deviation for Perf, with 95 % confidence intervals

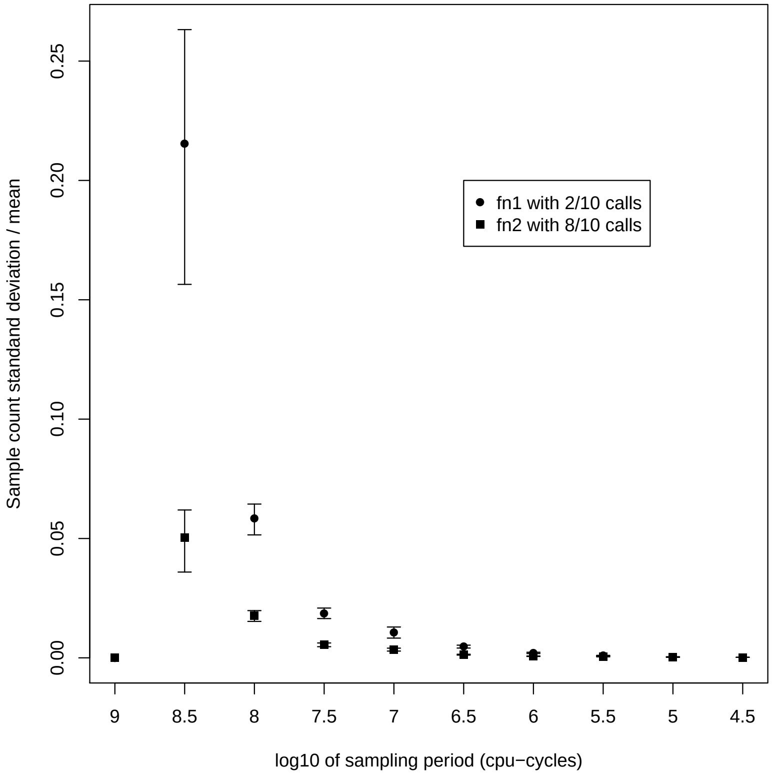

Figure 20: Id., standard deviation relative to mean sample count

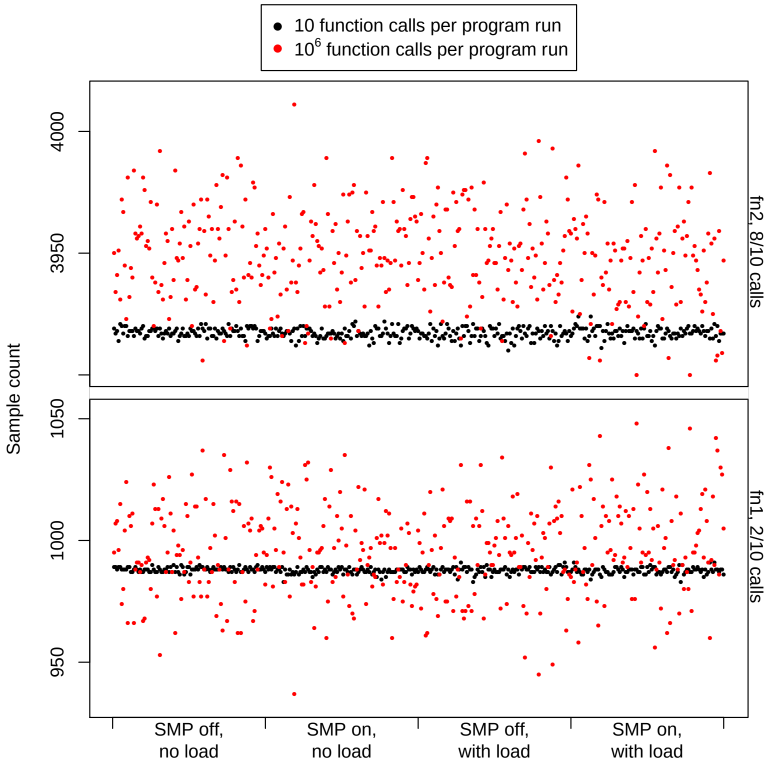

Figure 21: Perf sample count variability with competing loads

Figure 21 and Figure 22 show the results for Perf and Gprof respectively. Consistent with earlier results, self-time fragmentation correlated with an obvious increase in sample count variability; also, the higher sample count for `fn2` correlated with a slight increase in variability that is apparent in the black dots (10 function calls per run); but the presence of a competing load, with or without SMP, made relatively little difference.

For AXE under Windows 7, the experiment was adjusted as follows:

- CONFIG_HZ did not apply.
- To reduce effort, the new independent variables were reduced to two configurations: SMP on with no competing load and SMP off with a competing load. The SMP off condition was created by disabling multi-core support and Hyper-Threading in BIOS setup. The competing load was generated with IntelBurnTest v. 2.54 [39] running at "standard" level.
- Iterations were increased to 205 per configuration.
- Other conditions were as in the follow-up experiment in Section 5.5.

No data were lost this time. As shown in Figure 23, the results for AXE were quite different than those for Perf and Gprof. The competing load had a drastic effect on both the location and dispersion of measurement values and was different depending on the self-time fragmentation factor.

In preliminary experiments for Ref. [25], running multiple copies of a benchmark simultaneously as a competing load on a single CPU had an obvious impact. That benchmark differed from the Perf and Gprof experiments above both in the measurement instrument (high-level Posix timers instead of Perf events) and in the workload (memory-write intensive instead of arithmetic-logic intensive).

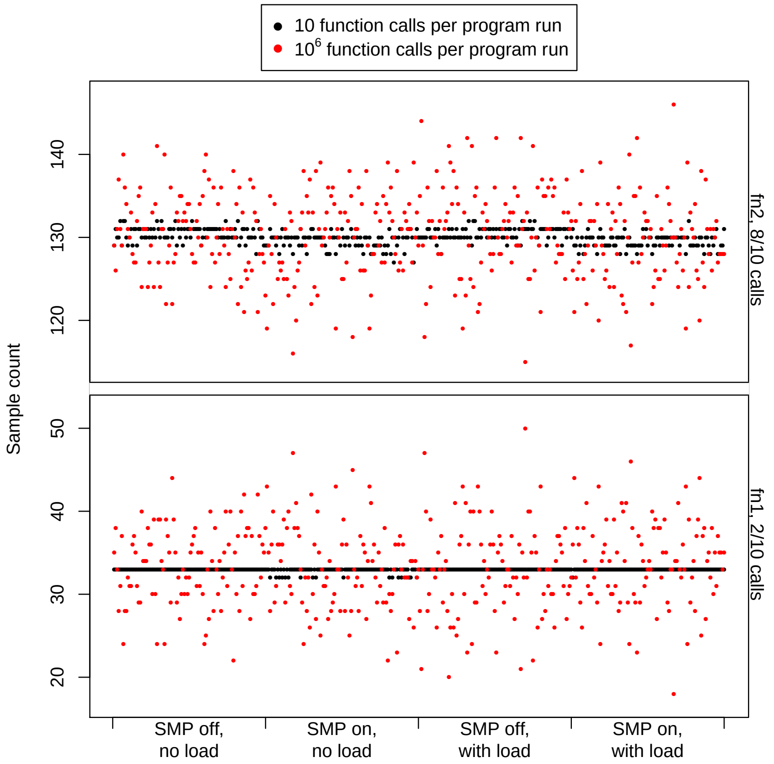

Figure 22: Gprof sample count variability with competing loads

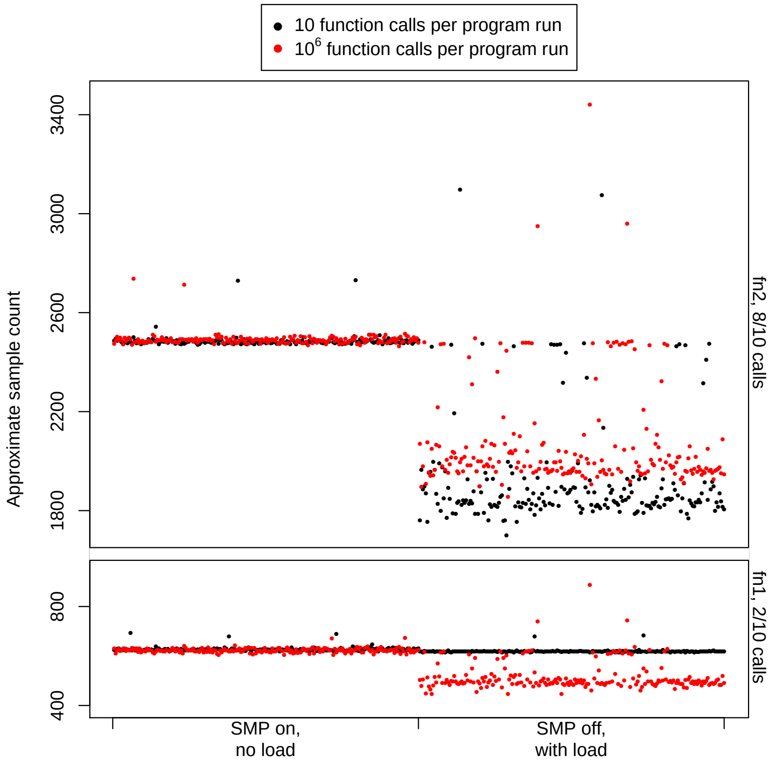

Figure 23: AXE approximated sample count variability with competing loads

7 Uncertainty of expected results

7.1 Background

In a previous report [11], we asserted that a particular test application (val1: triangular distribution of time among N functions) produced anomalous self-time results under certain conditions. We now provide the additional data collection and analysis needed to support that assertion. In this report, it serves as a worked example where the dominant component of uncertainty that is critical to the conclusion arises not from the variability of individual measurements but from the extrapolation of an expected result.

With default values of all parameters, the expected distribution of time (both self and total) among functions in the test application val1 is approximately

Function	Time
function1	5/15
function2	4/15
function3	3/15
function4	2/15
function5	1/15
main	0

Data were collected with the default values for all parameters and in two other configurations:

1. –DINNERLOOP=1 has the effect of reducing the number of busy-work iterations in the leaf functions from 256 to 1 and of increasing the number of main loop iterations by the same factor of 256 to maintain a similar total run time. The main program therefore could be expected to have a greater proportion of self time than it would have in the default configuration. Nevertheless, the leaf functions would still be expected to show a similar linear relationship in how self time is allocated among them.
2. –DINNERLOOP=1 –DREVERSE reverses the order in which the main program invokes leaf functions but is otherwise the same as –DINNERLOOP=1.

7.2 Test conditions

- Controlled variables
  - Dell Precision T5400 PC as used in [25], fixed CPU frequency
  - Slackware Linux 14.0 booted in single-user mode
  - Linux kernel version 3.7.2 with profiling features enabled
  - Test cases built with GCC 4.6.3; kernel with GCC 4.7.2
  - 100 repeats of each combination, run serially
- Independent variables
  - 3 levels of Val1 configuration
    1. Default (nickname “original”)
    2. –DINNERLOOP=1 (nickname “forward”)
    3. –DINNERLOOP=1 –DREVERSE (nickname “reverse”)
  - 6 levels of program compile options × profiling options
    - ∗ –O1 –pg, implicit –fomit-frame-pointer
      1. gprof
    - ∗ –O3, implicit –fomit-frame-pointer
      2. perf
      3. perf-PEBS[12] (PERFEVENT=cpu-cycles:pp)

[12] Precise Event-Based Sampling [40, §18.4.4].

- –O3 –fno-omit-frame-pointer

4. operf (the new tool) from OProfile 0.9.8
5. opcontrol (the "legacy" tool) from OProfile 0.9.8
6. perf

Call chains were not recorded during any of the runs. There was no rotation of combinations; each one was tested with 100 consecutive repetitions.

7.3 Summary data

7 means (`function1` through `function5` plus `main` and "other") from 18 combinations (6 combinations for each of original, forward, reverse) gives 126 means (of 100 results each) with their associated uncertainties.

Possible comparisons to be made include:

- 7 functions against each other;
- 3 executables against each other;
- 6 measurements of a given configuration against each other;
- 3 configurations against each other;
- Everything against the expected results.

Unless there are very close comparisons where statistical significance is a problem, it is most expedient to use the Šidák inequality [23, 24] to set the coverage probabilities of intervals for each of the 126 means high enough so that the levels of confidence for all possible statements about the results are simultaneously controlled. Allowing for 126^2 possible statements about the means individually or pairwise[13] and using 0.95 as the minimum level of confidence to be achieved for any statement, the level of confidence for each mean is taken to be $0.95^{(1/126^2)} = 0.999996769$. The corresponding k value for 99 degrees of freedom (obtainable using the Octave or R syntax given in Section 3.3) is approximately 4.936.

The resulting means with their expanded uncertainties are shown in Figure 24. Several patterns are immediately obvious:

- In the forward configuration, the leaf functions `function1` through `function5` do not present the expected straight-line relationship of their sample counts. Furthermore, Gprof's results for this configuration diverge from those of the other tools.
- The main program does show significant self time in the forward and reverse configurations.
- Gprof results have wider confidence intervals. Distributions of results notwithstanding, this is to be expected since Gprof's sampling rate is about 3.3 % that of the other tools (100 Hz / 2992.5 Hz) and the sample counts are correspondingly lower.
- In the forward and reverse configurations, perf-PEBS culls an apparently unbiased selection of samples from the leaf functions and transfers these samples to "other." (Specifically, it transfers about 3 % of samples to an "[unknown]" function.)

All of the distributions of results for the original configuration were well-behaved. In the other two configurations, Gprof exhibited some long tails and outliers, while Opcontrol exhibited isolated extreme high values appearing in every plot for `function1` through `function5`. Autocorrelation was significant only for "other," which is explainable by the inclusion of time-varying kernel overhead in that category.

This was the only experiment in which the bootstrap method made a significant albeit immaterial difference in the confidence intervals for the means (for Gprof and Opcontrol). Use of the bootstrap method with solitary extreme outliers in the input raises validity questions since the original sample was not large enough to provide a credible estimate of their frequencies in the steady state. With a sample size of 100, their frequencies could not be estimated any lower than $\frac{1}{100}$. Since the difference that bootstrap made was not enough to materially impact the findings, we have stayed with the original method for the confidence intervals of the means.

[13] Less conservatively, one could allow for $\binom{126}{2} + 126$ possible statements, but in this case it makes little difference.

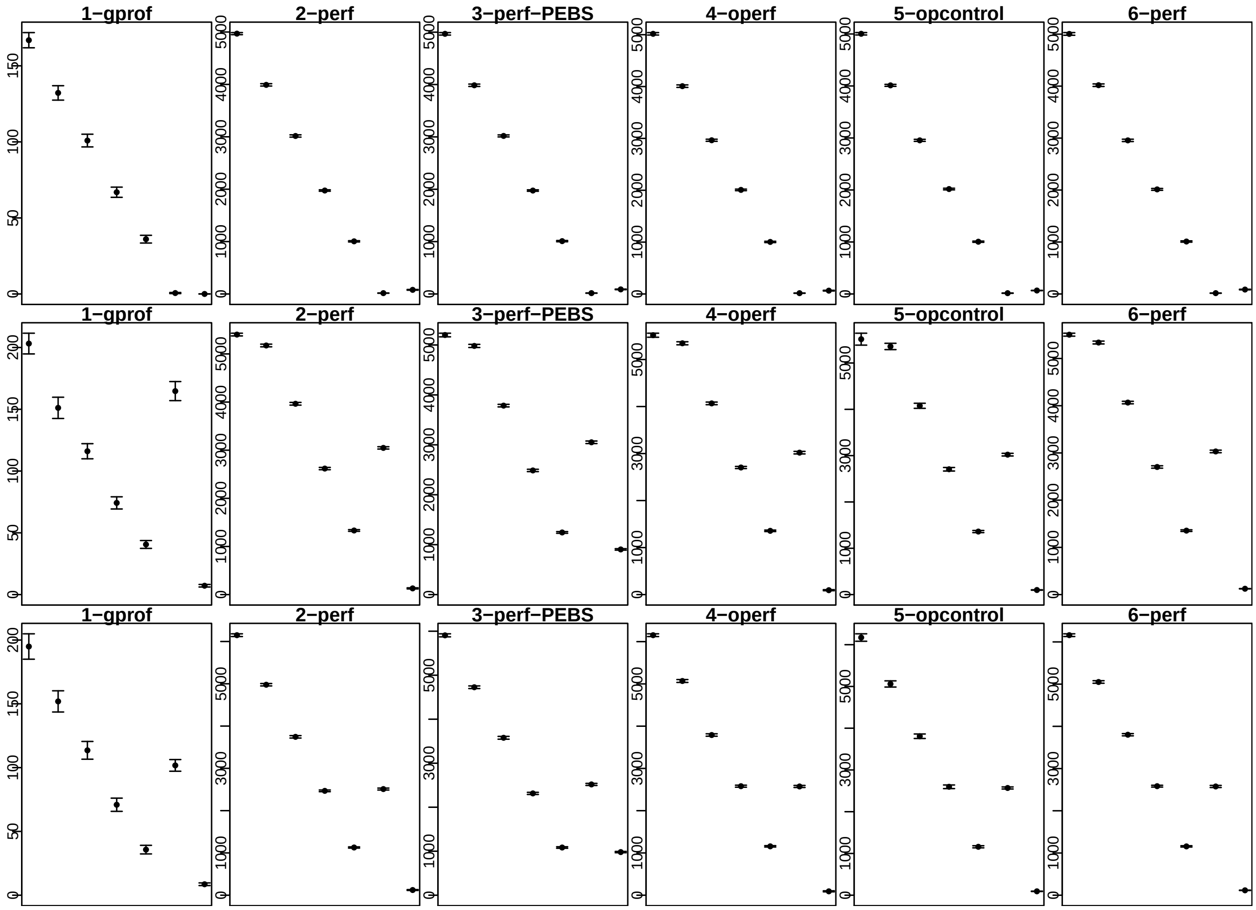

Figure 24: Mean sample counts with expanded uncertainty from each combination of options for configurations original (row 1), forward (row 2), and reverse (row 3). In each plot, the functions from left to right are `function1` through `function5`, `main`, and "other." Coverage is presumed to be less than specified for Gprof and Opcontrol in the forward and reverse configurations.

7.4 Demonstration of anomaly

A result is anomalous only if the confidence interval defined by the result and its associated uncertainty excludes the expected result. In this example, the expected result for one function is understood in terms of its proportion of self time in relation to that of four other functions.

Figure 25 shows results from Perf with –fomit-frame-pointer in the original configuration. As expected, the sample counts for the five leaf functions form a linear progression. To enable an apples-to-apples comparison in discussion below, the expected sample count for `function1` was projected based on the other four functions' sample counts. The linear model is an ordinary least-squares fit, and the uncertainty of the projection is based on the residual standard deviation from the fit and a 95 % confidence interval. The standard deviations of means were considered for use in a weighted least squares (WLS) fit, but the residual standard deviation from the fit of the straight-line model is much larger, negating the need for WLS.

Figure 26 shows the corresponding results from Perf with the forward configuration. Using the same approach that was used for Figure 25, the sample count of `function1` this time falls significantly below a line fit through the other four functions' results and outside of the derived confidence interval for the projection. Having accounted for the additional uncertainty incurred by the linear extrapolation, it is possible to conclude that an anomaly exists in the reported sample count of `function1` as it relates to the other four leaf functions. Of course, this does not explain why such an anomaly would exist. Hypotheses regarding this particular anomaly were discussed in [11].

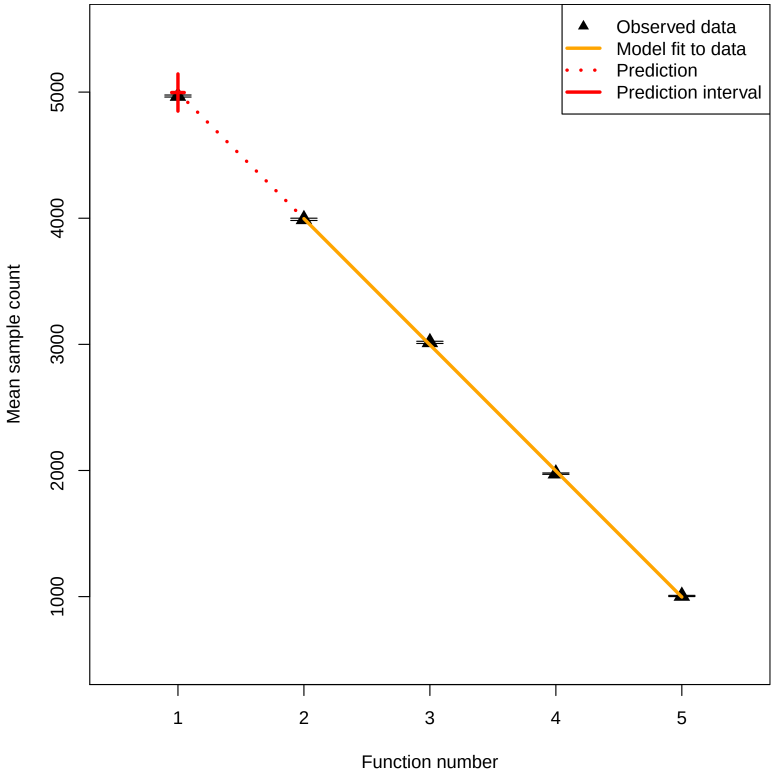

Figure 25: "Original:" Linear fit to Perf mean sample counts for functions 2 through 5 with an observed `function1` execution time consistent with the data for functions 2 through 5 demonstrated by the fact that the observed result for `function1` is captured in the 95 % prediction interval shown. (95 % confidence intervals are also shown for the individual means.)

Finally, Figure 27 shows the corresponding results from Gprof with the forward configuration. In this case, the sample count for `function1` falls well above the expected value, but is still barely contained in the 95 % prediction interval. The statistical significance of the deviation is therefore not quite great enough to conclude that the Gprof results are also anomalous.

8 Possible biases

The usual interpretation of sampling-based profiles assumes that the proportion of samples that are taken within a particular function, relative to the total number of samples, is representative of the proportion of execution time that that function consumed.

If the sampling is unbiased, then pooling the samples from N separate runs is comparable to increasing the sampling frequency within a single run by a factor of N. Pooling is a simple and effective way of obtaining summary results while preserving the equal weighting of all samples across all runs. The downside is that the original method of estimating uncertainty is then no longer feasible, since only one result is produced.

Biased sampling would result in a function being sampled more or less frequently than its share of the application's execution time would indicate, causing its self time to be overestimated or underestimated accordingly. In that case, pooling the samples from multiple runs is correspondingly less productive and an uncertainty calculated based on the assumption that the runs were independent of one another likely would be too optimistic. In the extreme case of sampling that hits exactly the same instruction points every time a program is executed, the results would be completely self-consistent, and the additional runs contribute no new information about the closeness of the measured quantity values to the true or ideal values.

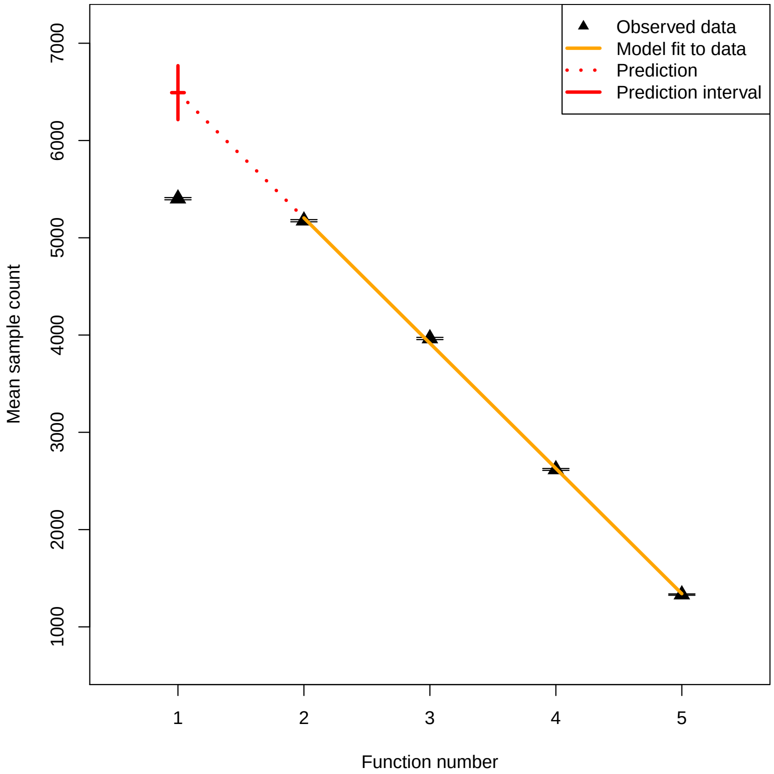

Figure 26: "Forward:" Linear fit to Perf mean sample counts for functions 2 through 5 with anomalous `function1` result demonstrated by the fact that the observed result for `function1` is not contained in the 95 % prediction interval shown.

Bias could arise in several ways:

1. For periodic sampling of a nominally deterministic program repeating the same workload run after run, the natural tendency would be for samples to be taken at or near the same instruction points every time, which artificially decreases the run-to-run variation and increases the bias. Actual variability in execution times and in the delivery of sampling interrupts partially mitigates the problem.

2. If the execution times of groups of functions and the nominal sampling period have some mathematical relationship to one another, even significant variation in execution times and in the delivery of sampling interrupts might be insufficient to mitigate the bias. For example, if an application cycles among 10 functions that have an execution time of 100 μs and samples are taken at a nominal frequency of 10 ms, the expected result would be to sample one or two functions repeatedly and miss the others.

3. If the hardware or kernel has an attraction or aversion to servicing profiling interrupts at a particular instruction point, the samples will avoid certain areas and clump up in others. If this causes samples to shift from one function to another, the measurement of self time will be biased accordingly.

4. If the servicing of interrupts is delayed by an equal amount for every instruction point, there is no net bias except at the very beginning, where an unsampleable "blind spot" exists.

Pragmatically, when one is unsure whether some controllable factor could be introducing bias into results, it is best to include that factor as an independent variable in the experiment. The first two of the biases enumerated above could be mitigated by explicitly randomizing the interval between samples (not supported in current tooling) or by at least comparing the results obtained with several different sampling frequencies.

Regarding the third bias, we performed an experiment on the Linux PC in which a function consisting of 256 identical, 1-byte `nop` instructions followed by a `ret` instruction was invoked repeatedly by a main program that incorporated some random noise. Specifically, on each iteration, the main program read a byte from

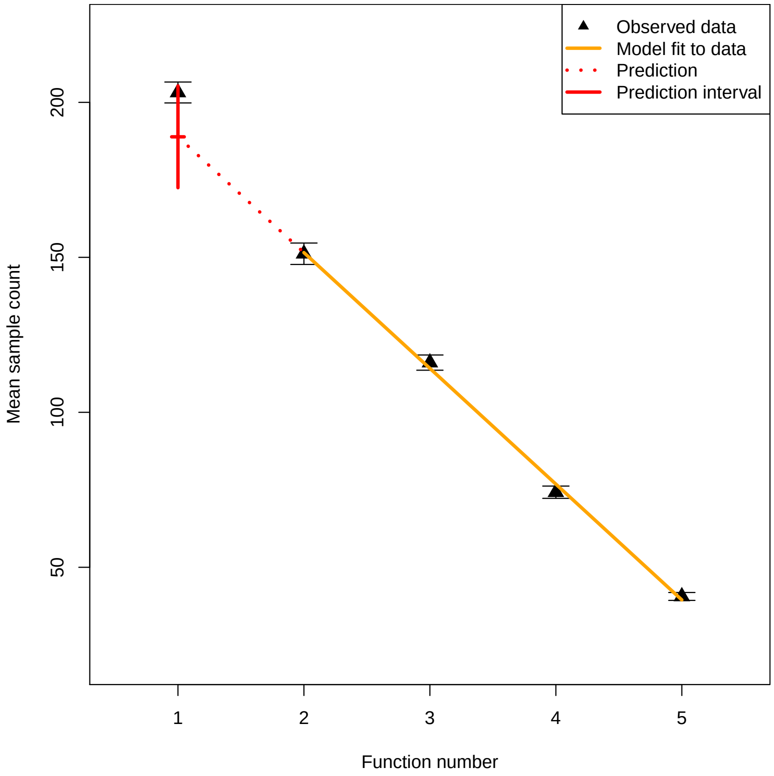

Figure 27: "Forward:" Linear fit to Gprof mean sample counts for functions 2 through 5 with an observed `function1` execution time that is barely contained in the 95 % prediction interval.

/dev/urandom (an operation which itself contributes noise) and then performed a do-nothing loop for 0–15 iterations as determined from the four high bits. The test program was then profiled with perf record –e cpu-cycles:pp –c 1000003.

The purpose of both the main program noise and the prime number for the event count was to attempt to control factors that could cause a nonuniform sampling of the `nop` function. It was already clear from the anomaly reported in the previous section, which arose in the "forward" configuration but not in "reverse," that the main program could influence the results for sampled functions in a simple test program. We hoped to see a nearly flat distribution in which each address within the `nop` function was sampled (i.e., appeared as the instruction pointer address reported by Perf) with approximately the same frequency. Ultimately, 253 of the 257 addresses in the function were sampled at least once; however, the distribution of addresses was distinctly nonuniform (Figure 28), showing a strong but not absolute preference for addresses ending in binary 10 (0b10) (Figure 29) as well as unexplained patterns on a larger scale. This unexpected result adds to a growing list of anomalies that deserve further investigation.

9 Conclusion

This report has demonstrated the role of uncertainty estimation in the interpretation of application profile self-time results and in comparisons between results, investigated several factors impacting the variability of such results, and discussed possible biases.

Our findings are as follows:

- Our results do not support either of two previously published estimates for the uncertainty of self-time measurements.

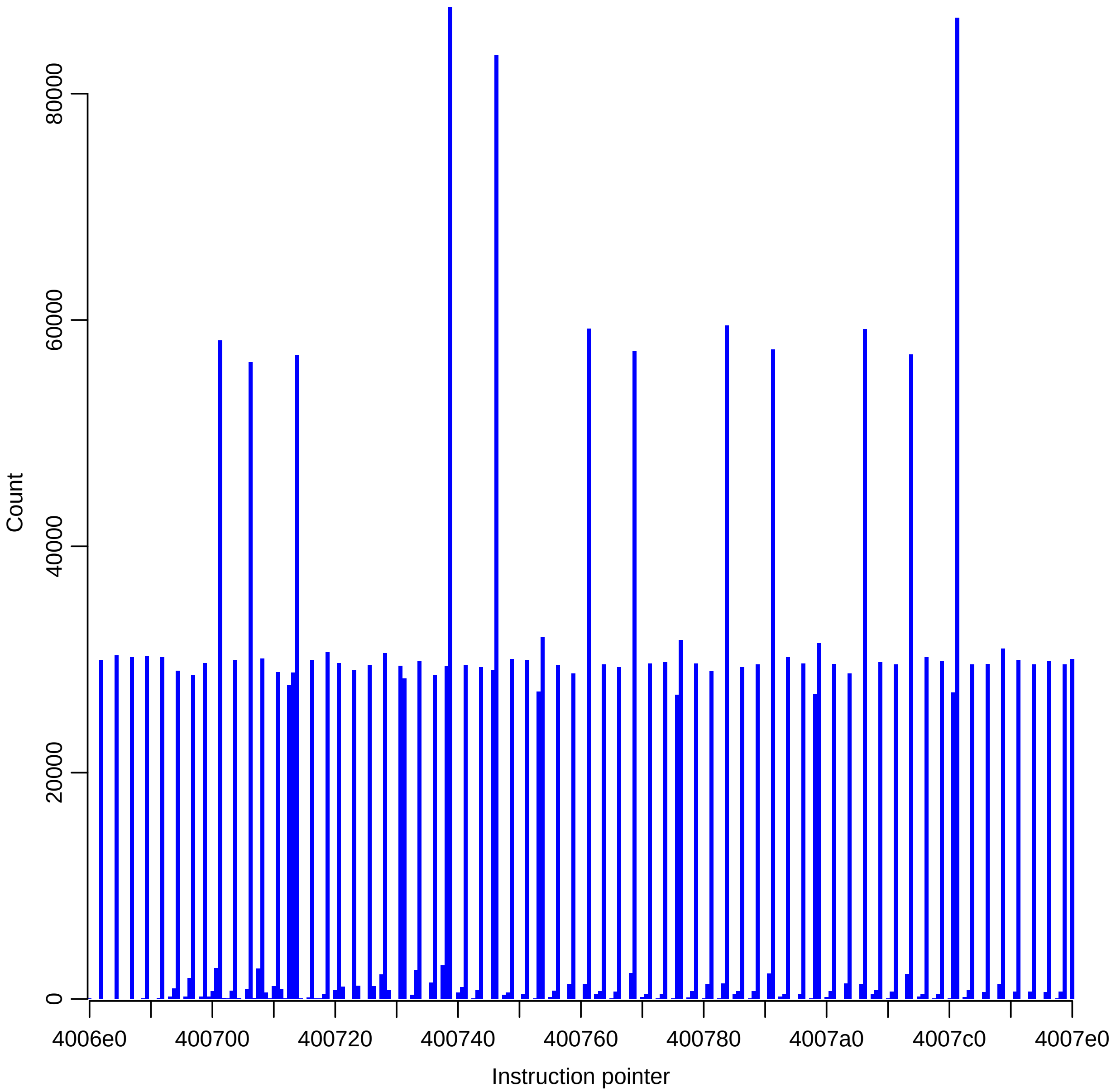

Figure 28: Histogram of instruction pointer addresses sampled by Perf within a function consisting of 256 identical, 1-byte nop instructions followed by a ret instruction

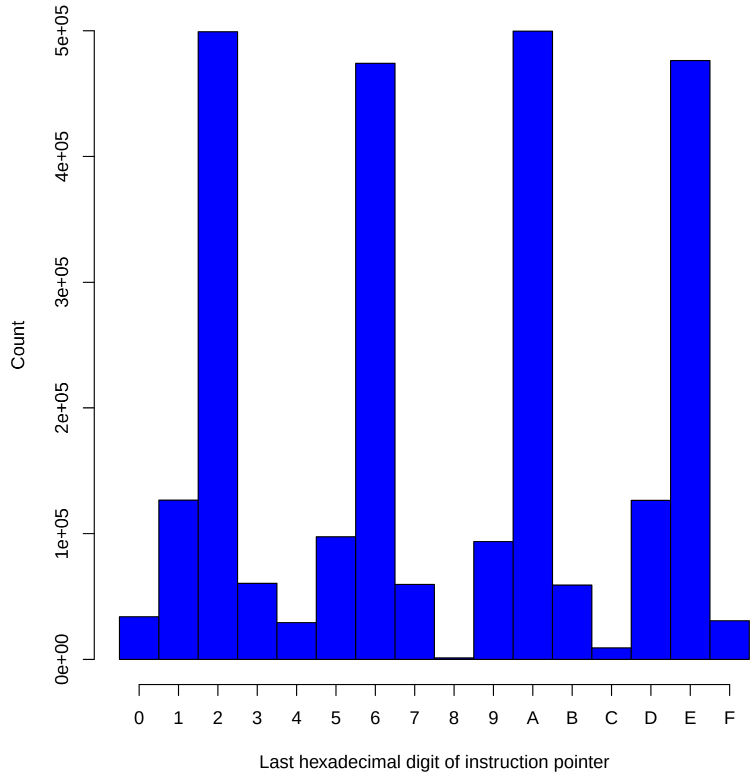

Figure 29: Id., instruction pointer addresses modulo 16

- Our results show a pattern of the actual variability being strongly influenced by self-time fragmentation, a factor which cannot be determined from the data yielded by sampling alone. Although the model of quantization error in Section 3.2.3 attempts to account for the uncertainty arising from self-time fragmentation, the observed variances in most cases are much less than the uncertainty bounds based on this model.

- Sources of noise that are specific to a particular environment or measuring instrument can have a greater effect, as can competing loads (but not always).

- Results can be skewed by seemingly extraneous factors, such as the main program differences that impacted the test in Section 7, and potentially by sampling bias, which the experiment of Section 8 did nothing to exclude.

- Projections made in the process of matching observed results with expected results can introduce uncertainties that exceed those of the original measurements.

- Turbo Boost and any functionally similar dynamic CPU frequency adjustment features *may* introduce significant time- and temperature-dependency, which invalidates the original method of determining uncertainty that assumes that the observations are independent and identically distributed.

- Small-sample results are vulnerable to being invalidated by outliers whose relative frequency becomes apparent only in larger samples.

Given those findings, we recommend the following:

- The number of repetitions of benchmarks should not be determined based on rules of thumb or subjective consistency of results. Benchmarks should be repeated a sufficient number of times to permit assessment of the distribution of results and to achieve an acceptably small confidence interval according to the original method *at least*.

- If the distribution of results is obviously non-normal or changes based on unknown or uncontrollable factors, the risk of drawing invalid conclusions from small samples is correspondingly amplified, and the repetitions should be increased accordingly to limit that risk. If the distribution is predictable and any secondary modes have been adequately characterized, the bootstrap method can be applied to find a more accurate confidence interval or to validate the output of the original method.

- With functions having short execution times being particularly susceptible to self-time fragmentation and suffering greater impact from any skewing and sampling bias that might occur, it is advisable to identify short, frequently-invoked functions when possible and use results with extra caution whenever they are found.

- If measurements cannot be isolated from competing loads, then the impact of competing loads on the measurements should be evaluated *in situ* and taken into account when analyzing results.

- When applicable, non-experimental sources of uncertainty, such as the uncertainty introduced when a model is fit to experimental observations, should be evaluated.

- If relative comparisons of self time are adequate for the purpose, then one should prefer a count of CPU cycles, rather than time, as the trigger for sampling events, and prefer a count of samples, rather than CPU or elapsed time, as the measure of self time, to factor out the effects of CPU frequency scaling.

- Check recorded data for evidence of time- and temperature-dependency. If it is evident, repeat the experiment with dynamic CPU frequency adjustment features such as Turbo Boost and SpeedStep disabled and note in analysis that performance under normal conditions would vary. If thermal limits do not permit this, collected data will have to be filtered to deal with the time- and temperature-dependency of results. At a minimum, “cold” and “hot” results should be evaluated separately.

Additional research on this topic should

- Narrow down the self-time fragmentation effect by comparing results with a modified test program in which `fn1` and `fn2` have the same number of calls but the `fn2` workload is four times as long;

- Determine whether call counts or function entry-exit traces provide sufficient additional information to accurately predict the uncertainty;

- Determine whether explicit randomization of sampling intervals has a beneficial effect;

- Follow up on plausible explanations of observed anomalies to confirm or refute the presence of bias;

- Do more experiments with different languages and measuring instruments; and

- When possible, follow up on AXE for Windows with the “runsa” data collection driver and on Perf for Android to determine whether the secondary modes go away and results are comparable to those seen under Linux with Perf and Gprof.

The raw data and source code pertaining to this report are available online [41].

Acknowledgments

William F. Guthrie of the Statistical Engineering Division provided the analysis for Section 7, the second bound and visualization in Section 3.2.3, and extensive consultation throughout.

The test driver and test program for Android were developed by Ramón Luis De Jesús as part of a Summer Undergraduate Research Fellowship (SURF).

Thanks to Vreda Pieterse, Barbara Guttman, Ram Sriram, and other reviewers for their helpful suggestions.

Special thanks to Barbara Guttman for group management and operational support to make this research possible.

References

[1] Martin Reiser. *The Oberon System User Guide and Programmer's Manual*. ACM Press, 1991. As quoted in "Wirth's law," Wikipedia, 2013, http://en.wikipedia.org/wiki/Wirth's_law.

[2] Perf: Linux profiling with performance counters, 2012. https://perf.wiki.kernel.org/.

[3] OProfile, 2012. http://oprofile.sourceforge.net/.

[4] Jay Fenlason. GNU gprof, 1988. In GNU binutils, http://www.gnu.org/software/binutils.

[5] GCC, the GNU Compiler Collection, 2012. http://gcc.gnu.org/.

[6] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: a call graph execution profiler. *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 17(6):120–126, June 1982. http://docs.freebsd.org/44doc/psd/18.gprof/paper.pdf.

[7] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. An execution profiler for modular programs. *Software—Practice and Experience*, 13:671–685, 1983.

[8] Intel VTune Amplifier XE, 2013. http://software.intel.com/en-us/intel-vtune-amplifier-xe.

[9] Intel Composer XE, 2013. http://software.intel.com/en-us/intel-composer-xe.

[10] Documentation on Android Debug class, 2013. http://developer.android.com/reference/android/os/Debug.html.

[11] David Flater. Configuration of profiling tools for C/C++ applications under 64-bit Linux. NIST Technical Note 1790, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, March 2013. http://dx.doi.org/10.6028/NIST.TN.1790.

[12] Joint Committee for Guides in Metrology. *Evaluation of measurement data—Guide to the expression of uncertainty in measurement*. JCGM 100:2008, http://www.bipm.org/utils/common/documents/jcgm/JCGM_100_2008_E.pdf.

[13] Joint Committee for Guides in Metrology. *Evaluation of measurement data—Supplement 1 to the "Guide to the expression of uncertainty in measurement"—Propagation of distributions using a Monte Carlo method*. JCGM 101:2008, http://www.bipm.org/utils/common/documents/jcgm/JCGM_101_2008_E.pdf.

[14] Barry N. Taylor and Chris E. Kuyatt. Guidelines for evaluating and expressing the uncertainty of NIST measurement results. NIST Technical Note 1297, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, 1994. http://www.nist.gov/pml/pubs/tn1297/.

[15] Barry N. Taylor and Peter J. Mohr. Uncertainty of Measurement Results, in The NIST Reference on Constants, Units, and Uncertainty, October 2000. http://physics.nist.gov/cuu/Uncertainty/.

[16] Barry N. Taylor and Peter J. Mohr. Evaluating uncertainty components: Type A, in The NIST Reference on Constants, Units, and Uncertainty, October 2000. http://physics.nist.gov/cuu/Uncertainty/typea.html.

[17] Barry N. Taylor and Peter J. Mohr. Evaluating uncertainty components: Type B, in The NIST Reference on Constants, Units, and Uncertainty, October 2000. http://physics.nist.gov/cuu/Uncertainty/typeb.html.

[18] David J. Lilja. *Measuring computer performance: A practitioner's guide*. Cambridge University Press, 2000.

[19] Gprof version 2.22.52.0.2 manual, §6.1, Statistical Sampling Error. GNU info document, available from Linux command line by typing `info gprof`.

[20] Joint Committee for Guides in Metrology. *International vocabulary of metrology—Basic and general concepts and associated terms (VIM)*, 3rd edition. JCGM 200:2012, http://www.bipm.org/en/publications/guides/vim.html.

[21] GNU Octave, 2013. http://www.gnu.org/software/octave/.

[22] The R project for statistical computing, 2013. http://www.r-project.org/.

[23] Hervé Abdi. Bonferroni test. In Neil J. Salkind, editor, *Encyclopedia of Measurement and Statistics*, pages 103–107. SAGE Publications, 2007. A different version of the article is available as http://www.utdallas.edu/~herve/Abdi-Bonferroni2007-pretty.pdf.

[24] Zbyněk Šidák. Rectangular confidence regions for the means of multivariate normal distributions. *Journal of the American Statistical Association*, 62(318):626–633, June 1967. Corollary 1. http://www.jstor.org/stable/2283989.

[25] David Flater and William F. Guthrie. A case study of performance degradation attributable to run-time bounds checks on C++ vector access. *NIST Journal of Research*, 118:260–279, May 2013. http://dx.doi.org/10.6028/jres.118.012.

[26] Persi Diaconis and Bradley Efron. Computer-intensive methods in statistics. *Scientific American*, 248(5):116–130, May 1983.

[27] Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall, 1993.

[28] James Carpenter and John Bithell. Bootstrap confidence intervals: when, which what? A practical guide for medical statisticians. *Statistics in Medicine*, 19(9):1141–1164, May 2000.

[29] Brian Ripley. Boot: bootstrap functions (originally by Angelo Canty for S), version 1.3-9, April 2013. http://cran.r-project.org/web/packages/boot/.

[30] Bandwidth selectors for kernel density estimation, in inside-R, 2013. http://www.inside-r.org/r-doc/stats/bw.nrd0.

[31] Ronald E. Walpole and Raymond H. Myers. *Probability and Statistics for Engineers and Scientists*. Macmillan Publishing Company, 4th edition, 1989.

[32] Wikipedia. Estimation of parameters, in Normal distribution, July 2013. http://en.wikipedia.org/wiki/Normal_distribution#Estimation_of_parameters.

[33] Greg Kroah-Hartman. Re: how to list the current HZ value?, August 1 2009. http://www.mail-archive.com/kernelnewbies@nl.linux.org/msg08855.html.

[34] Gprof version 2.22.52.0.2 manual, §9.1, Implementation of Profiling. GNU info document, available from Linux command line by typing `info gprof`.

[35] Wikipedia. Interquartile range of distributions, August 2013. http://en.wikipedia.org/wiki/Interquartile_range#Interquartile_range%_of_distributions.

[36] Droid X, July 2013. http://en.wikipedia.org/w/index.php?title=Droid_X&oldid=564860365.

[37] Documentation on Android dmtracedump tool, 2013. http://developer.android.com/tools/help/dmtracedump.html.

[38] Test driver for Android, version 1.0, available at Software Performance Project web page, 2013. http://www.nist.gov/itl/ssd/cs/software-performance.cfm.

[39] IntelBurnTest version 2.54, July 2012. https://www.xgamingstudio.com/files/IntelBurnTest.zip.

[40] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2, August 2012. http://download.intel.com/products/processor/manual/253669.pdf.

[41] Raw data and source code distribution for "Estimation of uncertainty in application profiles," available at Software Performance Project web page, 2013. http://www.nist.gov/itl/ssd/cs/software-performance.cfm.