\textbf{Abstract—}To analyse runtime behaviour and performance of software systems, accurate time measurements are needed. For this, timer methods are used, which are based on hardware timers and counters that are read and processed by several software layers. These processing layers introduce overhead and delays that impact accuracy and statistical validity of measurements, especially for fine-granular measurements. To understand and to control these impacts, the resulting accuracy of timer methods and their invocation costs must be quantified. However, quantitative properties of timer methods are mostly unavailable, as they are platform-specific due to differences in hardware, operating systems and virtual machines. Also, no algorithm exists for precisely quantifying the invocation costs and the accuracy of timer methods, so programmers have to work with coarse estimates and cannot evaluate and compare different timer methods and timer APIs. In this paper, we present \textsc{TimerMeter}, a novel algorithm for platform-independent quantification of accuracy and invocation cost of any timer methods, without inspecting their implementation. We evaluate our approach on timer methods provided by the Java platform API, and also on additional timer methods that access hardware and software timers from Java. We additionally perform a comparison of obtained results, which benefits researchers and programmers by forming a basis for selecting appropriate timers.

\section{I. Introduction}

Analysing extra-functional properties of software systems (such as performance or reliability) often requires quantitative statements about time intervals and time points of events. In order to obtain time points, programmers are accustomed to calling timer methods provided by APIs of operating systems, virtual machines, third-party frameworks, etc. But when performing fine-granular or accuracy-sensitive measurements, scientists need to account for the different timer methods and their invocation costs (i.e. execution duration overhead) of such timer calls, as these factors have a great impact on the accuracy and statistical validity of their measurements. For example, a timer method that uses a timer which is updated once every 15 ms is not appropriate for measuring an operation that takes 250 ns.

Unfortunately, quantitative properties of timer methods are not specified in documentation because these properties are platform-specific: they depend on the underlying hardware, and on the software stack that processes the hardware signals. Also, no algorithms or tools exist to quantify timer accuracy and invocation costs, as the update frequency of hardware signals (e.g. from the CPU clock) differs among vendors, and in general is not exposed by the operating system APIs. Additionally, the operating system and other software layers hide the management of CPU throttling and multi-core CPUs.

Hence, programmers and scientists have to guess the accuracy and invocation costs of software timers, or perform ad-hoc experiments to estimate these values. Published timer properties are mostly vague and provided without the code that produced them, so it is not possible to apply these platform-specific results to other hardware/software platforms without re-running that code. For example, the \texttt{nanoTime} method in the Java platform API is officially documented to provide “nanosecond precision, but not necessarily nanosecond accuracy”. For execution platforms that perform extensive run-time optimisations (e.g. Java Virtual Machines that perform just-in-time compilation), optimisations must be considered when investigating and comparing timer methods.

The contribution of this paper is a novel algorithm for quantifying accuracy and invocation costs of timer methods. Our algorithm is called \textsc{TimerMeter} and it can be applied to any timer method, no matter which underlying hardware and software counters are accessed. Due to its transparency and ease of use, it can be used by programmers and scientists on-site, allowing credible, precise and quick statements about timers for a concrete hardware/software system and its configuration. Our solution does not require any modification or reconfiguration of the hardware or software on which it runs.

Our work supports both timer methods that return \textit{absolute} time points (e.g. calendar date offsets as returned by the \texttt{currentTimeMillis} method in the Java API) and timer methods that return \textit{relative} time points (e.g. nanoseconds passed since “fixed but arbitrary time”, as returned by the \texttt{nanoTime} method in the Java API).

We evaluate the \textsc{TimerMeter} approach by applying its Java implementation to all timer methods in the Java SE 6 platform API. Additionally, we apply \textsc{TimerMeter} to several timer methods that directly access the OS timers and to two third-party Java time measurement tools (JETM [1] and GAGE [2]), and compare the obtained results.

The remainder of this paper is structured as follows: Sec. II provides the foundations, while Sec. III discusses the effects of rounding and truncating in timer method implementations. Sec. IV describes our approach for quantifying the accuracy and the invocation cost of timer methods and we extensively evaluate it in Sec. V. In Sec. VI, we present related work.
Finally, the paper concludes with Sec. VII.

II. FOUNDATIONS

In this paper, the timer method properties such as accuracy are considered as they are seen at the application level by the method user who invokes the timer method synchronously. In addition to wall-clock time which advances globally, there exist timers that allow to measure thread time or user time. Our work is applicable to both types of timers, as will be shown in Sec. V.

In literature, the terms accuracy, precision, resolution of timer methods are often used interchangeably, or even with contradicting meanings. For this paper, we adopt and extend the terminology from the official Java platform API documentation of Sun Microsystems, Inc., and take the following view:

Accuracy (synonymously: resolution) is the smallest non-zero time interval that can be measured with the timer, i.e., $\text{accuracy} := \min(t_2 - t_1)$ with $t_2 > t_1$. Accuracy of hardware counters may be a floating-point number.

Precision is the smallest time unit in which the timer returned values may be represented. Thus, it holds that $\text{accuracy} \geq \text{precision}$. For example, the precision of java.lang.System.nanoTime() is 1 ns, although in practice, its accuracy is often hundreds of ns.

Invocation cost is a synonym for execution duration and spans the interval from the timer method invocation until it returns, as seen by the method’s invoker.

A. The Relation between Accuracy and Timer Method Values

In this section, we show which issues with timer methods need to be considered. We assume that (i) during the considered measurements, no jumps in wall-clock time happen (e.g. no switch from summer to winter time occurs) (ii) no timer overflow happens (i.e. all timer values grow monotonically) (iii) the same timer instance is used throughout an example (i.e. on multi-core platforms, hardware counters/registers that are used belong to the same core).

To compute the time value to return, a timer method reads a counter which is updated (increased) at regular intervals of the same length. This means that several subsequent timer method invocations can return the same value if the counter value has not been increased in between.

To see why, consider Fig. 1: $U_k, U_{k+1}$ etc. are the moments where the counter is updated (i.e. incremented). When the timer method reads the counter value in the interval $[U_k, U_{k+1})$, it will use $U_k$ as the counter value. In the case shown in Fig. 1, the timer method accuracy $A$ is $U_{k+1} - U_k$ (for any $k \geq 0$), and it will be equal to or larger than the timer method’s precision (cf. the definition of accuracy above).

This means that a measurement at time point $t_x$ is not necessarily returned as $t_x$, as illustrated by the effect of timer accuracy $A$ in Fig. 1: The timer method returns the last stored timer value $U_k$ instead of the (precise) value of $t_x$, this is hinted by the dashed line in Fig. 1 and in the following figures. In the best case, the returned value $U_k$ is equal to $t_x$ while in the worst case, the returned value $U_k$ is smaller than $t_x$ by almost the entire size of $A$.

B. A Naive Approach to Estimating Timer Invocation Costs

The most straightforward way to measure the duration of a method call $\text{meth}()$ is to place it between two timer method invocations and to compute the difference of the measured values as in Listing 1 (here, we use a fictional timer $\text{Timer}$).

Listing 1. Oversimplified measurement of method execution duration

```java
long time1 = Timer.time();
meth();
long time2 = Timer.time();
long duration = time2 - time1;
```

However, there are several critical issues with the Listing 1. Consider the case in Fig. 2: the invocation cost of the $\text{time}()$ call is larger than its accuracy, and also the duration of $\text{meth}()$ is larger than the timer accuracy. Hence, the value of duration corresponds to the duration of $\text{time}()$ plus that of $\text{meth}()$, not just to that of $\text{meth}()$!

Likewise, consider the case in Fig. 3: there, the execution duration of $\text{time}()$ is less than the accuracy of the timer, and so is that of $\text{meth}()$. Yet the duration of $\text{time}()$ is still large enough to influence the (supposedly) measured duration of $\text{meth}()$.

Fig. 1. Effects of timer accuracy and on measurements (Legend: $t_x$: actual time to be measured; $U_k$: counter updates; $A$: timer method accuracy)

Fig. 2. Timer method execution duration larger than its accuracy

Fig. 3. Timer method execution duration smaller or equal to its accuracy
Trying to obtain the invocation cost of `time()`, the straightforward way is to remove the call to `meth()` from Listing 1, and re-run the measurement. Meyerhoefer’s code [3] does so, and his algorithm is enhanced to account for cases where `time2==time1`. However, for timers where the invocation cost is smaller than half of the accuracy (e.g. `java.lang.System.currentTimeMillis()` in Java - cf. Sec. V), duration is likely to be zero, as in the case shown in Fig. 3 if `meth` were removed.

Hence, to know whether the timer accuracy suffices for measuring the duration of `meth()` according to Listing 1, we have to obtain both the timer invocation costs and the timer resolution. Stochastic approaches [4], [5], [6] which are described in Sec. VI are not helpful in this situation, as they would measure either the invocation cost or the timer accuracy: they cannot know beforehand whether the situation in Fig. 2 or in Fig. 3 is present.

III. THE EFFECTS OF Rounding AND TRUNCATING

Consider an example counter that is updated with a fixed frequency of 3,579,545 Hz (Sec. V discusses such an OS counter). This frequency means that the counter’s accuracy (\(= \frac{1}{\text{frequency}}\)) is 297.4 ns when rounded to one decimal place (in the remainder of this paper, we omit time units to simplify the discussion). Yet most timer methods, such as `java.lang.System.nanoTime()`, return values as whole-numbered `longs` and not as `doubles`, i.e. without any decimal places. Therefore, the timer method implementation has two choices to convert `double` values such as 297.4 to `longs`: (i) truncating (e.g. using Java casting operator) and (ii) rounding (e.g. using Java API method `java.lang.Math.round(double d)`), both of which introduce numerical errors.

As we consider the timer methods as “black boxes” (i.e. we don’t analyse their implementations), we cannot know beforehand whether truncation (or rounding) is used or not. Yet for devising our algorithm in Sec. IV, the effects of rounding and truncating on timer values and time intervals will play a crucial role. Thus, in this section, we prove that when using truncation or rounding to record `double`-typed time points as whole-numbered `long`-typed values, it is possible that two time intervals of the same actual length will be recorded as `long`-typed intervals whose lengths differ by 1.

For truncating (i.e. cutting off all decimal places), consider a timer interval \(E - S\) that starts at \(S\) and ends at \(E\). Let \(A\) be the accuracy of the timer, \(\text{trunc}(S)\) be the truncated value of \(S\) and \(\text{trunc}(E)\) the truncated value of \(E\).

Due to truncation, the computed time intervals can appear larger than they are in some cases and smaller than they are in others. As an example, consider a case with \(A = 297.4\) and two intervals of length 3 · \(A\) each: the first starting at 7 · \(A\) and ending at 10 · \(A\), and the second starting at 10 · \(A\) and ending at 13 · \(A\). Without truncation, the duration of the both intervals is computed to 3 · 297.4 = 892.2. With truncation, the duration of the first interval is computed to \(\text{trunc}(10 \cdot 297.4) - \text{trunc}(7 \cdot 297.4) = \text{trunc}(2974.0) - \text{trunc}(2081.8) = 2974 - 2081 = 893\), which is larger than the actual duration of 892.2. In contrast to that, the duration of the second interval appears shorter due to truncation: \(\text{trunc}(13 \cdot 297.4) - \text{trunc}(10 \cdot 297.4) = \text{trunc}(3866.2) - \text{trunc}(2974.0) = 3866 - 2974 = 892 < 892.2\).

We define the truncation-caused interval measurement error \(\text{IME}_{\text{trunc}}(E, S) := (E - S) - (\text{trunc}(E) - \text{trunc}(S))\), which is equivalent to \((E - \text{trunc}(E)) - (S - \text{trunc}(S))\). It holds that \(0 \leq (E - \text{trunc}(E)) < 1\) and as \(0 \leq (S - \text{trunc}(S)) < 1\).

Thus, we can state that the largest value of \(\text{IME}_{\text{trunc}}(E, S)\) is achieved when \(S - \text{trunc}(S) = 0\) and \(E - \text{trunc}(E)\) is maximised (yet still \(E - \text{trunc}(E) < 1\)). Correspondingly, the smallest value of \(\text{IME}_{\text{trunc}}(E, S)\) is achieved when \(S - \text{trunc}(S)\) is maximised (yet still \(S - \text{trunc}(S) < 1\)) and \(E - \text{trunc}(E) = 0\). Finally, we can summarise that \(-1 < \text{IME}_{\text{trunc}}(E, S) < 1\), and the open interval \((-1, 1)\) contains at most two whole-numbered `long` values (i.e. without decimal spaces), 892 and 893 in the above example.

For rounding, again consider time interval start \(S\) and end \(E\) and assume that time values with decimal values of 0.5 and larger are rounded up, while smaller decimal values are rounded down. Using above example accuracy of 297.4, consider the time interval between \(S = 1 \cdot 297.4\) and \(E = 2 \cdot 297.4 = 594.8\). \(S\) is rounded to 297 while \(E\) is rounded to 595, the resulting interval \(E - S\) is 298. At the same time, for \(S = 2 \cdot 297.4 = 594.8\) and \(E = 3 \cdot 297.4 = 892\), the same underlying time interval (1 · 297.4) after rounding is computed to 892 – 595 = 297. Thus, an interval can appear both longer and shorter due to rounding. For the rounded value \(\text{round}(S)\) and \(\text{round}(E)\), it holds that \(-0.5 < \text{round}(S) - S \leq 0.5\) and \(-0.5 < \text{round}(E) - E \leq 0.5\).

We define the rounding-caused interval measurement error \(\text{IME}_{\text{round}}(E, S) := (E - S) - (\text{round}(E) - \text{round}(S))\), which is equivalent to \((E - \text{round}(E)) - (S - \text{round}(S))\). \(\text{IME}_{\text{round}}(E, S)\) achieves its largest (positive) value \(E - \text{round}(E)\) is maximized and \(S - \text{round}(S)\) is minimised. Let \(\epsilon\) be an arbitrarily small value with \(0 < \epsilon < 1\). The maximum value of \(E - \text{round}(E)\) is \(0.5 - \epsilon\) (when \(E\) is rounded down) and the minimum value of \(S - \text{round}(S)\) is \(-0.5\) (when \(S\) is rounded up). Hence, the maximum value of \((E - \text{round}(E)) - (S - \text{round}(S))\) is \(1 - \epsilon\), which is smaller than 1. In a similar way, the minimum value of \(\text{IME}_{\text{round}}(E, S)\) is achieved when \(E - \text{round}(E)\) is minimised (i.e. it is \(-0.5\)) and \(S - \text{round}(S)\) is maximised (i.e. \(0.5\)). Thus, the minimum value of \((\text{round}(E) - E) - (\text{round}(S) - S)\) is \(-1 + \epsilon\). Altogether, it holds that \(-1 < \text{IME}_{\text{round}}(E, S) < 1\), and the open interval \((-1, 1)\) contains at most two whole-numbered `long` values (i.e. without decimal spaces).

Thus, we conclude that both for truncating and for rounding, if the real duration of a time interval \(S - E\) has non-zero decimal values, its duration as a whole-numbered `long` can be computed in up to two versions which have a difference of 1. We will pay respect to this conclusion on truncation and on rounding in the next section, where we describe our novel approach to compute both the execution cost and the accuracy of timer calls.
IV. Quantifying Timer Invocation Cost and Accuracy

In this section, we start by computing the timer method invocation cost using a simple, motivating example shown in Fig. 4 and described in Section IV-A. We then expand that algorithm in Sec. IV-B to compute the accuracy of timer methods.

A. Initial Algorithm for Quantifying Timer Method Properties

Algorithm 1 in Fig. 4 measures the duration of the java.lang.System.nanoTime() timer method. Analyzing the source code of Algorithm 1 in the left part of Fig. 4 and the bytecode compiled from it (shown on the right part of Fig. 4), we see that between the two timer calls, only a simple LSTORE operation exists (it is highlighted in the shown bytecode listing), which stores the time value into timel. We note that the LSTORE instruction is short and quick when compared to the implementations of the called timer method(s). Hence, we do not expect the LSTORE instruction to contribute significantly to the difference time2-time1 when compared to the cost of nanoTime.

Now, when running the above code on Sun JDK 1.6.0_07 (default JIT and JVM settings, Windows XP Professional OS, Intel T2400 CPU), we obtain the following statistics for elements of results: minimum value is 1676 ns, median value is 1956 ns, and the maximum value is 4190 ns. The initial interpretation of these results can be the following: the lower values (1676 ns) are the minimal cost of invoking nanoTime(), the larger median values (e.g. 1956 ns) are due to delays caused e.g. by CPU scheduling, and the largest values are outliers caused by garbage collection, significant OS interference etc.

We repeated the algorithm in Fig. 4 one hundred times, and obtained same minimum and median values in results. We have also repeated the measurements for nanoTime() on different hardware, operating systems and JVMs and with a larger number of measurements than 300 as used in Fig. 4, and observed similar outcomes. During the experiments, we have minimised the load on the machine to allow the minimal timer invocation cost to appear.

A closer look at the measured results reveals that there are a few measurement results that yield 1676 ns or 1677 ns, and the remaining majority yields 1955 ns or 1956 ns. In particular, there are absolutely no measurements between 1677 ns and 1955 ns, and the measurements following 1956 ns have a significant distance (278 ns and 279 ns, as well as multitudes of those) to 1956 ns, which is very similar to the distance between 1676 ns/1677 ns and 1955 ns/1956 ns. Thus, the results are forming “clusters” with small intra-cluster element distances of 1 ns and larger inter-cluster distances of ca. 279 ns.

A plausible explanation of intra-cluster differences is given by the effects of rounding and truncating (cf. Section III). The inter-cluster differences appear to be due to the accuracy of the timer method, i.e. the values of 1955 ns/1956 ns are equal to “timer invocation cost + 1 timer method accuracy”.

To see whether the inter-cluster distances indeed allow to derive the timer method accuracy, additional measurements must be performed. The central, novel idea for such additional measurements is to “insert” small but sufficient amount of computation between the timer method invocations, so that the time value read by the second timer method invocation is one accuracy “step” larger than without the “inserted” work. In Section IV-B, the details of an algorithm implementing this idea are given.

Unfortunately, using the timer method currentTimeMillis() instead of nanoTime() in Fig. 4 leads to measurements where results contains almost only zeros. This means that the minimal and the median invocation cost of currentTimeMillis are smaller than its accuracy. Also, there are no “clusters” which would hint at currentTimeMillis’ accuracy.

Hence, Algorithm 1 cannot be used in general for computing the timer method accuracy, and a more refined algorithm is needed. In the next subsection, we show how the accuracy of timer methods can be computed for both the case where the invocation cost is smaller than the accuracy, and for the opposite case. We also discuss how rounding/truncating effects are dealt with.

B. Quantifying Timer Method Accuracy and Invocation Cost

As discussed in the previous section, if the invocation cost of the timer method is smaller than its accuracy, the two timer method calls as in Algorithm 1 in Fig. 4 are likely to return the same value for time1 and time2, which is not helpful in finding the timer method’s accuracy using clustering. Hence, we must “force” the second timer invocation to return a value which is one accuracy “step” higher.

To do so, instead of invoking the second timer call in Algorithm 1 immediately after the first one, we shall execute a very small task between the two timer calls so that the inserted task cannot be optimised away by the execution environment. If the inserted task is too small for a non-zero difference to appear, it should be enlarged until time2-time1 (cf. Algorithm 1 in Fig. 4) is non-zero. Further enlargement of the inserted task shall lead to time2-time1 becoming another accuracy “step” larger. In Listing 2, Algorithm 2 which performs this task is given and it is described in this section in more detail. Algorithm 2 is the simplified representation of the core of our TImeMeter approach.

In Part A of Algorithm 2, the timer invocation cost is computed. Note that if the invocation cost is smaller than the precision of the timer method, minimum- and/or median- and/or maximumTimerInvocationCost can be zero, similar to Algorithm 1.

In Part B of Algorithm 2, the work performed between the timer invocations is gradually increased, to allow the time interval to grow by one duration of timer accuracy. Note that as the globalVariable incremented in Algorithm 2 is global and could be read by other methods/classes, the incrementation task will not be “optimised away” by the JVM (this has been confirmed in our observations). Hence, it can be
expected that each iteration of the loop will be executed, and no iteration will be skipped by the execution environment. The employed mathematical operations (addition etc.) have equivalents in most current programming languages, so our approach can be implemented using them, not only Java. Our current Java implementation of TIMERMETR is further, efficiency-increasing checks (not shown in the listing) that let the algorithm terminate as soon as a predefined number of distinct values have been saved into frequencies. The algorithm aggregates individual interval length values into the Map-typed frequencies structure that maps the interval lengths to the number of their occurrences.

In Part C of ALGORITHM 2, we cluster the intervals to obtain the accuracy of the timer method. The motivation for using clustering is that one interval value may have up to two long-typed values due to rounding/truncation, as shown in Sec. III. Thus, a cluster can contain at most two intervals which must be immediate neighbors with intra-cluster distance of 1 precision unit. If an interval length with distance 1 to the larger element in a given cluster appears, it starts a new cluster (in accordance with our findings in Sec. III). For example, 1676 ns and 1677 ns would belong to the same cluster, and 1955 ns and 1956 ns to another one. Based on observations made in Sec. IV-A, the timer method accuracy can then be computed from inter-cluster distances.

Finally, in Part D, the first two clusters are used to compute the accuracy of the timer method as the distance between their cluster centers. We define a cluster center as the average of the two (or one) value(s) contained in the cluster, independently from which value of each value in the cluster. For example, the cluster center for a cluster with 224 values of 1676 ns and 101 values of 1677 ns is still 1676.5 ns. With the cluster center of 1955 ns/1956 ns being 1955.5 ns, the timer accuracy would be computed to 1955.5 ns-1676.5 ns=279 ns.
currCluster = new Cluster();
currCluster.firstElement = newValue;
currCluster.firstElementFrequency = frequencies.get(newValue);
currOpen = true;
} else {
  if (currOpen){
    currCluster.secondElement = newValue;
currCluster.secondElementFrequency = frequencies.get(newValue);
clusters.add(currCluster);
currOpen = false;
} else {
  currCluster = new Cluster();
currCluster.firstElement = newValue;
currCluster.firstElementFrequency = frequencies.get(newValue);
currOpen = true;
}
}

// D. compute accuracy from the first two clusters
// (this is a simplified view of the algorithm)
Cluster clusterA = clusters.get(0);
Cluster clusterB = clusters.get(1);
double accuracy, clusterCenterA, clusterCenterB;
if (clusterA.secondElement != -1){
  clusterCenterA = (clusterA.firstElement + clusterA.secondElement) / 2;
} else {clusterCenterA = clusterA.firstElement;}
if (clusterB.secondElement != -1){
  clusterCenterB = (clusterB.firstElement + clusterB.secondElement) / 2;
} else {clusterCenterB = clusterB.firstElement;}
accuracy = clusterCenterB - clusterCenterA;

For the ALGORITHM 2 to work, several constraints must be fulfilled.

Firstly, there must be at least two clusters, and the centers of neighboring clusters must indeed be one timer method accuracy apart. TIMERMETIER fulfills this constraint by a sufficiently high numberOfWorkIncreaseSteps (e.g. 1000) and other inputs, for which TIMERMETIER already provides suitable defaults. Using them, the first constraint is fulfilled in practice by all timer methods we studied (cf. Sec. V).

Secondly, the first two clusters should be “complete”, i.e. they should contain two values if these values can indeed occur. For the above example, if 1677 ns would be missing in the first cluster and 1955 ns would be missing in the second, the timer method accuracy would be computed as 1956 ns - 1676 ns = 280 ns. On the other hand, if 1676 ns would be missing in the first cluster and 1956 ns would be missing in the second, the timer method accuracy would be computed as 1956 ns - 1675 ns = 281 ns. Thus, having only one value in first or second cluster (or both) may lead to a deviation of at most \pm 1 precision unit (here, \pm 1 ns). The current Java implementation of TIMERMETIER analyses all clusters that emerge from step D of ALGORITHM 2 to detect and correct such situations.

V. EVALUATION

In this section, we present a case study which evaluates the Java implementation of the TIMERMETIER approach on two different hardware platforms, using two different operating systems and 8 different timers, including timers provided by hardware, operating systems, Java platform API, and also third-party libraries/tools.

We begin by describing the individual timer methods in a bottom-up way, from hardware over the operating systems to the JVM and third-party libraries. All discussed timer methods return 64 bit values, which overflow after 2^{64} timer precision units (however, even for nanosecond precision, 2^{64} ns > 550 years, making an overflow very unlikely).

A. Studied Timers and Timer Methods

TSC (Time Stamp Counter) is a 64-bit register present on many, but not all, x86 and x64 processors [7]. It was available on both computers used in our evaluation. TSC counts the number of CPU ticks since the last CPU reset, and is accessible through the RDTSC (“read TSC”) assembler instruction. The RDTSC can be wrapped for Java access using JNI. However, this wrapping differs between Linux and Windows. For the case study, the used Windows version was based on a DLL and associated JNI code provided by Roedy Green [8].

Although the TSC is considered to have a high accuracy and a low overhead, its use is problematic when the CPU clock rate changes (e.g. in energy-saving CPU modes), when out-of-order execution of instructions happens, or on multi-core/multi-CPU machines (due to unsynchronised TSCs). Relying on TSC may also reduce portability, and a number of Intel processors include a constant-rate TSC, i.e. it is read at the CPU’s maximum clock rate regardless of the actual CPU clock rate, invalidating measurements where execution is partially performed at a lower clock rate.

Also, hardware timers similar to TSC exist, such as HPET (High-Precision Event Timer, which has a guaranteed update frequency of 10 MHz and thus may have a higher accuracy than TSC), RTC (Real Time Clock) or PIT (8254 Programmable Interval Timer). However, HPET’s use is restricted: it is not available from Windows XP, Windows Server 2003 or Linux with Kernel 2.4 and older. Hence, we focused on TSC in this paper, as it is the only hardware timer broadly available and widely used. Still, the TIMERMETIER algorithm can be applied to the other timers, e.g. using a JNI implementation accessing them.

OS-provided timer methods abstract away from hardware timer problems and the intricacies described above. They include, for example, the C/C++ Windows API methods QueryPerformanceCounter() (which returns the underlying counter’s state, not time units) and QueryPerformanceFrequency() (which reports the update frequency of the counter used by the first method). For Linux, the methods clock_gettime and clock_get time (defined in time.h C header file) are available, as well as gettimeofday.

However, the OS-provided timers introduce additional overhead when compared to TSC, and they often rely on TSC, leading to issues with CPUs not properly implementing it [9], [10]. Furthermore, many applications are built on top...
of virtual machines (VMs) which provide their own timer methods that should (or must) be used instead of the specific timer methods provided by operating systems.

**VM-provided timer methods** provide uniform timer access independent of the underlying hardware/software platform. Yet as discussed in Sec. II, the characteristics of VM-provided timer methods can differ depending on the underlying platform. In this paper, we focus on the Java platform and its API, although our approach would be applicable to other VMs, such as Microsoft’s .NET CLR.

The Java platform API provides several timer methods, of which the wall-clock timer methods `nanoTime()` (since Java 1.5) and `currentTimeMillis()`, both in class `java.lang.System`, are the most popular ones. Additionally, the Java platform’s management API provides the method `getThreadCpuTime()` in the class `java.lang.management.ThreadMXBean`, which returns the calling thread’s CPU time once enabled with `setThreadCpuTimeEnabled(true)`.

We also used Sun Microsystems’s proprietary (and undocumented) high-resolution timer method `highResCounter()` in the class `sun.misc.Perf`, which is shipped with JDK 1.6. Using the method `highResFrequency()`, the frequency of this timer can be queried. However, in practice, this timer is rarely used directly, and before the `nanoTime` method was added to the Java platform API in version 1.5, many third-party tools were created to provide timers with better precision (and, thus, better accuracy) than `currentTimeMillis()’ milliseconds`. Some of these tools are still used today, e.g. for systems that run on pre-1.5 JVMs.

Several **third-party tools** that provide Java-accessible timer methods exist. In the case study, we have only considered timer methods that are available both for Windows and Linux operating systems; thus, PAPI [11] and PCL [12] were not considered, though our algorithm and its Java implementation can be applied to them as well. Also, while PAPI is being developed and updated, the last version of PCL dates from January 2003. Instead, we have studied JETM (Java Execution Time Measurement Library [1]), and GAGElimer (Genuine Advantage Gaming Engine timer [2]). From the JETM library, we let JETM select the “best” available timer using `bestAvailableTimer()` method of its class `EtMMonitorFactory`. The timer method used on the obtained instance was `getThreadCpuTime()`. From the GAGEtimer library, we considered the method `getHighCounter()` in class `AdvancedTimer`; the clock’s frequency can be queried using `getTicksPerSecond()`.

**B. Case Study Results**

The case study was performed on two platforms:

- **P1** had an Intel Pentium M 1.73 GHz CPU, 1 GB of RAM, and two operating systems, each with Sun Microsystems JDK 1.6.0_07:
  - openSUSE Linux with Kernel 2.6.25 incl. HPET support (kernel-reported HPET frequency 14,318,180 Hz, i.e. 1 tick every 69.8 ns)
  - Windows XP Professional SP2

- **P2** had an Intel Core Duo T2400 CPU with core frequency of 1.83MHz, 1.5 GB of RAM and Windows XP Professional SP2, with the JVM from Sun Microsystems JDK 1.6.0_07.

This study setup allows for comparisons where of the four factors (CPU, operating system, JVM, timer method), any three factors are fixed, while the remaining one is varied. For example, we can compare the characteristics of a different timer methods on the same execution platform, but we can also compare which OS is better for fine-grained measurements using a particular timer method and a particular JVM.

Table I shows the results of our case study, where “Accuracy” denotes accuracy (i.e. resolution), and “Cost” denotes the invocation cost, i.e. the execution duration of one timer method invocation. The “unit” column corresponds to one precision unit, except for `highResCounter` and `QueryPerformanceCounter`, for which the precision is not specified, but must be retrieved using each timer’s frequency-returning method (`highResFrequency` and `QueryPerformanceFrequency`, respectively).

If the accuracy of a timer is (much) larger than its invocation cost, `TIMER` can only conclude that the invocation costs are between zero and one accuracy (cf. Sec. II). Our case study showed that this is the case for the timer method `getCurrentThreadCpuTime()`, which has a (declared) precision of 1 ns, while, in practice, `TIMER` yields an accuracy of around 15 milliseconds.

To compute the invocation cost for this timer method more precisely, we used a more precise timer (`nanoTime()`). The result, given in Table I, suggests that the invocation cost (less than 1000 ns) can be ignored when using this timer. In a similar way, we used `nanoTime()` to obtain the invocation cost of `currentTimeMillis()`. Hence, the invocation cost is a fraction of the accuracy of `currentTimeMillis()`, which itself is above the method’s precision of 1 nanosecond.

The timer method of GAGElimer is not included in Table I, since it produced results that were absolutely identical to those of `nanoTime()` in all three cases on both platforms. A short inspection of the source code revealed that the timer class of GAGElimer provides the best timer at initialisation, and selects either `nanoTime()` if available, and otherwise either `QueryPerformanceCounter` (if running on Windows), or the method `currentThreadTimeMillis()` (as the “fallback default”). When `nanoTime()` is available, GAGE incorrectly states that the timer accuracy is 1 ns, while `TIMER` returns the correct, platform-specific accuracy.

**C. Discussion, Assumptions and Limitations**

As expected, the TSC timestamp counter has a very high accuracy: 2 CPU cycles on P1 and 11 CPU cycles on P2 (i.e. 1.156 ns on P1 and 6.006 ns on P2). However, the problems
with TSC (cf. Sec. V-A) prevent TSC from becoming the universal (best) choice.

Windows-specific `QueryPerformanceCounter()` method has a precision that depends on the frequency with which the counter is updated. The Windows method `QueryPerformanceFrequency()` returns 3,579,545 (with Hz as unit) as the update frequency on both CPUs, i.e. the (rounded) time spent between the updates is 279.4 ns.

Notably, this counter update frequency does not correlate in any way with the CPU frequencies. Though `TimerMeter` returns the results in counter increments (as returned by the called Windows function), the results can be easily converted into time values (which are same for P1 and P2). Thus, the accuracy of `QueryPerformanceCounter()` is 279.4 ns (= 1 timer increment interval), while its invocation cost is 6 * 279.4 ns = 1676.2 ns.

The timer method `nanoTime()` has (on Windows) an accuracy and an invocation cost that are strikingly similar to those of `QueryPerformanceCounter()`. Hence, it seems that the JVM uses `QueryPerformanceCounter()`, but it must convert the counter ticks into long-typed values that `nanoTime()` returns. Hence, `n` ticks are computed to \((3,579,545 Hz)^{-1} \cdot n\) and then rounded (here: \(n = \frac{1676}{279} \approx 6.007\), so 1676 is in fact 6 * 279.4 = 1676.4). This reinforces our finding on rounding and truncating, as discussed in Sec. III.

On Linux (with the same underlying hardware), `nanoTime()` has a better accuracy and lower invocation costs. The accuracy corresponds to the (rounded) time interval between two successive updates of the HPET timer, whose update frequency the Linux kernel reports to be 14,318,180 Hz. Hence, this interval is \((14,318,180 Hz)^{-1} \approx 69.841\) ns. However, the timer method invocation cost for `nanoTime()` on Linux is smaller than on Windows, but is a larger multiple of the accuracy: \(978 \approx 14.01\).

The unofficial `sun.misc.Perf` counter found in the JDK is not documented in the Java platform API, and does not bring any advantage on the Windows platform in our case study: its accuracy is identical to that of `nanoTime()`, and its invocation cost is even one accuracy unit higher, whereas the accuracy unit can be computed using `sun.misc.Perf's highResFrequency()`, which is identical to the value returned by Windows’ `QueryPerformanceFrequency()` method.

On Linux, `sun.misc.Perf's highResFrequency()` returns 1,000,000; i.e. 1 microsecond (=1000 ns) is spent between the two counter updates. The resulting accuracy of `highResCounter()` is 1000 ns, which is not as good as that of `nanoTime()`. The invocation cost of 2000 ns is over two times larger than that of `nanoTime()`.

The standard Java platform API timer method, `currentTimeMillis()`, has a declared precision of 1 millisecond, i.e. 1,000,000 nanoseconds. Even though `nanoTime()` achieves an accuracy well below 1 ms, `currentTimeMillis()`’s accuracy on Windows is at most 15 ms, which is too coarse-grained for fine-granular scientific measurements. On Linux, the accuracy is 1 ms, equal to one precision unit, and thus an order of magnitude better than using Windows on the same hardware platform.

Given the fact that the timer methods discussed above had an invocation cost of 2000 ns at most, using `currentTimeMillis()` itself in Java implementation of `TimerMeter` to compute its invocation cost returns 1 ms. In fact, this is an upper bound that is too high: hence, we used `nanoTime()` to obtain a better approximation. This resulted in 0.004 ms on Linux and 0.0002 ms on Windows, which is supported by statements of Holmes in [13]: “[Windows implementation of] `currentTimeMillis()` essentially just reads the low-accuracy time-of-day value that Windows maintains. Reading this global variable is naturally very quick [...]”.

The final timer method in the Java platform API that we considered is `getCurrentThreadCpuTime()`, which is documented in the API to have a nanosecond precision. However, `TimerMeter` revealed an accuracy of 15,625,000 nanoseconds, i.e. 15.625 milliseconds on Windows, and an accuracy of 15,000,000 ns (i.e. 15 ms) on Linux. Thus, this timer method is also very coarse-grained.

For `JETM`, as for `GAGE`, the results on both platforms of our case study were identical as for `nanoTime()`. Hence, `JETM` does not bring any advantage on Java 1.5 or later.

### Table I

<table>
<thead>
<tr>
<th>Timer method / Counter</th>
<th>Precision unit</th>
<th>Accuracy</th>
<th>Cost</th>
<th>Cost</th>
<th>Accur.</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Platform 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux 2.6.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDK 1.6.0_07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Win XP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDK 1.6.0_07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Platform P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Win XP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDK 1.6.0_07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Legend:** ⋆: calculated from frequency; ○: invocation cost measured using `java.lang.System.nanoTime()` method; 

---

**TABLE I**

**TimerMeter evaluation in Java**, for 7 timer methods, 2 operating systems and 2 platforms (Legend: ⋆: calculated from frequency; ○: invocation cost measured using `java.lang.System.nanoTime()` method; )

---

**Microarchitecture**
difference to GAGE (as confirmed by its source code) is that JETM uses the sun.misc.Perf timer where nanoTime() is not available. As with GAGE, TIMERMeter dependable quantifies platform-dependent characteristics of JETM’s timer methods.

Among the timer methods we have considered, (RD)TSC has by far the best accuracy, and also the lowest cost (except on Windows, where getCurrentThreadCpuTime() is faster). Thus, from all these results, TSC seems to be the most suitable timer method, unless either a thread CPU time must be measured (instead of wall-clock time), or a complex platform is used (several cores, changing CPU/core frequency). nanoTime() seems another good choice, but its high invocation costs and suboptimal accuracy must be taken into account for fine-granular measurements.

We have assumed that the accuracy of a timer method is stable over time, i.e. the accuracy (resolution) does not change over the course of several timer method invocations. This is a very basic requirement that is needed by any measurements, not only by TIMERMeter. In the course of evaluation, we have not encountered a setup where this assumption would have been violated. Interferences (such as garbage collection) will produce measurement outliers (i.e. longer time intervals than expected), while speedups (e.g. due to JIT) will decrease the timer invocation cost. The latter case is addressed by our approach through the introduction of a warmup phase (not detailed in Algorithm 2 in Sec. IV-B). However, neither interferences nor speedups will affect the timer method accuracy, which is dictated by the usage of underlying timers/counters from the operating system or hardware.

Finally, the efficiency of our approach must be discussed. For any of the considered timer methods, the execution of the current TIMERMeter Java implementation takes less than 3 minutes, on each of the platforms used in the case study (P1 and P2). This time includes the substantial warmup to encourage JIT compilation of timer methods (15,000 calls to the timer method), as well as clustering of the results.

VI. RELATED WORK

Books on performance measurement, evaluation and benchmarking (e.g. [14], [15]) discuss the importance of timer accuracy for quantifying the errors in measurements, but do not provide algorithms for computing the accuracy. Also, the role of the timer method invocation costs is not discussed.

Language-specific books also consider this topic. In “Java Performance Tuning” [16], Shirazi states that “[java.lang.]System.currentTimeMillis() can take up to half a millisecond to execute” (p. 15), but does not explain the origins of this (rather imprecise) statement, and no other timer methods of the Java platform API are discussed. As the 2nd edition of [16] is from 2003, newer methods such as java.lang.System.nanoTime() are not discussed at all. The same is true for [17], which was published in 2000.

In the “Effective Java” book [18], Bloch states that “for interval timing, always use [java.lang.]System.nanoTime in preference to [java.lang.]System.currentTimeMillis. System.nanoTime is both more accurate and more precise, and it is not affected by adjustments to the system’s real-time clock” (p. 276). Also here, it is not explained how this conclusion was reached, and no concrete values are given.

In the remainder of this section, we describe further related work in a top-down manner, from application-level approaches, over third-party tools, virtual machines and operating systems down to hardware.

In [13], Holmes provides an overview of clocks, timers and scheduling events accessible from Java, but does not provide any reusable means to obtain precise characteristics of timer methods. For example, he states (in 2006) that “typically, a Windows machine has a default 10 ms timer interrupt period, but some systems have a 15 ms period”. At the same time, our measurements in 2008 on a machine running Windows XP on a Intel dual-core processor show that the accuracy of Java’s nanoTime() is better than a microsecond, which means that “better” timers are used by the JVM in newer versions.

In [3], Meyerhofer describes time measurements from and within Java on a variety of operating systems and platforms. He computes the accuracy of currentTimeMillis() in Java using an algorithm that does not consider the effects of the timer invocation cost and hence would not be applicable to the nanoTime() timer method or other fine-granular timers where the invocation costs are larger than the accuracy. He also does not account for the effects of just-in-time compilation.

In [4], Danzig and Melvin describe how to measure time intervals that are shorter than the precision of available timers (in their case, the precision corresponds to the accuracy of the hardware clocks they use). In [4], the authors assume that the clock accuracy/resolution (i.e. timer resolution) is known, and disregard the cost of timer invocations. They compute the number of measurements needed to achieve a given confidence level for a given number of significant digits, using statistical techniques and approximations. Our paper presents an approach to compute the timer precision on which [4] relies.

In [5], Beilner describes a stochastic measurement technique and corresponding statistical evaluation that are applied to sub-accuracy operations in a distributed, message-based system; however, Beilner has to guess the (smallest) duration of the operations to be measured. In [6], Lambert and Power build on [4] and [5] to obtain platform-independent timings of Java Virtual Machine bytecode instructions, using the RDTSC (read time stamp counter) instruction of the Intel Pentium processors. However, they also do not try to obtain the accuracy or the invocation cost of RDTSC calls.

In [11], Browne et al. introduce PAPI, a “portable programming interface for performance evaluation on modern processors”. The purpose of the PAPI project is to “specify a standard application programming interface (API) for accessing hardware performance counters”. However, PAPI does not offer any means to query the accuracy or the invocation cost of the timer methods it provides. Similar interfaces to hardware or operating system timers are PCL [12], JETM [1]
and GAGEtimer [2], but none of them provides information on both accuracy and invocation costs.

Information about invocation cost and accuracy of timers in virtual machines, operating systems and assembler-accessible hardware counters is also usually unknown. Some timers of particular vendors, such as the undocumented `sun.misc.Perf` timer in Sun Microsystems’ JVM, offer methods to query the accuracy of the counter in ticks per second, but still do not document the invocation costs.

VII. CONCLUSIONS

In this paper, we have presented TIMERMeter, a novel, easily portable algorithm for quantifying accuracy and invocation cost of timer methods in a transparent way. We provided the results of running TIMERMeter in a wide array of settings to demonstrate the significant differences between timer methods, and the quantitative importance of timer method call costs. Our results show that the accuracy of the same timer method on the same hardware can differ by up to an order of magnitude, depending on the operating system. For example, we have demonstrated that the widely used `nanoTime()` Java platform API timer method performs differently than expected: it has an accuracy of only 70 ns or 279 ns (depending on the hardware), with an invocation cost overhead of 978 ns up to 1676 ns.

Researchers and developers benefit from using TIMERMeter when they need to obtain accuracy and invocation cost of timer methods. This is often the case while performing reliable and statistically sound measurements, for example in microbenchmarking and during fine-granular measurements.

TIMERMeter treats the timer methods as black boxes, i.e. it does not try to analyse their implementation to make statements about a timer method’s behaviour and characteristics. This is because timer methods must, at a deeper layer, access hardware timers (or counters), and this access happens in a platform-dependent way, while the accessed counters’ behaviour is not known and in general cannot be queried. Hence, decomposing a timer method’s implementation would delegate the problem and would not solve it - in contrast to that, TIMERMeter obtains the accuracy and the invocation cost empirically, in a transparent and platform-independent way. Thus, TIMERMeter is applicable to any kind of absolute and relative timer, independent of the underlying hardware or software stack. It does not require modifications of the execution platform, and it can also be easily ported to other object-oriented or procedural languages.

We have evaluated the applicability and the benefit of our approach using a Java implementation of TIMERMeter, and provide an extensive discussion of the obtained results. In the evaluation, we applied TIMERMeter to the timer methods provided by the Java SE platform API and additionally other timers accessible from Java, including hardware and software timers, as well as to third-party timing tools.

Our results have been successfully used for high-precision measurements in bytecode benchmarking [19] and bytecode-based performance prediction [20]. In the future, we plan to implement TIMERMeter in C# for use on the virtual machine of the .NET framework.

ACKNOWLEDGMENT

The authors would like to thank Petr Tuma, Thomas Moschiny, Lubomir Bulej and Vladimir Mencl for their useful hints about timers and measurements. We also appreciate the comments and suggestions of Klaus Krogmann, Samuel Kounev, Viktoria Firus, Anne Martens, Erik Burger, Fabian Brosig and all other members of the Palladio research group.

REFERENCES