Accurate Measurement of Small Execution Times – Getting Around Measurement Errors

Carlos Moreno, Sebastian Fischmeister

Abstract—**Engineers and researchers often require accurate measurements of small execution times or duration of events in a program. Errors in the measurement facility can introduce important challenges, especially when measuring small intervals. Mitigating approaches commonly used exhibit several issues; in particular, they only reduce the effect of the error, and never eliminate it. In this letter, we propose a technique to effectively eliminate measurement errors and obtain a robust statistical estimate of execution time or duration of events in a program. The technique is simple to implement, yet it entirely eliminates the systematic (non-random) component of the measurement error, as opposed to simply reduce it. Experimental results confirm the effectiveness of the proposed method.**

I. Motivation

Software engineers and researchers often require accurate measurements of small execution times or duration of events in a program. These measurements may be necessary for example for performance analysis or comparison, or for measurement-based worst-case execution time (WCET) analysis. Obtaining execution times analytically from the assembly code is increasingly difficult with modern architectures, and often the most practical alternative is actual measurement during execution.

Measurement errors and uncertainties introduce an important difficulty, especially for short intervals. For example, a basic approach to measure the execution time of a given function or fragment of code $\mathcal{F}$ is the following:

```
time start = get_current_time()
Execute F
time end = get_current_time()
// execution time = end - start
```

Unfortunately, the resulting measurement includes the actual execution time of $\mathcal{F}$ plus an unknown amount of time corresponding to the internal processing time of **`get_current_time()`**. With modern OSs, invocation of the **`get_current_time()`** facility can even involve a context switch to kernel mode and back. This unknown overhead comprises a systematic (non-random) component and a noise (random) component. The systematic error is in general related to the execution time of the **`get_current_time()`** facility, whereas the random component can be due to a variety of factors such as measurement based on clock ticks or scheduling issues, or even electrical noise in the case of measurement based on pin-toggling. Though our work addresses both, our focus is on the systematic error, which is the one that commonly used approaches fail to effectively eliminate.

Some of the common mitigating approaches only *reduce* the effect of these errors and never completely eliminate it. In our example, a typical mitigation approach consists of executing $\mathcal{F}$ multiple times, as shown below:

```
time start = get_current_time()
Repeat N times:
    Execute F
time end = get_current_time()
// execution time = (end - start) / N
```

The execution time that we obtain is subject to the measurement error divided by N; by choosing a large enough N, we can make this error arbitrarily low. However, this approach has several potential issues: (1) repeating N times can in turn introduce an additional unknown overhead, if coded as a **`for`** loop. Furthermore, this is subject to uncertainty in that the compiler may or may not implement loop unrolling, meaning that the user could be unaware of whether this overhead is present; (2) the efficiency of the error reduction process is low; that is, we require a large N (thus, potentially large experiment times) to significantly reduce the error; and (3) re-executing $\mathcal{F}$ may require re-initialization through some call to an additional function, which then introduces a new uncertainty:

```
time start = get_current_time()
Repeat N times:
    initialize_parameters()
    Execute F
time end = get_current_time()
// execution time = (end - start) / N
```

In this case the measured time corresponds to the execution time of $\mathcal{F}$ plus the execution time of **`initialize_parameters()`** and it is not possible to determine either of the two values.

Other mitigating approaches include profiling the execution time of the **`get_current_time()`** facility. For example, measure the execution time of a null fragment, and the measured average value corresponds to the systematic error or overhead in **`get_current_time()`**:

```
Repeat N times:
    start = get_current_time()
    end = get_current_time()
    sum = sum + (end - start)

overhead = sum / N
```

This approach, however, can be ineffective for modern architectures where low-level hardware aspects such as cache and pipelines can cause a difference in the execution time of **`get_current_time()`** when executed in succession vs. when executed with other code in between calls.

Embedded engineers often measure execution time by observing a signal at an output pin. They instrument the code with an output port instruction to toggle a pin, and measure time between edges to obtain execution times. Though this mechanism in general yields higher accuracy, it is still subject to the same types of measurement errors discussed above.

Carlos Moreno and Sebastian Fischmeister are with the Electrical and Computer Engineering Department of the University of Waterloo, Canada. E-mail: {cmoreno,sfischme}@uwaterloo.ca

II. Related Work

To the best of our knowledge, no concrete effective approaches have been suggested to get around these issues related to measurement errors. Some common ideas and themes seem to be present, some of them oriented to measuring variations in timing parameters (see for example [1]). The effect of outliers is a recurrent theme. Typical approaches involve statistics that are immune to the effect of outliers, such as the median.

Jain [2] and Laplante [3] both discuss performance analysis including estimation of execution times and statistical analysis of experimental data. Oliveira et. al [4] proposed a system where statistically rigorous measurements are extracted under carefully controlled environment, effectively getting around some of the issues that can disrupt the parameters being measured. These works, however, focus on the extraction and analysis of experimental data for performance evaluation, and not on the actual measurement of execution times.

Stewart [5] covers basic techniques for measuring execution time and applicability to WCET and performance analysis. Lilja [6] also covers some of the basic techniques. It provides a good set of definitions, terminology, and modeling of measurement errors and their sources. In particular, [6] presents a good discussion on the notions of accuracy, precision, and resolution. CPU clock cycle counters in modern processors (e.g., Intel [7] and ARM [8]) can provide high resolution, but they don't necessarily avoid the measurement artifacts that reduce accuracy and precision in the measurements.

In the context of WCET analysis, hybrid approaches [9] use static analysis to determine WCET in terms of execution times of fragments. These times are measured with pin-toggling instrumentation, assisted by a special logic-analyzer hardware. Our proposed approach exhibits important advantages over the pin-toggling instrumentation approach, yet it can benefit from the static analysis component that can compensate for complex hardware features that introduce a relationship where execution paths affect the execution times being measured.

III. Getting Around Measurement Errors

We start by addressing the systematic error only. In the next sections we include the random part (noise) in the analysis.

For N consecutive executions of $\mathcal{F}$, the measured execution time T corresponds to $T = NT_\mathrm{e} + \epsilon$, where T_e is the execution time of $\mathcal{F}$ and ϵ is the overhead from the invocations of **`get_current_time()`**. We notice that it is impossible to determine T_e, since we have two unknowns.

The key observation is that with an additional measurement for a different number of consecutive executions, we obtain two independent equations for the two unknowns T_e and ϵ (since ϵ is the systematic error, thus common to all measurements). This allows us to *completely eliminate* the effect of ϵ and thus determine the actual execution time of $\mathcal{F}$:

$$\begin{matrix} T_1 = N_1 T_\mathrm{e} + \epsilon \\ T_2 = N_2 T_\mathrm{e} + \epsilon \end{matrix} \quad \Rightarrow \quad T_\mathrm{e} = \frac{T_1 - T_2}{N_1 - N_2} \tag{1}$$

The simplest strategy to implement this idea, namely *Differential Measurements*, derives directly from Equation (1), choosing $N_1 = 1$ and $N_2 = 2$. The resulting measurement, as discussed above, eliminates the systematic error but is subject to noise. However, we can reduce the effect of noise by repeating this differential measurement multiple times. We remark that coding these repetitions as a **`for`** loop is not an issue, since the loop overhead occurs outside of the measurements. The idea is illustrated below:

```
Repeat N times:
    time T1 = get_current_time()
    Execute F
    time T2 = get_current_time()
    Execute F
    Execute F
    time T3 = get_current_time()
    total += (T3 - T2) - (T2 - T1)
// execution time = total / N
```

This technique is simple to implement, and by choosing a sufficiently large N, we can make the effect of the noise arbitrarily low. However, as our experimental results confirm, the technique presented in the next section produces measurements with higher precision for the same total number of measurements.

IV. Straight Line Fitting

We now present a technique that is more efficient in terms of reduction of the noise for a given total number of measurements. Without loss of generality, we assume a zero-mean model for the measurement noise.[1] In the simpler case where no re-initialization is necessary before each invocation of $\mathcal{F}$, a statistical estimate of T_e can be obtained through a straight line fitting given the multiple points (N_k, T_k), where T_k is the measured time when executing $\mathcal{F}$ N_k times. The slope of this line corresponds to T_e.

We can set up a scheme where M measurements are taken, where the first time we measure one execution of $\mathcal{F}$, then two executions, then three, and so on until measuring M executions of $\mathcal{F}$. Following the above notation (N_k, T_k), this would correspond to the case where $N_k = k$, with $1 \leqslant k \leqslant M$.

Since we do not require a large number of repetitions for $\mathcal{F}$ (i.e., M can be a relatively low value), it is feasible to execute this sequence without requiring a **`for`** loop. This avoids the issue of introducing additional unknown overhead, as mentioned in Section I.

The straight line $y = ax + b$ for a set of M points (x_k, y_k) can be easily determined [10]. In our case, we obtain the value of T_e (the slope, a) substituting x_k $(= N_k) = k$ and $y_k = T_k$. Figure 1 shows an example with $M = 20$ using POSIX's **`clock_gettime()`** to measure a $\approx$40 ns execution time (best-fitting line is $y = 40.4\,x + 18.8$).

One of the important advantages of our method is its robustness against sporadic measurements with large errors. These could be caused, for example, by a timer or I/O interrupt that occurs while executing the function $\mathcal{F}$. The typical approach of using the median of multiple measurements does indeed provide robustness against these occasional large deviations, but it cannot do anything about the systematic error. Averaging multiple medians of multiple sets of measurements becomes expensive in terms of the required experimental time for a given level of accuracy, and it still only reduces the systematic error, instead of eliminating it.

[1] Any non-zero mean — a non-random parameter — can be seen as part of the systematic error ϵ

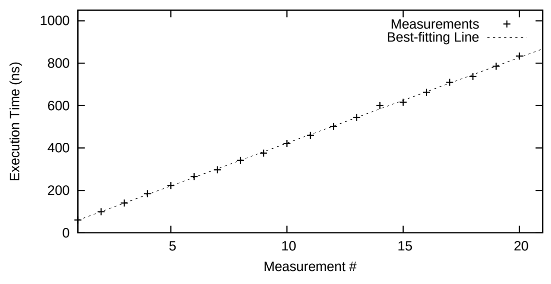

Fig. 1. Example of Straight Line Fitting

A key advantage for our technique is that when performing the line fitting process, the mean square deviation of the points from the best-fitting line quantifies how well the straight line models the measurements; if the line closely models the points, then the measurements are of good quality. Thus, if one (or a small number) of the measurements is subject to a large error, one can easily identify them, as one or a few points will exhibit a deviation from the straight line much larger than the median deviation;[2] thus, this or these few points can be discarded and the straight line is determined with the remaining points.

The differential measurements technique described in Section III also exhibits robustness with respect to outliers. The measurements are already immune to the systematic error ϵ; taking the median of multiple samples introduces immunity to outliers.

V. Overdetermined System of Equations

If the function or fragment $\mathcal{F}$ requires each invocation to be preceded by some initialization function, then our measurement corresponds to $T_{\mathrm{e}} + T_{\mathrm{i}}$, where T_{i} is the execution time of the initialization function, leading again the problem that we can only determine the sum of these two values.

The key observation in this case is that the above problem occurs because the multiple equations are not independent. This linear dependency in the equations is a consequence of the initialization function being executed exactly once per execution of $\mathcal{F}$. If at round k we execute $\mathcal{F}$ N_k times and **`initialize_parameters()`** M_k times ($M_k \geqslant N_k$), then the measured time T_k corresponds to:

$$T_k \;=\; N_k T_{\mathrm{e}} + M_k T_{\mathrm{i}} + \epsilon + \delta_k \tag{2}$$

where δ_k is the random error (noise) for the k-th measurement.

With suitable choices for N_k and M_k to ensure that the equations are independent, we obtain a system of linear equations. To reduce the effect of measurement noise, we take $K > 3$ measurements to obtain an overdetermined system:

$$\begin{bmatrix} N_1 & M_1 & 1 \\ N_2 & M_2 & 1 \\ \vdots & \vdots & \vdots \\ N_K & M_K & 1 \end{bmatrix} \cdot \begin{bmatrix} T_{\mathrm{e}} \\ T_{\mathrm{i}} \\ \epsilon \end{bmatrix} = \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_K \end{bmatrix} \tag{3}$$

This system can be easily solved in the least-square error sense, through standard numerical techniques [10].

[2] Since the mean deviation is affected by the points with large deviations, the median provides a more robust mechanism in this case.

This approach can also be suitable when measuring the execution time of multiple blocks of code, for example the blocks in the control-flow graph (CFG) of a given fragment.

A typical approach in this case is instrumenting the code to toggle a pin at the beginning of each block. However, the instrumentation disrupts the measurements in a way that may be significant, depending on the application.

With our approach, we create an instrumented version of the code and execute it offline to determine the number of times each block executes (e.g., using print statements at the beginning of each block). We also run the code in the target, with pin-toggling only at the entry and exit points, to measure the execution time of the entire fragment of code. Each execution provides one linear equation, allowing us to obtain an overdetermined system. Notice that both versions must run with the same input data (e.g., using a pseudo-random number generator initialized with the same seed).

VI. Experimental Setup and Results

This section describes the experimental setup used to test our techniques. The experiments were performed using an AVR Atmega2560 [11] 8-bit microcontroller running at 1 MHz with the clock signal generated by a crystal.

We performed a calibration phase to compensate for the variations in the frequency of the crystals: a hardware timer divides the system clock frequency to produce 1-second intervals that were measured and used as a 1 second reference. Since deviations in the crystals are due to manufacturing tolerance and temperature variations, we assume that these frequencies remain exactly the same throughout all the experiments. This is the case because the devices were warmed up and thus their temperatures did not change enough during the measurements to have an observable effect on the frequency of the crystals.

In terms of notational convention, we adopt the terminology presented in [6]: accuracy relates to the difference between the measurement and the true value; precision relates to the variation between multiple measurements of the same value; and resolution relates to the size of the quantization step.

A. Differential and Straight-Line Fitting Measurements

Table I shows the results of our measurements using our differential and straight line fitting techniques, as well as the conventional technique of multiple measurements. In all cases, the measurements were based on pin-toggling. We measured the execution time of an assembler-coded routine that takes exactly $100\,\mu$s (exactly 100 clock cycles).

The results are consistent with the intuition that the line fitting technique eliminates the systematic error while the commonly used technique of measuring the time for multiple executions only divides the error by the number of executions. Indeed, we observe in Table I the inverse proportionality relation between the error and the number of measurements for the direct multiple measurement approach (≈ 8 for 1 measurement, ≈ 0.8 for 10 measurements, ≈ 0.4 for 20 measurements, etc.). The differential measurement technique also provides good accuracy, but we see that its precision (as suggested by the variance in the measurements) is low compared to the straight line fitting method for the same total number of

TABLE I
EXECUTION TIME FOR A REFERENCE 100 μS TIME

Technique	**Execution Time (μs)**	**Std. Deviation (μs)**
Line fitting 1× – 10×	99.986	0.125
Line fitting 1× – 20×	100.001	0.044
Differential 55×	99.834	0.237
Differential 210×	100.009	0.121
Differential 1000×	100.003	0.05
Direct (1×)	108.521	1.119
Executing 10×	100.808	0.135
Executing 20×	100.412	0.068
Executing 55×	100.154	0.024
Executing 100×	100.087	0.012
Executing 210×	100.004	0.004

measurements. Increasing the number of measurements in the differential technique can lower this variance, as shown by the results for this technique with 1000 repetitions. Since the differential measurement method is arguably simpler to implement, practitioners could choose it if the experiment duration is not too high.

Though the results show a higher precision in the conventional method, its accuracy is so low that the intervals determined by the variance do not include the true value. The only exception is the last row, corresponding to the highest number of repetitions. An analysis of the reasons for this outcome is beyond the scope of this letter; however, the proposed methods still show higher performance: we recall that a large number of repetitions in this conventional method makes the measurement less robust with respect to outliers, since the probability of measurements with large deviations increases with the number of repetitions. It also makes it hard to code without a for loop, which would further reduce the accuracy. Neither of these are issues with our proposed straight line fitting or differential measurement methods.

B. Execution Time for CFG Basic Blocks

We implemented the technique described in Section V using one of MiBench [12] functions. We chose `adpcm_coder` given its non-trivial CFG. Given the CFG structure, some groups of blocks are bound to execute the same number of times. In those cases, we combined them to obtain the sum of their execution times. This corresponds to identifying sets of identical columns in the resulting matrix and leaving only one instance for each set; the corresponding unknown represents the sum of the execution times (which, if not combined, would lead to a singular matrix given the repeated columns). A more detailed discussion can be found in [13]. Each measurement was done over 1000 executions of the function, each time with different input data. We took 100 measurements, which produced reasonably tight 95% confidence intervals. Notice that this setup accounts for randomness in the equations' coefficients that result from an individual experiment, and also for the measurement noise. Table II shows the results for all the unknowns; the first two values correspond to the sum of several blocks. In all cases, the $\pm$ figures correspond to the 95% confidence intervals. Though we only verified blocks BB2, BB4, BB6, and BB8 against the assembler code, the fact that all of the values are extremely close to integer values, with tight confidence intervals, suggests that the results exhibit a good accuracy. We recall that execution times are quantized with 1 μs resolution, since the Atmega2560 MCU instructions all execute in an integer number of clock cycles.

TABLE II
EXECUTION TIME FOR CFG BLOCKS

Basic Block	**Execution Time (μs)**
BB0 et al.	52.01 $\pm$ 0.044
BB1 et al.	98.98 $\pm$ 0.003
BB2	7.997 $\pm$ 0.005
BB4	6.994 $\pm$ 0.006
BB6	4.998 $\pm$ 0.003
BB8	3.000 $\pm$ 0.004
BB18	9.987 $\pm$ 0.016

VII. CONCLUSIONS

In this letter, we proposed a practical approach to perform precise measurements of short execution times or events in programs or embedded systems. The approach is simple and exhibits robustness with respect to outliers. Experimental results confirm the validity and applicability of the technique.

ACKNOWLEDGEMENTS

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada and the Ontario Research Fund.

REFERENCES

[1] D. Brumley and D. Boneh, "Remote Timing Attacks are Practical," *Proceedings of the 18th USENIX Security Conference*, 2003.
[2] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.
[3] P. A. Laplante, *Real-Time Systems Design and Analysis*, Third ed. Wiley-IEEE Press, 2004.
[4] Oliveira et al., "DataMill: Rigorous Performance Evaluation Made Easy," *International Conference on Performance Engineering*, 2013.
[5] D. B. Stewart, "Measuring Execution Time and Real-Time Performance," in *Embedded Systems Conference (ESC)*, 2001.
[6] D. J. Lilja, *Measuring Computer Performance – A Practitioner's Guide*. Cambridge University Press, 2004.
[7] G. Paoloni, "How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures (White Paper)," 2010.
[8] "ARM1156T2F-S – Technical Reference Manual (§3.2.36)," 2007.
[9] Rapita Systems Ltd., "RapiTime Explained (White Paper)," https://www.rapitasystems.com/system/files/RapiTime%20Explained.pdf.
[10] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, Second ed. Cambridge University Press, 1992.
[11] Atmel Corporation, "AVR 8-bit and 32-bit Microcontrollers," 2012, http://www.atmel.com/products/microcontrollers/avr.
[12] Guthaus, M. R. et al., "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE International Workshop on Workload Characterization*. IEEE Computer Society, 2001.
[13] C. Moreno and S. Fischmeister, "Accurate Measurement of Small Execution Times – Getting Around Measurement Errors. Extended Technical Report," https://uwaterloo.ca/embedded-software-group/publications.