Analysing the Fidelity of Measurements Performed with Hardware Performance Counters

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

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

ABSTRACT

Performance evaluation requires accurate and dependable measurements of timing values. Such measurements are usually made using timer methods, but these methods are often too coarse-grained and too inaccurate. Thus, direct usage of hardware performance counters is frequently used for fine-granular measurements due to higher accuracy. However, direct access to these counters may be misleading on multicore computers because cores can be paused or core affinity changed by the operating system, resulting in misleading counter values. The contribution of this paper is the demonstration of an additional, significant flaw arising from the direct use of hardware performance counters. We demonstrate that using JNI and assembler instructions to access the Timestamp Counter from Java applications can result in grossly wrong values, even in single-threaded scenarios.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems; D.2.5 [**Software**]: Software Engineering–Testing and Debugging

General Terms

Performance, timer selection

Keywords

Timer Method, Performance Counter, Fidelity, Dependability, Accuracy, Timestamp Counter, TSC

1. INTRODUCTION

Timer methods are provided by APIs of operating systems and virtual machines (e.g. JVM), and also by third-party libraries. When using timer methods to perform fine-granular or accuracy-sensitive measurements, scientists already have the tools [5] to choose among the timer methods on the basis of accuracy and invocation costs (which vary significantly).

However, timer methods are often too coarse for measuring short (nanosecond-level) durations while the underlying hardware offers more accurate facilities. Thus, hardware performance counters such as the TSC [4] (Timestamp Counter) are used alongside available timer methods. Unlike timer methods provided by platform APIs, third-party methods for accessing hardware performance counters are not tested for correct functioning in the light of dynamic frequency scaling, CPU core affinity changes etc. While published issues such as the TSC drift/instability [1] are concerned with concrete cases on individual execution platforms, there exist no vendor-independent test cases to assess dependability of self-written, counter-based timer methods. Additionally, most performance counter users assume that no problems will occur while measuring single-threaded workloads.

The contribution of this paper is a first step towards a platform-independent approach for testing the suitability of hardware performance counters for measuring time intervals. The developed approach is evaluated on different execution platforms using Java Native Interface access to the TSC hardware performance counter.

The remainder of this paper is structured as follows: Section 2 presents the setup of the proposed test scenario. Section 3 applies its Java implementation to the Timestamp Counter (TSC) on different execution platforms and shows that the TSC is not dependable and is impacted by the use of standard methods from the Java platform API. Section 4 concludes and discusses future work.

2. EXPERIMENT SETUP

The main idea of the test scenario is to measure the same task execution both with an existing, proven timer method and using the direct access to a hardware performance counter. The two measurements are compared to each other, as captured in the following pseudocode:

```
time1 = firstTimer();  //proven API timer method
time2 = secondTimer(); //new, based on HW counter
workload();            //single-threaded task
time3 = firstTimer();
time4 = secondTimer();
durationFirst = time3 - time1;
durationSecond = time4 - time2;
```

Let `firstTimer()` be a proven, platform-provided API timer method, for example `System.nanoTime()` in Java. In contrast to `firstTimer()`, `secondTimer()` is a self-written method accessing a hardware performance counter, and `secondTimer()` needs to be tested for dependability.

Assume that `workload()` does not start separate threads or processes, and that only one thread is executing the above code. Also assume that no other concurrent accesses to `firstTimer()` or `secondTimer()` will be taking place.

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

Then, the difference between `durationFirst` and `durationSecond` will only be dictated by the accuracies and invocation costs of `firstTimer()` and `secondTimer()`, as well as external disturbances (e.g. interrupts) in executing the above listing. Configuring the duration of `workload()` to be large enough allows to ignore the accuracies and invocation costs of `firstTimer()` and `secondTimer()`. Thus, *uninterrupted* executions of the example should lead to `durationFirst` and `durationSecond` being very close to each other.

3. EVALUATION

For evaluation, `java.lang.System.nanoTime()` served as `firstTimer()`; its accuracy is at most 1000 ns [5]. A JNI implementation [2] of Java access to the Timerstamp Counter (TSC) served as `secondTimer()`. As `workload()`, the `java.util.Thread.sleep(long)` method was used. TSC monotonicity on every platform has been confirmed separately through repeated invocations in a single thread. The execution platform is a computer with Core 2 Duo T9600 CPU (2.8 GHz), running Mac OS X 10.6.4 with Apple JDK 1.6.0_22.

During the evaluation, the requested sleep durations passed as parameter to `Thread.sleep()` started at 20 ms and were increased in steps of 10 ms to 160 ms. For each requested sleep duration, 20 repetitions were made, resulting in a total of 300 measurements. As expected, the values measured with `firstTimer()` (i.e. `nanoTime()`) were virtually identical to the requested sleep times and are therefore used as reference values for analysing TSC-based measurements.

Yet for TSC-based measurements, the outcome is a negative surprise, as shown by Fig. 1: the zigzagged blue line connects the 300 measurements and shows that the TSC-measured sleep times vary significantly between samples with the *same* requested sleep time. Even worse, the TSC-measured values (which are CPU ticks [4]) are significantly *below* the number of CPU ticks that correspond to the requested sleep time (shown as the straight red line in Figure 1; 1 tick=2.8 ns).

This means that the TSC cannot be used for dependable (let alone accurate) time interval measurements. Since the TSC-measured values are too *small*, it appears that the TSC failure is not related to OS scheduling or execution interruptions, but rather related to the usage of `java.util.Thread.sleep` for `workload()`.

Therefore, we have replaced `Thread.sleep` with single-threaded code that computes Fibonacci numbers (the workload size is then the amount of numbers to compute). After this replacement, TSC produces dependable measurements: `durationFirst` and `durationSecond` values are virtually identical. Also, for the samples with the same Fibonacci problem size, the spread of the values of `durationSecond` became negligible.

We have also observed these `Thread.sleep`-caused problems with the TSC on several other platforms, e.g. on a computer with Intel Pentium M 1.73 GHz CPU, running openSUSE Linux with Kernel 2.6.34.

4. CONCLUSION

In this paper, we have shown that using hardware performance counters for measuring time intervals can lead to grossly wrong values even in a simple, single-threaded scenario. On several studied platforms, the usage of the Java platform API method `java.lang.Thread.sleep` means that JNI-based reading of the Timestamp Counter (TSC) returns TSC values that result in incorrect time intervals. The presented black-box test case detects this issue by comparing the measurements obtained with a proven timer method to those of the considered hardware performance counter.

In the future work, the presented approach should be applied to other performance counters; its simplicity makes it easy to apply it to other programming languages as well. It can also be integrated into tool suites such as LTTng [3] or TimerMeter [5]. Possible extensions of the presented algorithm include the testing in multi-threaded scenarios, and tests involving forced changes of the CPU core affinity.

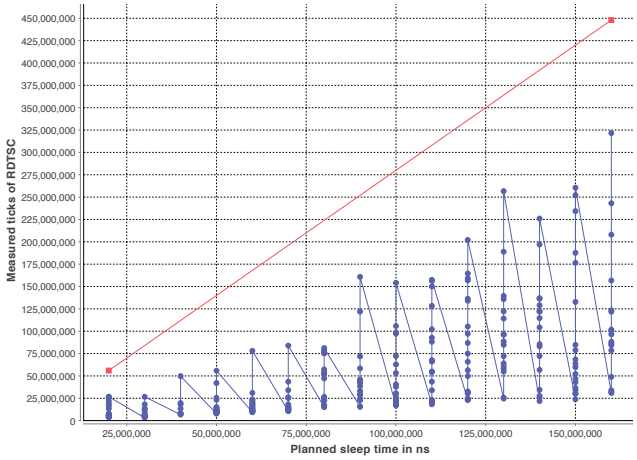

Figure 1: TSC is not dependable: Zigzagged blue line shows the relation between requested sleep times (x-axis, in ns) and values measured with TSC (y-axis, in CPU ticks; 1 tick = 1/2.8 ns); straight red line shows the number of CPU cycles (y-axis) corresponding to the requested sleep time (x-axis)

5. REFERENCES

[1] Bhavana Nagendra (AMD Developer Central). AMD TSC Drift Solutions in Red Hat Enterprise Linux, 2006. http://developer.amd.com/pages/1214200692.aspx.

[2] R. Green. Pentium RDTSC Access using JNI, 2008. http://www.mindprod.com.

[3] P. Heidari, M. Desnoyers, and M. Dagenais. Performance analysis of virtual machines through tracing. In *Canadian Conference on Electrical and Computer Engineering, 2008*, pages 261–266. IEEE, 2008.

[4] Intel. Time Stamp Counter, Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z, Pages 251–252. http://developer.intel.com/design/processor/manuals/253667.pdf.

[5] 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.