Metric-based Selection of Timer Methods for Accurate Measurements

Michael Kuperberg
Karlsruhe Institute of Technology
Am Fasanengarten 5
76131 Karlsruhe, Germany
michael.kuperberg@kit.edu

Martin Krogmann
Karlsruhe Institute of Technology
Am Fasanengarten 5
76131 Karlsruhe, Germany
krogmann@kit.edu

Ralf Reussner
Karlsruhe Institute of Technology
Am Fasanengarten 5
76131 Karlsruhe, Germany
reussner@kit.edu

ABSTRACT

Performance measurements are often concerned with accurate recording of timing values, which requires timer methods of high quality. Evaluating the quality of a given timer method or performance counter involves analysing several properties, such as accuracy, invocation cost and timer stability. These properties are metrics with platform-dependent values, and ranking and selecting timer methods requires comparisons using multidimensional metric sets, which make the comparisons ambiguous and unnecessary complex. To solve this problem, this paper proposes a new unified metric that allows for a simpler comparison. The one-dimensional metric is designed to capture fine-granular differences between timer methods, and normalises accuracy and other quality attributes by using CPU cycles instead of time units. The proposed metric is evaluated on all timer methods provided by Java and .NET platform APIs.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems; D.2.8 [**Software**]: Software Engineering–Metrics

General Terms

Performance, timer selection

Keywords

Timer Method, Performance Counter, Accuracy, Resolution, Invocation Cost, Quality Metric, Precision, Granularity

1. INTRODUCTION

For measuring timing values and time intervals, *timer methods* are provided by APIs of performance counter libraries, operating systems, virtual machines, etc. When using timer methods to perform fine-granular or accuracy-sensitive measurements, scientists need to select suitable timer methods among the available ones. For example, to measure an operation that takes 250 ns, a timer method that uses a counter which is updated once every 15 ms is not appropriate.

However, the accuracy of timer methods depends on the underlying hardware (e.g. performance counters), whose accuracy is often platform-specific and unknown. Published values as in [11] or [15] are mostly platform-specific, vague, outdated and provided without the code that produced them, so it is not possible to transfer these results to other hardware/software platforms without re-running the original code. In addition to accuracy, other quality attributes (e.g. invocation cost) also play a significant role and need to be considered.

Recently, techniques for quantifying the accuracy and invocation cost of black-box timer methods have been developed in the form of the TimerMeter approach [10]. However, TimerMeter provides the user with several metrics and often, a timer method is better than another in one metric and worse in another. To make it easier for humans to compare and rank timer methods, it is needed to unify these metrics into one well-designed quality metric. Also, in the context of measurements requiring high accuracy, it makes sense to compare timer methods not only on one platform, but also across platforms. Comparing accuracy and other metrics needs to account for platform differences, as shown by the following example: an accuracy of 1000 ns on a CPU with 1 GHz is *better* than an accuracy of 700 ns on a CPU with 2 GHz, because the first accuracy value corresponds to 1000 CPU cycles and the second to 1400 CPU cycles.

The contribution of this paper is a novel quality metric for timer methods that allows for dependable and precise comparison and ranking of timer methods, even across execution platforms. The metric unifies and encapsulates several quality metrics, such as accuracy, invocation cost and invocation cost spread. The metric value range is normalised to $[0.00, 1.00]$, which allows to interpret the quality values as percentage values. The design of the metric computation formula ensures that even the finest differences of the input metrics are captured and can be expressed using at most two decimal places of the metric value in %.

The properties and advantages of the introduced metric are evaluated by computing the quality metric values for eight different timer methods on four execution platforms that differ significantly in hardware and software. The results of the evaluation demonstrate how platform-dependent timer method quality values are, and show that there is no

ICPE'11, March 14–16, 2011, Karlsruhe, Germany.
Copyright 2011 ACM 978-1-4503-0519-8/11/03 ...$10.00.

clear universal winner among the timer methods. The studied timer methods include all methods for measuring timing values that are available on the Java and .NET platform APIs.

The remainder of this paper is structured as follows: Section 2 discusses the foundations, incl. the computation of the metric inputs using the TimerMeter approach. Section 3 presents related work. Section 4 introduces the new timer method quality metric, and explains its design decisions. Section 5 applies the metric to 8 timer methods on 4 platforms, and discusses the results as well as conclusions we have made from them. Section 6 concludes with the discussion of the presented approach and future work.

2. FOUNDATIONS OF TIMER METHODS

A *timer method* is a software method that accesses a hardware *timer*, i.e. a counter which is incremented at regular intervals by a non-negative constant value. An example of a periodic counter is the HPET (High-Performance Event Timer) [1]. The counter's value can be converted to timing values when the interval between two subsequent increments is known. When reading the counter's value, the timer method reads the last (i.e. most recent) value of the counter – this is hinted by the dashed line in Figure 1.

Accuracy is described by Lilja [12, p. 44] as "the absolute difference between a measured value and the corresponding reference value". As Figure 1 shows, the distance between the real timing value and the measured value (based on counter reading) is up to one counter update interval.

Thus, for timer methods, [10] equates *timer method accuracy* with the resolution of the underlying counter, and uses Lilja's definition of resolution as the "smallest incremental change that can be detected and displayed". Note that the timer method accuracy (and the counter resolution on which it is based) are different from a timer method's unit. For example, the unit of `java.lang.System.nanoTime()` is 1 ns, although in practice, its resolution (and resulting accuracy of measurements) are often hundreds of ns [8, 10]. The TimerMeter approach quantifies the timer method accuracy as it is seen by the application which invokes the timer method.

Invocation cost of a timer method is a synonym for *execution duration* of that timer method and spans the interval from the timer method invocation until it returns a value, as seen by the method's invoker. The invocation cost can be smaller than the accuracy or larger than it. The invocation cost may vary from call to call due to CPU interrupts, OS scheduling and other runtime influences, as well as due to Just-in-Time compilation.

When calculating the average invocation cost, outliers (e.g. caused by Garbage Collection of Java Virtual Machines) can bias the obtained value very significantly. Instead of using the average, the median value can be used as it captures the "normal" case more accurately. Still, the "spread" of invocation costs observed in practice should be captured in a metric that quantifies the platform-specific quality of a timer method, as this spread impacts the statistic validity of measurements.

For calculating the platform-specific accuracy and invocation cost of timer methods, this paper uses the TimerMeter approach [10]. This approach treats the timer method implementation as a black box, i.e. it does not analyse its implementation and does not require information about the elements of the execution platform. The most interesting part of TimerMeter is the quantification of accuracy and its main principle is sketched in the remainder of this section; for further details, the reader is referred to [10].

TimerMeter works both in the case where the invocation cost of the timer method is larger than its accuracy, and in the case where it is smaller. Both cases occur frequently, and a given timer method can fall in different cases on different platforms; TimerMeter can recognise which of the two cases applies for a given method in a given environment. We only outline the approach for the first case, and a visual explanation for it is shown in Figure 1 and explained in the following. The considered timer method is denoted as `time()`, and it reads the last value of the underlying counter/timer (hinted by the dashed line), processes it and then returns it.

As the upper part of Figure 1 shows, invoking two timer methods in a row can return a time interval of length 0 when the timer method invocation cost is smaller than its accuracy. To obtain a time interval of non-zero length, TimerMeter inserts a small amount of work between timer method invocations and re-measures the time interval. The work amount is increased until the measured interval "jumps" from 0 to a non-zero value which corresponds to the value of timer method accuracy. The work amount is further increased to obtain further jumps (i.e. 2·accuracy, 3·accuracy, etc.), and the measurements are sorted and clustered to address outliers and other challenges.

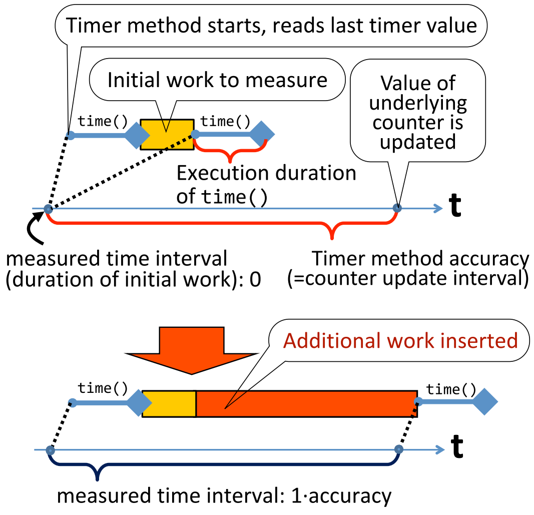
Timer method starts, reads last timer value
Initial work to measure
Value of underlying counter is updated
time()
time()
Execution duration of time()
t
measured time interval (duration of initial work): 0
Timer method accuracy (=counter update interval)
Additional work inserted
time()
time()
t
measured time interval: 1·accuracy

Figure 1: Quantifying the timer method accuracy (for the case accuracy $>$ invocation cost)

3. RELATED WORK

The question of designing quality metrics that unify other metrics has been researched in software complexity [7], software QoS [2] as well as in computer vision and video quality [5, 16, 17], but there is no general approach or design guidelines beyond these domains. Machine learning approaches such as neural networks, genetic algorithms and others can be used to approximate a function that computes *known* output values from given input values - however, this requires existing results to which the function is fitted, which is not the case for the challenge of timer method quality addressed in this paper. Analytic Hierarchy Process (AHP [14]) or other decision making approaches from operations research require the *user* to state her preferences and assign weights to the individual decision criteria, while the presented unified metric does not need this level of interaction.

Discussions of quality properties of timer methods are mostly limited to accuracy, and only a few publications attempt to quantify it. Books on performance measurement, evaluation and benchmarking (e.g. [9, 12]) discuss the importance of timer accuracy for quantifying the errors in measurements, but do not provide algorithms for *computing* the accuracy or other quality metrics.

Some technology-specific books (e.g. [4, 15] for Java), make statements on the accuracy of two popular Java platform API timer methods, but do not provide any algorithms or explanations on the origin of the accuracy values. In [8], 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".

In [13], Meyerhoefer describes time measurements from and within Java. 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 often larger than the accuracy.

In [6], 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 [6], 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. In [3], 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.

While TimerMeter [10] provides platform-independent algorithms for computing accuracy and invocation cost of timer methods, it does not provide a unified, "one-stop" quality metric, which is the focus of this paper.

4. A UNIFIED TIMER QUALITY METRIC

Most users prefer a *single* metric as a simple way to compare things, instead of using multidimensional metric sets. Therefore, the individual timer method quality properties such as accuracy, invocation cost etc. should be composed to form a new *unified* (i.e. single-valued) and pragmatic metric. The *value range* of the unified quality metric should be intuitively understandable without in-depth knowledge of timer method quality attributes.

A value range of "0 % to 100 %" is fulfilling this requirement, and it is clear that "larger is better" applies to it; it also has an advantage of making the metric unitless. Therefore, this value range (which can also be interpreted as $[0.00, 1.00]$) was defined as a requirement during the development of the metric. The metric *values* should be floating-point numbers rather than integers to allow for a more fine-granular expression of timer method differences.

In the following, we describe a new metric (referenced as $Quality_{timer}$) which was developed according to these considerations.

4.1 Accounting for Different CPU Clock Frequencies

Quality properties of timer methods are computed from measurements collected at runtime and they are valid for the specific execution platform and the settings in which the measurements were performed. A unified timer quality metric should depend on the properties of the execution platform, in particular on its processing speed.

For example, consider two execution platforms: platform P1 has a 1.0 GHz CPU and platform P2 has a CPU with 2.0 GHz. A timer method that is available on both platforms has an accuracy of 1000 ns on platform P1 and an accuracy of 700 ns on platform P2. At the first glance, the timer method is more accurate on platform P2, yet the timer method accuracy on platform P1 corresponds to 1000 cycles but on platform P2, the timer method accuracy corresponds to 1400 cycles.

For an algorithm implementation which takes a constant number of cycles to execute independent of a concrete CPU and platform, measurements should be done on platform P1 rather than on platform P2, as the timer accuracy will result in lesser measurement error on P1 than on P2. Not only accuracy, but also the timer method invocation cost should be expressed in CPU cycles, rather than in time units.

Based on the fact that the smallest unit of time-related measurements is 1 CPU cycle, the following discussion presumes that the minimum value of accuracy and invocation cost is 1 CPU cycle. We assume that the CPU frequency of the execution platform on which the measurements were performed remained constant *over the course of the measurements*, and therefore the effective CPU processing speed remained constant as well.

Of course, modern CPUs provide dynamic (runtime) frequency scaling, which means that the CPU frequency adapts to the CPU load. So far, our observations have shown that a changed CPU frequency does not alter the accuracy values (in time units) of timer methods, while invocation costs of timer methods (in time units) do change proportionally.

This behavior can be explained by the fact that many hardware counters operate independently of the CPU frequency (e.g. HPET [1]). For example, when decreasing the CPU frequency of a Core 2 Duo processor (platform **MBP53** in Section 5) from 2793 MHz to 1596 MHz ($\frac{1596}{2793} = \frac{4}{7} \approx 0.571$), the accuracy of `System.nanoTime` (cf. Table 1) remains at 1000 ns. However, its invocation costs increase from 97 ns to 172 ns, i.e. proportionally ($\frac{97}{172} \approx 0.564$). Further research is needed to study the impacts of CPU frequency and CPU *voltage* on timer methods.

4.2 Designing the Unified Quality Metric

$Quality_{timer}$ is computed by Formula (1), which shows that it is a product of three elements, and each element is in the range $[0.0, 1.0]$, as explained below.

$$Quality_{timer} := accuracy^{-0.1} \cdot invocationCost_{median}{}^{-0.1} \cdot invocationCostSpread^{0.5} \quad (1)$$

The first element of Formula (1) is based on timer method accuracy, for which it holds that "smaller value is better". Yet for the quality metric to design, it applies that "larger value is better" – thus, a *negative* exponent is chosen (next section explains why -0.1 was chosen over -1). Since $1 \leq accuracy$, it holds that $0 < accuracy^{-0.1} \leq 1$. The accuracy value is expressed in CPU cycles (with the minimum value being 1) and not in conventional time units such as nanoseconds for the above reasons; the unit is dropped to make $Quality_{timer}$ unitless.

The second element of Formula (1) is based on the timer method invocation cost, again with minimum value of 1 CPU cycle. As with accuracy, "smaller value is better" applies to invocation cost. While there is a *minimal* invocation cost, the execution duration of the timer method varies from invocation to invocation, and the *median* invocation cost is a realistic statistic for the majority of samples. For the same reasons as for accuracy, invocation costs are expressed in CPU cycles, the units are dropped and a negative exponent is chosen; it holds that $0 \leq invocationCost_{median}{}^{-0.1} \leq 1$. Since the second element of Formula (1) uses the *median* invocation cost, Formula (1) needs to express how the entirety of all recorded invocation cost values is spread around the median invocation cost. This need is addressed by the next element in Formula (1).

The third element of Formula (1) is called *invocationCostSpread* and it is based on the percentage of invocation cost values (samples) within ±1 *accuracy* of the median invocation cost. To make *invocationCostSpread* have the value range $[0.00, 1.00]$, the percentage values are divided by 100%. For *invocationCostSpread*, it holds that "larger value is better", since the less invocation cost samples are too far away from the median, the easier it is to capture the timer method overhead. *invocationCostSpread* will never become 0 as long as there is at least one sample invocation value and therefore also a median invocation cost which makes the aforementioned percentage non-zero. The definition of *invocationCostSpread* allows it to become 1.00 even if the invocation cost varies between samples – as long as all samples remain within ±1 *accuracy*. Note that the distribution of timer method invocation costs does not follow a Gaussian distribution, and thus we decided not to use standard deviation or variance for expressing the invocation cost spread.

4.3 Choice of the Exponents for the Unified Timer Quality Metric

The choice of decimal-point exponents for the first two contributions is motivated by the range of the raw values *accuracy* and $invocationCost_{median}$. For example, assume that we were using exponents -1, -1 and $+1$, i.e. $Quality_{timer}$ were computed as $accuracy^{-1} \cdot invocationCost_{median}{}^{-1} \cdot invocationCostSpread^{1}$. Then, for a timer method with 1 ms accuracy, 200 ns invocation cost and invocation cost spread of 1.0 on a CPU running at 2 GHz, it would have resulted in a metric value of $\frac{1}{2{,}000{,}000} \cdot \frac{1}{200} \cdot 1.0 = 0.0000000025 \mathrel{\widehat{=}} 0.00000025\ \%$, which is a very small value compared to the range $[0.0, 1.0]$. For another timer method with a smaller invocation cost of 100 ns instead of 200 ns (and same values otherwise, on the same machine), the formula with the trivial exponents would yield 0.000000005. While the values are clearly different (by the factor of 2), they are hard to compare because they are too small.

With the exponents in Formula (1), things look differently and better for these two timers: $Quality_{timer}$ is ≈ 13.79 % for the first timer and $\approx$ 14.79 % for the second timer. The quality values do not differ by the factor of two anymore, but this is an advantage: since the (identical) accuracy is rather poor, the differences in invocation cost are not so important anymore, which is made clear by the quality values. In Section 5, the quality values for different timer methods on different platforms will be compared, which will add further empirical justification to the choice of exponents in Formula (1).

For the invocation spread, the exponent is set to 0.5 to decrease its impact onto the total result; it holds that $0 < invocationCostSpread^{0.5} \leq 1$ since $0 < invocationCostSpread \leq 1$. To see the reasons for adjusting the impact of the spread, consider the following two results (which are real-life values, obtained on the same execution platform): Timer **a** has an accuracy of 2400 CPU cycles, an invocation cost of 4800 CPU cycles, and an invocation cost spread of 0.993. Timer **b** has an accuracy of 168 CPU cycles, invocation cost of 1680 CPU cycles and a spread of 0.578.

For **a**, the resulting quality metric value is $\approx$ 19.60 % for spread's exponent being 0.5 and would be $\approx$ 19.53 % if the exponent were 1.0. For **b**, the quality metric value is $\approx$ 21.67 % for exponent 0.5 but would be $\approx$ 16.48 % for exponent 1.0. Despite its higher spread, **b** is more accurate and causes less overhead: thus, its quality should be *higher* than that of **a** – this is the case when the exponent of the spread's contribution is 0.5 but is not the case when the exponent is 1.0. This small example illustrates the need to decrease the impact of the spread – still, note that the choice of the concrete exponent value has no formal underpinning.

4.4 Further Considerations

In Formula (1), all three contributions are within the interval $(0.0, 1.0]$, and thus so is their product. This allows to "reserve" the value 0.0 of $Quality_{timer}$ for special use: $Quality_{timer} = 0.0$ iff the timer method is non-monotonic, unstable, not thread-safe or a combination thereof (the identification of such problems is outside the scope of this paper). In all other cases, $Quality_{timer} > 0.0$.

Another advantage of the presented choice of exponents is that values of $Quality_{timer}$ are not too small. Consider the following "worst-case" scenario where a timer has an accuracy of 15 ms (i.e. 15,000,000 ns) and a median invocation cost of 16 μs, with the CPU running at 4.0 GHz. Such a coarse accuracy was in fact observed for `java.lang.System.currentTimeMillis()` on Windows XP computers. An invocation cost of 16 μs would correspond to 64,000 CPU cycles on the 4 GHz CPU, which is also a rather high value, though invocation costs of 47,709 CPU cycles have in fact been found for `java.lang.management.ThreadMXBean.currentThreadCpuTime()` on modern machines (Core 2 Duo CPU) running Linux for the `getCurrentThreadCpuTime` method of the Java API class `ThreadMXBean`.

The worst-case scenario assumes an invocation spread of 0.3, although until now, we did not observe values below 0.5 in practice. The value of $Quality_{timer}$ for the worst case scenario is calculated from timing values using the relation that 1 ns correspond to 4 CPU cycles on a 4 GHz CPU. Thus, $Quality_{timer} = (4{*}(15{*}10^6))^{-0.1} \cdot (4{*}(16{*}10^3))^{-0.1} \cdot 0.3^{0.5} \approx 0.1668 \cdot 0.3307 \cdot 0.5477 \approx 0.03021 \equiv 3.02\%$. Thus, while the value of $Quality_{timer}$ is very low, it still can be expressed as *integer-typed* percentage, i.e. even rounding would not make it 0.0.

5. EVALUATION

Tables 1 and 2 show the values of quality attributes and the quality metric for four different execution platforms. For the validation, we selected these execution platforms so that we could study the impacts of differences in hardware characteristics and operating system in isolation:

1. **MBP53**: a MacBook Pro notebook (model identifier "MacBookPro5,3") with 2.8 GHz Intel Core 2 Duo CPU (T9600), 4 GB of RAM, running Mac OS X 10.6.4 and Apple JVM (JDK 1.6.0_21) / Mono 2.6.7.
2. **MBP62**: a MacBook Pro notebook (model identifier "MacBookPro6,2") with 2.66 GHz Intel Core i7 CPU, 8 GB of RAM, running Mac OS X 10.6.4 and Apple JVM (JDK 1.6.0_21) / Mono 2.6.7.
3. **SAMSa**: a Samsung notebook with Intel Pentium M 1.73 GHz CPU, 1 GB of RAM, running openSUSE Linux with Kernel 2.6.34, and Oracle JDK 1.6.0_20 / Mono 2.6.7
4. **SAMSb**: same as SAMSa, but running Windows XP Professional and Oracle JVM (JDK 1.6.0_21) and .NET 4.0 in addition to Mono 2.6.7

The validation covers all timer methods provided by Java SE and .NET Platform APIs:

- `CTCT` is `java.lang.management.ThreadMXBean.getCurrentThreadCpuTime()`, a method which returns the calling thread's used CPU time in nanoseconds
- `CTM` is `java.lang.System.currentTimeMillis()`, a static wall-clock timer method with milliseconds as units
- `CTUT` is `java.lang.management.ThreadMXBean.getCurrentThreadUserTime()`, a method which returns the time a thread has spent in user mode
- `HRC` is `sun.misc.Perf.highResCounter()`
- `NANO` is `java.lang.System.nanoTime()`, a static wall-clock timer method with nanoseconds as units
- `PCT` is `com.sun.management.OperatingSystemMXBean.-getProcessCpuTime()` or `com.sun.management.UnixOperatingSystemMXBean.getProcessCpuTime()`, depending on the JVM
- `.DAT`: .NET API's `DateTime.Now` structure in the `System` namespace
- `.STO`: .NET API's start/stop methods in the `StopWatch` class (`System.Diagnostics` namespace)

Several observations can be made on the basis of Tables 1 and 2. The .NET methods `.DAT` and `.STO` have identical quality on Mac OS X platforms **MBP53** and **MBP62**, but are slightly different on Linux (**SAMSa**) and very different on Windows (**SAMSb**). Also note the large differences of their accuracy values on the same hardware depending on the operating system: 156,250 ticks on Windows but 10 ticks on Linux, i.e. more than five orders of magnitude. On **SAMSb**, the `.NET` timer method `.STO` has better quality than any Java SE platform API timer method.

Comparing Java timer methods, none of them has the best quality on all four platforms, though `HRC` has highest quality on all platforms but **SAMSb**, where its difference to `NANO` is less than 4 percentage points. The invocation cost differs by more than three orders of magnitude between the fastest (`HRC` on **MBP62**, 70 ns) and the slowest (`CTCT` on **SAMSa**, 30000 ns as confirmed by repeated measurements). The invocation cost spread decreases with the accuracy.

Quality metric values for `CTCT` and `CTUT` are very close on individual platforms, although they differ across platforms and are significantly better on **MBP53** and **MBP62**. `PCT` has a low quality on all four platforms, due to its unexpectedly bad accuracy. Same observation holds for `CTCT` and `CTUT` on **SAMSa** and **SAMSb**. Note that the quality metric captures even finest differences: on **MBP53**, `CTCT` quality is 18.8**6** % and and `CTUT` quality is 18.8**8** % due to a very slight difference of median invocation costs.

For the same hardware running different operating systems and JVMs (**SAMSa** and **SAMSb**), the quality is better on Windows (**SAMSb**) for one half of the timer methods, and better on Linux (**SAMSa**) on the other. For the same OS and JVM on different hardware, the quality metric's values are substantially different (cf. **MBP53** vs. **MBP62**).

Of course, there exist scenarios in which only one quality attribute (and not the entire metric) are important. For example, if a timer method is called very frequently but high accuracy is not needed, the person selecting the timer method to use needs to "drill down" and investigate the constituents of $Quality_{timer}$ in addition to the metric.

Finally, it should be noted that due to the use of exponents in $Quality_{timer}$, a difference of 5 % has different meanings depending on the compared values. For example, the difference between 15 % and 20 % has a different weight than the difference between 55 % and 60 % .

6. CONCLUSION

This paper has devised a novel quality metric for timer methods, which simplifies comparison and selection of timer methods to be used for accurate and precise measurements. The developed quality metric unifies the quality attributes accuracy, invocation cost and invocation cost spread into one value with the convenient range $[0.0, 1.0]$, i.e. [0 % , 100 %]. The design of the metric ensures that these attributes are balanced, and it accounts for the impact of hardware characteristics such as CPU clock frequency.

The metric has been evaluated on all timer methods of the Java SE and .NET 4.0 platform APIs, including methods which measure "thread time", "process time" and "user time", i.e. not only wall-clock time. In future work, we plan to apply the presented approach to countdown and periodic timers, as well as to timer methods provided by operating systems and to hardware performance counters. Additionally, the impact of dynamic CPU frequency scaling onto the devised metric needs to be studied.

A further challenge that we plan to address in future work is brought by the increased popularity of virtualisation: if the virtualiser/hypervisor must emulate the CPU and its counters/registers, the quantitative properties of the emulated CPU (update frequency of counters, etc.) can differ

Timer	Accuracy	Cost	Spread	Quality
Execution platform MBP53 (1 ns = 2.8 CPU cycles)				
`CTCT`	1,000 ns	2,232 ns ⋆	0.999	18.86 %
`CTM`	1 ms	101 ns ⋆	1.000	12.89 %
`CTUT`	1,000 ns	2,204 ns ⋆	0.999	18.88 %
`HRC`	3 ticks ⋄	51 ticks ⋄	0.778	54.08 %
`NANO`	1,000 ns	97 ns ⋆	1.000	25.82 %
`PCT`	10,000,000 ns	2,298 ns ⋆	1.000	7.49 %
`.DAT`	10 ticks ♣	2 ticks ♣	1.000	24.01 %
`.STO`	10 ticks ♣	2 ticks ♣	1.000	24.01 %
Execution platform MBP62 (1 ns = 2.66 CPU cycles)				
`CTCT`	1,000 ns	1,756 ns ⋆	0.983	19.36 %
`CTM`	1 ms	70 ns ⋆	1.000	13.51 %
`CTUT`	1,000 ns	1,643 ns ⋆	0.984	19.50 %
`HRC`	1 tick ⋄	36 ticks ⋄	0.648	41.44 %
`NANO`	1,000 ns	70 ns	1.000	26.95 %
`PCT`	10,000,000 ns	1,712 ns ⋆	1.000	7.79 %
`.DAT`	10 ticks ♣	2 ticks ♣	1.000	24.26 %
`.STO`	10 ticks ♣	2 ticks ♣	1.000	24.26 %

Table 1: Timer quality metric for execution platforms MBP53 and MBP62 (Legend: ⋆: invocation cost measured using `System.nanoTime()` method; ⋄: 1 tick = 1 ns; ♣: 1 tick = 100 ns.)

from the "real" one. This reaffirms the need to quantify the value of timer method quality metrics in a given scenario, rather than to use look-up tables. Finally, we plan to investigate more refined measures for invocation cost spread, for example by identifying the type and parameters of stochastic distribution of the observed method's invocation costs.

The authors would like to thank Jörg Henß, Samuel Kounev, Klaus Krogmann, Anne Koziolek, Omar-Qais Noorshams, Mircea Trifu, Erik Burger and the anonymous reviewers for their valuable comments and suggestions.

7. REFERENCES

[1] Intel 82801EB I/O Controller Hub 5 (ICH5) Datasheet, 2003. http://www.intel.com/Assets/PDF/-datasheet/252516.pdf, last visit: Jan 3rd, 2011.

[2] P. Alipio, S. R. Lima, and P. Carvalho. A unified metric for quality of service quantification. In *Simutools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 1–7, ICST, Brussels, Belgium, Belgium, 2009. ICST.

[3] H. Beilner. Measuring with Slow Clocks. Technical report, ICSI-Technical Report-88-O03, 1988.

[4] J. Bloch. *Effective Java*. Addison-Wesley Professional, 2. edition, 2008.

[5] L. Boroczky and Y. Yang. Artifact reduction for MPEG-2 encoded video using a unified metric for digital video processing. In *Proceedings of SPIE*, volume 5150, page 1390, 2003.

[6] P. B. Danzig and S. Melvin. High Resolution Timing with Low Resolution Clocks and Microsecond Resolution Timer for Sun Workstations. *ACM SIGOPS Operating Systems Review*, 24(1):23–26, 1990.

[7] R. Gonzalez. A unified metric of software complexity: Measuring productivity, quality, and value. *Journal of Systems and Software*, 29(1):17–37, 1995.

Execution platform SAMSa (1 ns = 1.73 CPU cycles)				
Timer	Accuracy	Cost	Spread	Quality
`CTCT`	10,000,000 ns	30,000 ns ⋆	0.999	6.37 %
`CTM`	1 ms	1,267 ns ⋆	1.000	11.02 %
`CTUT`	10,000,000 ns	8,000 ns ⋆	1.000	7.28 %
`HRC`	1 tick ⋄	1,283 ns ⋆	0.999	21.95 %
`NANO`	69 ns	978 ns ⋆	0.736	25.29 %
`PCT`	10,000,000 ns	555 ns ⋆	1.000	9.51 %
`.DAT`	10 ticks ♣	10 ticks ♣	0.996	22.47 %
`.STO`	1 tick ♣	11 ticks ♣	0.944	27.27 %
Execution platform SAMSb (1 ns = 1.73 CPU cycles)				
`CTCT`	15,625,000 ns	896 ns ⋆	1.000	8.66 %
`CTM`	16 ms	127 ns ⋆	1.000	10.51 %
`CTUT`	15,625,000 ns	889 ns ⋆	1.000	8.67 %
`HRC`	1 tick ■	5 ticks ■	0.999	24.74 %
`NANO`	279 ns	1,876 ns	0.997	24.00 %
`PCT`	15,625,000 ns	476 ns ⋆	1.000	9.22 %
`.DAT`	156,250 ticks ♣	8 ticks ♣	1.00	8.76 %
`.STO`	1 tick ♣	5 ticks ♣	0.992	30.25 %

Table 2: Timer quality metric for execution platforms SAMSa and SAMSb (Legend: ⋆: invocation cost measured using `System.nanoTime()` method; ⋄: 1 tick = 1,000 ns, calculated from frequency; ♣: 1 tick = 100 ns; ■: 1 tick = $\frac{1}{3579545}$ s ≈ 279 ns.)

[8] D. Holmes. Inside the Hotspot VM: Clocks, Timers and Scheduling Events, 2006. http://blogs.sun.com/dholmes/entry/, last visit: Jan 3rd, 2011.

[9] L. K. John and L. Eeckhout. *Performance Evaluation And Benchmarking*. CRC Press, 2006.

[10] M. Kuperberg, M. Krogmann, and R. Reussner. TimerMeter: Quantifying Accuracy of Software Times for System Analysis. In *Proceedings of the 6th International Conference on Quantitative Evaluation of SysTems (QEST) 2009*, 2009.

[11] C. Larman and R. Guthrie. *Java 2 Performance and Idiom Guide*. Prentice Hall PTR, 2000.

[12] D. J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.

[13] M. Meyerhöfer. *Messung und Verwaltung von Komponenten für die Performancevorhersage*. PhD thesis, University of Erlangen-Nürnberg, Germany, 2007.

[14] T. Saaty. *Multicriteria decision making: the analytic hierarchy process: planning, priority setting, resource allocation*. RWS publications Pittsburgh, 1990.

[15] J. Shirazi. *Java Performance Tuning*. O'Reilly, 2 edition, 2003.

[16] N. Venkatasubramanian and K. Nahrstedt. An integrated metric for video QoS. In *Proceedings of the fifth ACM international conference on Multimedia*, page 380. ACM, 1997.

[17] Y. Yang and L. Boroczky. A unified metric for digital video processing and its applications in home video systems. In *2003 IEEE International Conference on Consumer Electronics, 2003. ICCE*, pages 338–339, 2003.