Accurate Measurement of Small Execution Times – Getting Around Measurement Errors (Extended Technical Report)

Carlos Moreno, Sebastian Fischmeister

Department of Electrical and Computer Engineering
University of Waterloo. Waterloo, Canada.

E-mail: {cmoreno,sfischme}@uwaterloo.ca

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 technical report, 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.

This technical report is an extended version of the letter submitted to IEEE Embedded Systems Letters on May 2016, revised version submitted on November 2016, accepted for publication on January 2017.

1 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. In the context of safety-critical real-time systems, engineers often need to use a measurement-based approach to determine the worst-case execution time (WCET). In both scenarios, obtaining execution times analytically by inspecting the assembly code is increasingly difficult with modern architectures, and often the most practical alternative is actual measurement during execution. Undocumented proprietary mechanisms often aggravate the issue, leaving measurement as the only feasible alternative.

Measurement errors and uncertainties introduce an important difficulty, especially when measuring short durations of events in a program. 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.

Pin-Toggling Assisted Measurements

Embedded engineers often measure execution time by observing a signal at the output of some pin. They instrument the code with an output port instruction to toggle a pin right before execution of $\mathcal{F}$ and again right after $\mathcal{F}$ completes execution. The time between the two edges in the signal provides a measurement of the execution time of $\mathcal{F}$.

Though this mechanism in general yields substantially higher accuracy, it is still subject to the same types of measurement errors discussed above. Figure 1 illustrates the sources of error in this scenario. At time t_0 the software

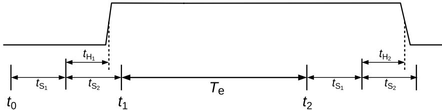

Figure 1: Measurement Errors in Pin-Toggling

starts execution of the pin toggling, consuming a t_{S_1} amount of time. Then, the hardware begins to produce the signal; it will take a t_{H_1} amount of time for the signal to actually change (i.e., produce the edge); this is due to the propagation delay in the internal digital circuitry, plus the propagation time of the analog signal. Notice that it is reasonable to assume that the time it takes for the pin to toggle from 0 to 1 may be different from the time it takes to toggle from 1 to 0, and thus we use t_{H_1} and t_{H_2} to consider this distinction. Simultaneously to this hardware reaction time, the software may take a certain amount of time, denoted t_{S_2}, to complete the pin-toggling procedure and resume execution. After this time has elapsed, at time t_1, execution of $\mathcal{F}$ begins, taking T_e to complete execution. After execution of $\mathcal{F}$, at time t_2, the software toggles the pin again, and the same timing pattern repeats (notice that the software related times are the same, unlike the hardware propagation times).

The measured time $\hat{T}_\text{e}$, from edge to edge in the external signal, is then

$$\begin{aligned}\hat{T}_\text{e} &= (t_2 + t_{S_1} + t_{H_2}) - (t_1 - t_{S_2} + t_{H_1}) \\ &= T_\text{e} + t_{S_1} + t_{S_2} + (t_{H_2} - t_{H_1}) \qquad (1)\end{aligned}$$

We notice a systematic error $\epsilon = t_{S_1} + t_{S_2} + (t_{H_2} - t_{H_1})$. In this case, t_{S_2} could be small or even negligible, but t_{S_1} is necessarily non-zero, and most likely much larger than t_{H_1}. Thus, it is virtually guaranteed that $\epsilon \neq 0$.

Like in the case of software-based measurements, it may be difficult to model this error by measuring the time with a null function $\mathcal{F}$ (i.e., executing the pin-toggle instruction immediately after the previous one, with nothing executed in between). The analog stage of signal propagation in general has a filtering effect (a *smoothing* of the signal) that could distort the edges' shapes. This could indeed considerably change the values of t_{H_1} and t_{H_2} and make this modeling procedure extremely inaccurate.

2 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 through simple statistical processing (see for example [1]). The effect of outliers is a recurrent theme; outliers in the measurements may be caused by hardware effects such as cache and pipeline artifacts, and software effects such as OS-level scheduling. Typical approaches involve statistics that are immune to the effect of outliers, such as taking 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. The authors also presented a study where the effect of the environment is investigated [5]. 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 [6] covers basic techniques for measuring execution time and applicability to WCET and performance analysis. Lilja [7] 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, [7] presents a good discussion on the notions of *accuracy* (related to the difference between measured and true values of the parameter), *precision* (related to the difference between multiple measurements), and *resolution* (related to the quantization effect of the measured value; in particular, the size of the quantum). [7] also discusses the relation of these parameters to the modeling techniques that provide statistical descriptions of measurement errors. CPU

clock cycle counters in modern processors (e.g., Intel [8] and ARM [9]) 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 [10] 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.

3 Getting Around Measurement Errors

For the purpose of simplifying the presentation, we first disregard any random error in the measurement (measurement noise) and only consider the systematic error. The next sections discuss the effect of the measurement noise.

For N consecutive executions of $\mathcal{F}$, the measured execution time T corresponds to:

$$T \;=\; NT_{\mathrm{e}} + \epsilon \tag{2}$$

where T_{e} is the execution time of $\mathcal{F}$ and ϵ is the overhead from the invocations of `get_current_time()`. Equation (2) involves two unknowns, making it impossible to determine T_{e}.

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, common to all measurements). This allows us to *completely eliminate* the effect of ϵ and thus determine the actual execution time of $\mathcal{F}$:

$$\begin{aligned} T_1 &= N_1 T_{\mathrm{e}} + \epsilon \\ T_2 &= N_2 T_{\mathrm{e}} + \epsilon \\ \Rightarrow \quad T_{\mathrm{e}} &= \frac{T_1 - T_2}{N_1 - N_2} \end{aligned} \tag{3}$$

3.1 Differential Measurements

We first show a simple implementation of this idea that eliminates the systematic error and can reduce the effect of measurement noise. The implementation derives directly from Equation (3), choosing $N_1 = 1$ and $N_2 = 2$. The resulting measurement, as discussed above, eliminates the systematic error but is subject to noise. By repeating this differential measurement multiple times, we can reduce the effect of the noise to an arbitrarily low level by choosing a sufficiently large number of repetitions. 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 by the pseudocode 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. This differential measurements technique is simple and is suitable if the total running time of the experiments/measurements is not a critical factor.

4 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. An interesting aspect is that we only need to determine the slope of this line, which corresponds to T_{e}, the parameter that we want to measure.

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 1.

The straight line $y = ax + b$ for a set of M points (x_k, y_k) is given by [11]:

$$a \;=\; \frac{1}{D}\left(M \sum_{k=1}^{M} x_k y_k \;-\; \sum_{k=1}^{M} x_k \, \sum_{k=1}^{M} y_k \right) \tag{4}$$

$$b \;=\; \frac{1}{D}\left(\sum_{k=1}^{M} x_k^2 \, \sum_{k=1}^{M} y_k \;-\; \sum_{k=1}^{M} x_k \sum_{k=1}^{M} x_k y_k \right) \tag{5}$$

with $D \;\triangleq\; M \sum_{k=1}^{M} x_k^2 - \left(\sum_{k=1}^{M} x_k \right)^2$

In our case, we obtain the value of T_{e} (corresponding to the slope, a) substituting x_k $(= N_k) = k$ and

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

$y_k = T_k$. Figure 2 shows an example with $M = 20$ using POSIX's `clock_gettime()` to measure a ≈40 ns execution time (best-fitting line is $y = 40.4\,x + 18.8$). The execu-

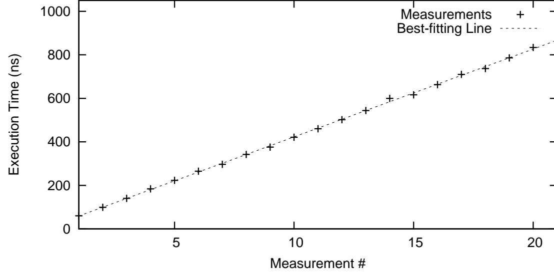

Figure 2: Example of Straight Line Fitting

tion was on a workstation running Ubuntu Linux 14.04 on an Intel Core i7 at 3.5GHz. We executed several times before measurements to "warm up" the system (cache memory, frequency scaling, OS buffers, etc.).

4.1 Robustness of the Measurement

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.

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. Figure 3 shows an example, including the best fitting line considering all points and the best fitting line discarding the outlier. We notice that the remaining points are well aligned (as measured by the mean square deviation from the line), suggesting that the slope of this line is indeed a good approximation of the execution time.

The differential measurements technique described in Section 3.1 also exhibits robustness with respect to outliers. The measurements are already immune to the systematic error ϵ; taking the median of multiple samples introduces

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

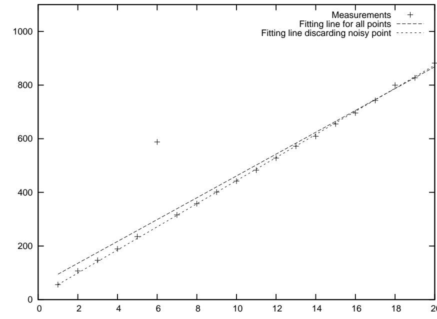

Figure 3: Robustness of the Measurement

immunity to outliers. Even better, we could discard the top and bottom percentiles of measurements and compute the average of the remaining values; this has the benefit that it can produce final measurements with finer resolution than that of the individual ("raw") measurements.

5 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:

$$T_k \;=\; N_k T_\mathrm{e} + M_k T_\mathrm{i} + \epsilon + \delta_k \tag{6}$$

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:

$$\begin{bmatrix} N_1 & M_1 & 1 \\ N_2 & M_2 & 1 \\ \vdots & \vdots & \vdots \\ N_K & M_K & 1 \end{bmatrix} \cdot \begin{bmatrix} T_\mathrm{e} \\ T_\mathrm{i} \\ \epsilon \end{bmatrix} = \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_K \end{bmatrix} \tag{7}$$

If we let $\mathbf{A}$ denote the $K \times 3$ matrix on the left-hand side of Equation (7), and $\mathbf{b}$ the K-element column vector on the right-hand side, then the optimal solution in the least-square error sense is given by [11]:

$$\begin{bmatrix} T_\mathrm{e} \\ T_\mathrm{i} \\ \epsilon \end{bmatrix} \;=\; \left(\mathbf{A}^t\,\mathbf{A}\right)^{-1}\,\left(\mathbf{A}^t\,\mathbf{b}\right) \tag{8}$$

Equation (8) need not be computed explicitly; rather, we premultiply $\mathbf{A}$ and $\mathbf{b}$ by $\mathbf{A}^t$ in the original system to obtain a system with a 3×3 matrix on the left-hand side and a 3-element column vector on the right-hand side. We then use some suitable method such as Gauss-Jordan elimination to determine a solution [11]. Furthermore, since we are only interested in the value of T_e, and considering that we have a system of size 3, a closed-form solution for T_e is feasible and efficient.

The values of N_k's and M_k's should be chosen to ensure that the matrix $\mathbf{A}^t\,\mathbf{A}$ is nonsingular. Like in Section 4, we could choose $N_k = k$. Given that each $M_k \geqslant N_k$, a suitable choice for the values of the M_k's is $M_1 = N_1 = 1$ and $M_k = N_k + 1$ for $k > 1$. It is easy to verify that this leads to a nonsingular $\mathbf{A}^t\,\mathbf{A}$.

5.1 Measuring Execution Time for Multiple Blocks

The approach of the overdetermined system of equations 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.

The idea of the overdetermined system of equations can be applied to this situation; for each execution of the program, we either simulate it or create an instrumented version that is executed off-line and prints the execution trace for the given input. This output reveals the number of times that each block executed; with a measurement of the execution time for the entire program or function, we obtain one equation.

Let $N_k^{(i)}$ denote the number of times that block k executed at the ith round with input data D_i. Let T_k denote the execution time for block k and $T^{(i)}$ the total execution time at the ith round. Then, if we have B blocks:

$$N_1^{(i)} T_1 + N_2^{(i)} T_2 + \cdots + N_B^{(i)} T_B = T^{(i)} \tag{9}$$

If we execute M rounds, with $M > B$, we obtain an overdetermined system of equations. Depending on the values of $N_k^{(i)}$, the system may or may not have a solution; running a large number of rounds, $M >> B$ increases the likelihood of obtaining a solvable (non-singular) system:

$$\begin{bmatrix} N_1^{(1)} & N_2^{(1)} & \cdots & N_B^{(1)} \\ N_1^{(2)} & N_2^{(2)} & \cdots & N_B^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ N_1^{(M)} & N_2^{(M)} & \cdots & N_B^{(M)} \end{bmatrix} \cdot \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_B \end{bmatrix} = \begin{bmatrix} T^{(1)} \\ T^{(2)} \\ \vdots \\ T^{(M)} \end{bmatrix} \tag{10}$$

Depending on the CFG, we may unconditionally obtain a non-solvable system, even for large values of M. Figure 4 shows an example of a CFG that exhibits this issue. In this example, it is clear that regardless of input data or state, entry and exit blocks will each execute exactly once in all instances. Thus, we obtain duplicate columns leading to a singular system; this corresponds to the intuition that every equation involves the sum of the execution times of these blocks, and we do not have enough information to split this sum into the individual values.

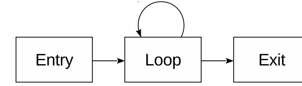

Figure 4: Example of a CFG that Produces a Non-Solvable System

In cases like this, the solution is simple: perform the measurements with one of the two "problematic" blocks excluded. We recall that the total execution time will be measured with either clock facilities or pin toggling, depending on the execution environment. Thus, we can measure total execution time from the beginning of the entry block to the beginning of the exit block (i.e., excluding exit block from the measurement). With this, the system is no longer singular, assuming that the loop block executes variable number of times on different instances. This is presumably easy to control through manipulation of the input data. Once the execution times of the entry and loop blocks are determined, we repeat the experiment including the exit block and easily determine its execution time (since the other two elements are no longer part of the unknowns in the system).

Though we presented a simple example, the idea can be in principle applied to more complex scenarios with more elaborate CFGs, as we will show in Section 6.2. As a last resort, the CFG can always be split into smaller subgraphs and we work individually with those.

The example can be extended to a more complex situation, where for example we could have a sequence of conditionals interleaved with unconditional blocks. Figure 5 shows a simple example where we can see that blocks 0, 3, and 6 will execute the same number of times regardless of input data or state.

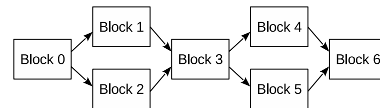

Figure 5: Example of a CFG With Multiple Correlated Blocks

In this case, we could merge the columns corresponding to those, and this produces a solution where the sum of the execution times of these three blocks are a single unknown. Once we determine this sum, we can take additional measurements and work individually on determining each of the merged execution times. Measuring the individual blocks execution times is relatively simple: for example, for the CFG shown in Figure 5, we could place a `goto` or some other "exit" control-flow statement after the last instruc-

tion of block 0 (or at the beginning of both blocks 1 and 2). This gives us one equation where the execution time for block 0 is involved without blocks 3 and 6, allowing us to determine T_0. Once we have T_0, we now repeat the same procedure for block 3, and so on, until we have determined all of the missing execution times.

One general difficulty remains, and it is that of ensuring that every block in the CFG is exercised during the executions. This depends on the inputs, and the problem of determining the inputs that produce a given outcome is known to be an NP-complete problem [12]. However, this is not a problem in practice, if we ensure that the sizes of the CFGs are reasonably small. In real-life systems, with code produced with reasonable quality standards, functions should never be excessively large. This means that we can safely assume that the CFGs of individual functions will have reasonable size.

Pin-Toggling Measurement Errors

Measurements based on pin-toggling can of course be combined with any of the methods presented in the previous sections to eliminate any systematic error and reduce the effect of random variations. Notice, however, that measuring from rising to falling edge produces a different value with respect to measurement from falling to raising edge. Thus, we should toggle the pin in a consistent manner throughout multiple measurements. That is, always toggle from A to $\neg A$ before executing $\mathcal{F}$ and from $\neg A$ to A after execution, where $A \in \{0, 1\}$. to avoid some of the measurements including $t_{H_1} - t_{H_2}$ as part of the error and other measurements including $t_{H_2} - t_{H_1}$.

6 Experimental Setup and Results

This section describes the experimental setup used to test our techniques. The experiments were performed using an AVR Atmega2560 [13] 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 (typically no more than 100 or 200 ppm) are due to manufacturing tolerance and temperature variations, we assume that these frequencies remain exactly the same throughout all the experiments, since the devices were warmed up and thus their temperatures did not change enough to have an observable effect on the frequency of the crystals.

In both experiments, the time measurements were based on pin-toggling. We digitized the signal through a PC sound card. The model used was the HT Omega Claro Halo [14], with 24-bit resolution and 192 kHz sampling rate. Once the signal was digitized, we measured the position of an edge (and thus, edge-to-edge times) by looking for the inflection point (the point where the second derivative changes sign) between two neighboring peaks. Figure 6 shows an example of the smoothed edge due to the effect of the bounded bandwidth in the analog signal, producing an inflection point in the transition. The inflection point can be easily determined

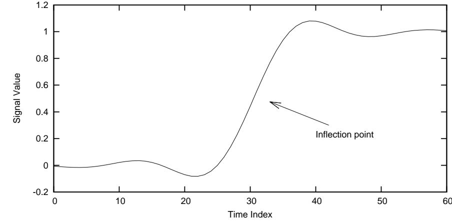

Figure 6: Example of Inflection Point at Edge in Limited-Bandwidth Signal

using the numerical approximations of the derivatives [11]. We also used second-order interpolation to determine the zero-crossing point of the second derivative. Thus, we obtain the position of the inflection point with sub-sample resolution.

In terms of notational convention, we adopt the terminology presented in [7]: 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.

6.1 Differential and Straight-Line Fitting Measurements

Table 1 shows the results of our measurements using our differential and straight-line fitting techniques, as well as the conventional technique of multiple measurements. We

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

Table 1: Execution Time for a Reference 100 μs Time

measured the execution time of an assembler-coded routine that takes exactly 100 μs (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 1 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 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 technical report; 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.

Figure 7 shows a log-log plot of the measurement errors matched against the theoretical ϵ/N error for N measurements. It is assumed from the observations that $\epsilon = 8.5\,\mu$s. This confirms the effect of the systematic measurement error

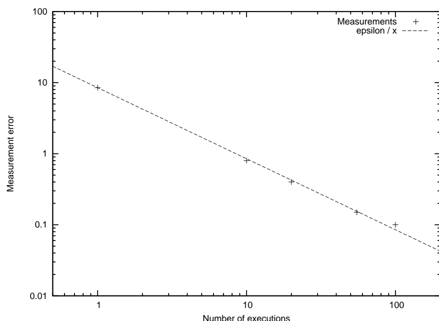

Figure 7: Measurement Error Over Multiple Measurements

ϵ over multiple measurements. Notice that we show these results only as confirmation that the measurements exhibit the expected characteristics, and we do not recommend it as a technique to recover the value of ϵ. We can of course use a curve-fitting procedure to fit the samples to a curve $y = a + b/x$, where a corresponds to the required measurement and b corresponds to ϵ. However, such procedure requires far too many measurements to produce a value of ϵ with good accuracy.

6.2 Execution Time for CFG Basic Blocks

We implemented the technique described in Section 5 using one of MiBench [15] 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). (in the example in Figure 5, one such set would be blocks 0, 3, and 6). 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 2 shows the results for all the

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

Table 2: Execution Time for CFG Blocks

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 $1\,\mu$s resolution, since the Atmega2560 MCU instructions all execute in an integer number of clock cycles.

7 Conclusions

In this technical report, 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] A. Oliveira, J.-C. Petkovich, and S. Fischmeister, "How Much Does Memory Layout Impact Performance? A Wide Study," *Proceedings of the International Workshop on Reproducible Research Methodologies*, 2014.

[6] D. B. Stewart, "Measuring Execution Time and Real-Time Performance," in *Embedded Systems Conference (ESC)*, 2001.

[7] D. J. Lilja, *Measuring Computer Performance – A Practitioner's Guide.* Cambridge University Press, 2004.

[8] G. Paoloni, "How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures (White Paper)," 2010.

[9] "ARM1156T2F-S – Technical Reference Manual (§3.2.36)," 2007.

[10] Rapita Systems Ltd., "RapiTime Explained – Whitepaper," https://www.rapitasystems.com/system/files/RapiTime%20Explained.pdf.

[11] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, Second ed. Cambridge University Press, 1992.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, Third ed. The MIT Press, 2009.

[13] Atmel Corporation, "AVR 8-bit and 32-bit Microcontrollers," 2012, http://www.atmel.com/products/microcontrollers/avr.

[14] HT Omega, "Claro Halo – Online specifications." [Online]. Available: http://www.htomega.com/clarohalo.html

[15] Guthaus, M. R. et al., "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE International Workshop on Workload Characterization.* IEEE Computer Society, 2001.