UNLV | University Libraries
University of Nevada, Las Vegas

UNLV Theses, Dissertations, Professional Papers, and Capstones

5-1-2019

Rethinking Timestamping: Time Stamp Counter Design for Virtualized Environment

Alexander Tabatadze
tabatadzealexander@gmail.com

Follow this and additional works at: https://digitalscholarship.unlv.edu/thesesdissertations

Part of the Computer Sciences Commons

Repository Citation

Tabatadze, Alexander, "Rethinking Timestamping: Time Stamp Counter Design for Virtualized Environment" (2019). *UNLV Theses, Dissertations, Professional Papers, and Capstones*. 3686.
https://digitalscholarship.unlv.edu/thesesdissertations/3686

This Thesis is protected by copyright and/or related rights. It has been brought to you by Digital Scholarship@UNLV with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis has been accepted for inclusion in UNLV Theses, Dissertations, Professional Papers, and Capstones by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.

RETHINKING TIMESTAMPING: TIME STAMP COUNTER DESIGN FOR VIRTUALIZED ENVIRONMENT

By

Alexander Tabatadze

Bachelor of Science
Russian New University, Russian Federation
2016

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
May 2019

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

April 16, 2019

This thesis prepared by

Alexander Tabatadze

entitled

Rethinking Timestamping: Time Stamp Counter Design for Virtualized Environment

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Jisoo Yang, Ph.D.
Examination Committee Chair

Ajoy Datta, Ph.D.
Examination Committee Member

Kazem Taghva, Ph.D.
Examination Committee Member

R. Jacob Baker, Ph.D.
Graduate College Faculty Representative

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Dean

ABSTRACT

RETHINKING TIMESTAMPING: TIME STAMP COUNTER DESIGN FOR VIRTUALIZED ENVIRONMENT

by

Alexander Tabatadze

Almost every processor supports Time Stamp Counter (TSC), which is a hardware register that increments its value every clock cycle. Due to its high resolution and accessibility, TSC is now widely used for a variety tasks that need time measurements such as wall clock, code benchmarking, or metering hardware usage for account billing.

However, if not carefully configured and interpreted, TSC-based time measurements can yield inaccurate readings. For instance, modern CPU may dynamically change its frequency or enter into low-power states. Also, time spent on scheduling events, system calls, page faults, etc. should be correctly accounted for. Even more complications arise when TSC measurements are done in virtual environments; virtual machines, on which TSC readings are taken, can be suspended, migrated, and scheduled on a machine with different clock rate and performance. In production virtualization systems, some management tasks are executed inside guests on behalf of the management system, effectively consuming end-user's CPU time, which we believe should be excluded from end-user billing.

We argue that the main problem with current TSC is that its hardware semantic is too vague to serve as a multi-purpose time source. In this thesis, we propose an improved TSC

design, called Caviar, to address most of the issues. Caviar extends existing TSC hardware interface by adding a control-register based configuration interface through which a system can set up secondary TSCs whose behavior should be correct when accessed in a localized execution context including virtualized environment.

We experimentally confirmed inaccurate readings with current TSC by conducting a series of TSC measurements on various x86 platforms, including virtualized cloud computing servers. We analyzed some of the results and argue that how our proposed solution can fix the problems. In conclusion, we believe that the simple interface of Caviar can solve most of current TSC complications, be implemented with minimal hardware cost, and be adopted easily by system software.

TABLE OF CONTENTS

ABSTRACT iii

LIST OF FIGURES vi

CHAPTER 1INTRODUCTION 1

CHAPTER 2BACKGROUND 6

CHAPTER 3TSC USAGE FOR DIFFERENT NEEDS 13
Wall-clock measurements 13
CPU performance measurements 13
CPU usage measurements for billing purposes 14

CHAPTER 4HARDWARE PROPOSED SOLUTION 15

CHAPTER 5USAGE OVER HARDWARE PROPOSED SOLUTION 17

CHAPTER 6EXPERIMENTAL RESULT 19

CHAPTER 7SURVEY OF TIME MEASUREMENTS AND TSC 25
Hardware Timers 25
Different Kinds of TSC 27
Timers in Virtual Environment 28
TSC Virtualization 30
Current state of TSC-based measurement 32

CHAPTER 8RELATED WORKS 33

CHAPTER 9CONCLUSION 36

Curriculum Vitae 40

LIST OF FIGURES

2.1 Example of using TSC to measure clock cycles during the execution of a code. . 8
2.2 Results of measurements in system with no load using the code provided in figure 2.1 10
2.3 Measurements in a system under heavy load using the code provided in figure 2.1 11

4.1 Sketch of Hardware proposed Solution 16

5.1 Caviar usage for wall-clock time 17
5.2 Caviar usage for CPU cycle counting 17
5.3 Caviar usage billing 18

6.1 Measurements in a system without any load, expected runtime per iteration 23ms. X-axis - Spent time on 1 iteration, Y-axis - Number of iterations 20
6.2 Measurements in a system under heavy load, expected runtime per iteration 23ms. X-axis - Spent time on 1 iteration, Y-axis - Number of iteration 21
6.3 Measurements in a system without any load, expected runtime per iteration 2.2ms. X-axis - Spent time on 1 iteration, Y-axis - Number of iteration 21
6.4 Measurements in a system under heavy load, expected runtime per iteration 2.2ms. X-axis - Spent time on 1 iteration, Y-axis - Number of iteration 22
6.5 Measurements in a system without any load, expected runtime per iteration 0.6ms. X-axis - Spent time on 1 iteration, Y-axis - Number of iteration 22
6.6 Measurements in a system under heavy load, expected runtime per iteration 0.6ms. X-axis - Spent time on 1 iteration, Y-axis - Number of iteration 23
6.7 Measurements on an Amazon EC2 Server without any load, expected runtime per iteration 0.6ms. X-axis - Spent time on 1 iteration, Y-axis - Number of iteration 24
6.8 Measurements on an Amazon EC2 Server under heavy load, expected runtime per iteration 0.6ms. X-axis - Spent time on 1 iteration, Y-axis - Number of iteration 24

CHAPTER 1

INTRODUCTION

Timekeeping has been around for a long time. With society and technology progressing on a constant rate so was the need for a more accurate timekeeping. Nowadays the needs for timekeeping vary greatly, from simple stopwatches for the purpose of measuring how fast a man can run an athletic course to different scientific tasks and researches like nuclear physics where the results often have to be in a microseconds or nanoseconds. Timekeeping devices in modern computers thus serves wide spectrum of important practical needs, such as keeping real-time as accurate as possible, accounting for software execution time as detailed as possible, and measuring how much CPU time a client has used in cloud as fairly as possible.

A convenient and popular CPU-based mechanism that allows software to measure time is a *Time Stamp Counter* (TSC). TSC has been around for a while now and used frequently by both system software and user-level applications. In PC, TSC is a 64-bit hardware counter which increments every clock cycle. TSC is then easily accessed by a program directly executing an instruction called `RDTSC` (Read Timestamp Counter). The fact that `RDTSC` is an instruction and TSC rate is pegged to CPU clock frequency makes using TSC for various timekeeping purposes very attractive.

However, even though TSC mechanism allows very accessible low-level interface, certain conditions are needed when TSC is to be used to solve practical timekeeping problems. In cases of measuring *wall-clock time* — the real time elapsed between two events —, TSC could

be easily utilized by taking two TSC readings at each event and dividing the difference with CPU clock frequency. Of course, this works as long as the rate at which TSC increments remains constant, and that TSC value hasn't been "reset" between the events, both of which may or may not be true in actual systems. In cases of measuring *user clock cycles* — the clock cycles needed to execute a stretch of user code —, two TSCs could be read on before and on after execution of the subject code. The difference of the readings then could be actual CPU cycles spent by the code, as long as there is no interruption during the code execution, and the frequency of CPU remains constant throughout. But these conditions are difficult to meet in practice. In cases of measuring *billable time* — the amount of CPU time actually used by a client of a shared system —, multiple TSC readings could be taken whenever the client occupies the host machine and thus consumes CPU time. Billing system would then process the readings to produce the time spent by the client, but the challenge is to attribute fairly time spent by system-level events, such as demand paging. In conclusion, the easy interface of current TSC belies the complications when it is used to solve practical timekeeping problems.

Moreover, the situation gets even more complicated when we consider hardware virtualization such as VmWare, Hyper-V, Xen, etc. Does an `RDTSC` instruction executed inside a guest virtual machine (VM) report a value that is real or virtualized? Does the TSC progress even if the guest VM gets scheduled out? Should the cost of hypervisor-induced VM in-guest paging (i.e., balooning) [14] be billed to client? If not, how do we distinguish operator-induced paging events [15] from regular guest paging event so that we can account for the measure time differently? Does the rate of TSC of a guest VM change when the guest VM migrates to different physical machine running at different clock frequency? Does the TSC value take on different value when the VM migrates? These are a few additional TSC-related questions

that needs to be addressed if we are to take virtualization into consideration.

But, surprisingly, current TSC interface does not provide enough flexibility to solve each of those problems. In addition, modern solutions solve a wide range of specific and unique problems and do not provide the flexibility that is needed.

In this paper, we first examine the current issues caused by the gap between the simplistic low-level TSC interface and the various timekeeping scenarios, and then propose a hardware extension which can address most of the issues. We performed various measurements using TSC on different computing platforms to confirm the unreliability and potential measurement errors of using TSC. In addition, we discovered that many current production systems rely on temporary fixes and inconsistent patchworks in order to address some of the issues.

To address these issues in a more fundamental way, we propose an improved version of TSC hardware interface, which we call *Caviar* (Counter Augmented by Virtually Interposable Adjustable Register). With Caviar, software can configure the behavior of TSC in a way that is deemed to be correct by the software in that particular context. Software then can execute `RDTSC`on the Caviar-enabled TSC to get correct TSC values all the time without having to worry about possible background events such as faults, VM exits, or CPU frequency scaling. Caviar is also designed with virtualization in mind – the interface makes it easy to virtualize the Caviar behavior via recursive instantiation of Caviar register. The simple minimalist interface of Caviar would be able to allow efficient, low-cost hardware implementation as well.

We subjected Caviar to a few usage scenarios of TSC-based time measurement under various system configurations. In the cases where we need to measure wall-clock time, software can reliably use Caviar-Master TSC to obtain TSC value that always runs at a constant fixed rate. In the cases where we need to measure user clock-cycles spent by a stretch of a user-level

code, one can use Caviar-Slave TSC, which is configured to suspend itself upon kernel entries and to adjust rate during frequency scaling events. In the cases where we need to measure billable time spent by a client, system can use Caviar-Slave TSC that is configured to include not only the user-time of client, but also a part of system-time incurred, depending on the agreed upon billing policy.

Contributions of this paper are as follows:

- Analysis of current TSC-based timekeeping mechanisms to assess practical reliability
- Proposal of new hardware interface that enables correct and reliable software implementation
- Wide-ranging review of time measurement issues on modern computer systems.

In many critical systems issues, a deficiency in hardware cannot be made up by software alone. For example, the dual modes of operations by processor are needed to implement any form of kernel protection and process isolation thereafter. The atomic instructions, such as compare-and-swap, are essential to correct and efficient implementation of any form of thread synchronization in shared-memory multi-core systems. In the same vein, we would like to argue that TSC-based timekeeping mechanism needs to be provided by hardware in a more systematic and reliable way. Our proposed hardware solution – Caviar – would be able to solve most current TSC issues in a fundamental and elegant way.

The rest of the paper is organized as follows. In Section 2, we highlight the problems of using TSC to measure time with an example. Section 3 analyzes why seemingly simple and innocuous TSC can fail to provide reliable outcome by analyzing different intentions that programs may have when measuring time using TSC. Section 4 proposes our hardware solution

– Caviar – that could address deficiencies in existing TSC. In Section 5, we revisit our previous examples using Caviar this time in order to demonstrate that Caviar can eliminate vagueness and unreliability. Section 6 provides experimental evidence of the deficiencies of current TSC. In Section 7, we provide extensive review on timekeeping hardware, TSC, system time accounting on both traditional and virtualized system platforms. Section 8 provides related works, and we conclude in Section 9.

CHAPTER 2

BACKGROUND

In this section, we first review time measurement mechanism in modern computer system and then examine measurement issues when a program uses such mechanism to measure time intervals or CPU time consumption.

Hardware timers and virtualization support are CPU features you can find in modern computing systems such as x86 [3]. System software — operating system (OS) and virtual machine monitor (VMM) — utilizes these features to efficiently share hardware resources and accurately measure usage of such shared resources. Operating systems share limited amount of hardware resources to multiple processes, each of which may belong to different end-user accounts. Usage statistics, such as how much CPU time a process has consumed, need to be accurately collected. Machine virtualization takes this resource sharing a step further by allowing multiple operating systems on a single hardware. Virtualization is popular in cloud computing because a hosting company can flexibly assign multiple virtual machines to smaller number of physical machines, thereby reducing costs, even if the hosting company sells computation assets on a per-machine basis to which customers can install *any* operating system. Therefore, it is even more important in a virtualized environment to collect accurate hardware usage statistics in order not only to meet service level agreement, but also to ensure billing correctness.

The most popular hardware timer is based on *Time Stamp Counter (TSC)*, which is a

CPU internal register that counts the number of clock cycles the CPU has made since the last reset. In x86, TSC is a 64-bit register whose content can be read by executing an instruction called `RDTSC` (Read Time Stamp Counter). A program can simply execute `RDTSC` instructions at certain events of interest to capture TSC values when these events occurred. By comparing captured timestamps, the program may deduce passage of real time or how much CPU clock cycles has been spent between events. Although there has been other hardware timekeeping mechanisms that predate TSC, due to TSC's extremely high resolution and accessibility, many OSes and systems libraries have moved to use TSC as there source for timekeeping.

The description on TSC presented above is accurate, but the actual reality of timekeeping hardware is very complicated with a long, convoluted history. We defer detailed exposition of timekeeping hardware to the later part of this paper at Section 7.

The very fact that user-level program can obtain timestamps at nanosecond resolution without calling into a system library or underlying operating system is indeed a huge advantage, and one may measure the "time" spent by a piece of code by simply executing two `RDTSC`s, as shown in Figure 2.1.

In the figure, we are explicitly using `RDTSC` instruction to probe TSC directly instead of invoking system calls to kernel, further improving accuracy and resolution of the measurement itself. The difference of timestamp values would then simply be the clock cycles spent by this processor during the execution of the code.

There is nothing special in the code above and the sequence of measurements is simple, but let us take a closer look at what the measured clock cycle number (`clk_elapsed`) can actually represents:

1. *Is it the wall-clock time?* That is, if we divide `clk_elapsed` by CPU's nominal clock

```
1  inline uint64_t rdtsc() {
2      unsigned int lo, hi;
3      __asm__ __volatile__("rdtscp" : "=a" (lo), "=d" (hi));
4      return ((uint64_t)hi << 32) | lo;
5  }
6
7  void clk_measure()
8  {
9      uint64_t tsc_before, tsc_after, clk_elapsed;
10
11     tsc_before = rdtsc();
12     do_some_computation();
13     tsc_after = rdtsc();
14
15     clk_elapsed = tsc_after - tsc_before;
16     printf("Computation␣duration:␣%lld␣\n", clk_elapsed);
17 }
```

Figure 2.1: Example of using TSC to measure clock cycles during the execution of a code.

rating (e.g., 4.0 GHz), are we then going to get the actual real-time elapsed?

2. *Is it the CPU cycles consumed by the stretch of the user code?* That is, can we use this method of clock measurement to reliably "benchmark" the performance of the user code?

3. *Is it a fair measure of CPU usage by which a customer should be billed?* That is, can one use this method to reliably account for CPU usage in multi-tenant systems including virtual machine based hosting service?

Careful examination quickly reveals that in order to be able to answer any of these questions, there are assumptions to be made on how exactly TSC hardware behaves and how the underlying system software treats TSC progressions upon entering kernel, as well as upon scheduling events.

Answer to question 1 can only be true if the hardware holds the TSC clock rate constant throughout regardless of CPU frequency scaling or power mode (i.e., C state) change, and the

system software should *never* pause, reset, or manipulate TSC [1] regardless of privilege mode or what process the CPU happens to execute at any given time. Hence, the above is required for hardware and system software if one wants to use TSC as a reliable source of wall-clock time or real time measurement. A stopwatch program that has to measure the passage of real-time, or a user time-out routine in an interactive application are examples that require such TSC behavior.

In contrast, however, answer to question 2 can only be true if the hardware *may vary* TSC clock rate when CPU undergoes frequency scaling or power state change, and system software *pauses* TSC advancement upon entry to kernel or when the program is being scheduled out. If these requirements are not met, any user measurement of CPU consumption of user code, as shown in Figure 2.1, has a non-zero chance of incorrect readings. When we allow CPU frequency to vary while TSC frequency remains constant, we may likely consider the measured cycles not precise. A worse cases of incorrect readings can happen when the user process gets interrupted or scheduled out in the middle of the execution. If such interruption happens and TSC keeps counting, then the measured clock cycles include not only the time spent by the kernel path but also the time spent by some other programs or I/O wait. Hence, should a program want to measure the amount of CPU consumption by a piece of code, then TSC must support aforementioned requirement, which is different from the requirement needed when we need to measure wall-clock time.

Lastly, answer to question 3 can only be true if TSC advances in kernel mode only when the kernel is entered in order to service the direct needs of the user process. In addition, if the

[1]Exception to this rule is when the CPU synchronizes TSC to external wall-clock time source.

Figure 2.2: Results of measurements in system with no load using the code provided in figure 2.1

hardware is a virtual machine, TSC should advance only when the virtual machine is scheduled on a physical machine, and the rate at which TSC advances should reflect the relative performance differences when the virtual machine migrates to different physical machine with different performance.

Therefore, correct interpretation of TSC measurement could only be done with understanding of systems configuration under which the measurements are taken. When a system executes a program, most operating systems allocates resources

Based on the code provided in figure 2.1 we conducted an experiment in order see how severely those issues affect the overall accuracy of readings.

As we see on the figure 2.2 in a system with no load at all, the results of a measurement are not fluctuating as much, this is because the system is not performing any other unrelated tasks and the probability of scheduling events or system calls occurring is very small.

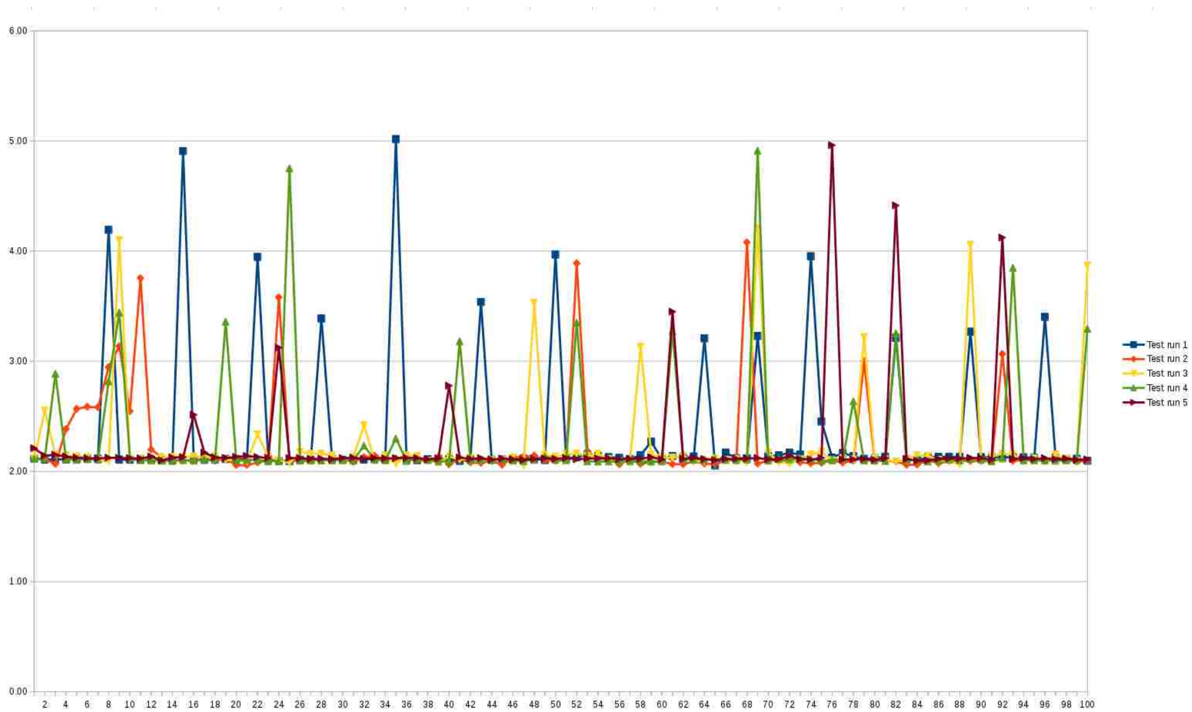

Figure 2.3: Measurements in a system under heavy load using the code provided in figure 2.1

The test on figure 2.3 on the other hand was conducted in a system that is under heavy load and as we see from the results, the fluctuations of reading are all over the place. This kind of scenarios can easily occur in a virtual environment, as one physical machine can be used for hundreds different users each utilizing a chunk of the systems computing power. Thus making the system execute and schedule hundreds of different processes that can impact the measurements of a single user.

Virtual Machine Monitor (VMM) is a software program that allows creation, management and governance of Virtual Machines (VM) and manages the operation of a virtualized environment on top of a physical host machine. VMM is the primary software behind virtualization environments, VMM controls the creation of VMs, each with their own operating system, resource allocation, and other input/output (I/O) resources.

As we have shown, despite its simplicity, TSC-based time measurement can lead to in-

accurate readings depending on the end purpose of the measurement. We argue that the "inaccuracy" actually comes from the gap between what TSC can provide and how TSC readings are used in actual end-use cases. In fact, we also believe that current TSC hardware interface provides inadequate support to meet various end purposes when clock-counting is required. To better understand the gap, we discuss in Chapter 3 how TSC should act depending on the various clock-counting use cases. Then in Chapter 4, we provide our enhanced TSC mechanism, called Caviar, that addresses these needs.

CHAPTER 3

TSC USAGE FOR DIFFERENT NEEDS

Continuing the discussion risen in the previous section, each of the asked questions come with a specific set of needs, and each of them require a unique approach.

Wall-clock measurements

One of the needs is to measure wall-clock time in seconds, milliseconds, microseconds, nanoseconds etc. For that purpose the expected TSC measurements should be accurate enough as long as a constant rate TSC is used and the exact frequency of the processor is known. Even if a more accurate result is needed on a lower resolutions like microseconds or nanoseconds.

CPU performance measurements

Another need that may arise is measuring the actual clock cycles spent by user code for the purpose of profiling the code performance or behaviour. For that purpose the expected behaviour of TSC should depend on what specific needs does the user have towards performance measuring. In some cases the measurement might include any interrupts, scheduling events that had occurred during the measurement if it has to be a part of the measurement, but in most cases, system executions that run outside the user code are bottlenecks that need to be excluded from the final results, so the TSC should be able to stop the counting process while those processes are executing and then resume counting to deliver accurate results on

user code execution.

CPU usage measurements for billing purposes

In some cases it is necessary to measure the CPU time spent by the user for billing purposes, so the user will pay exactly the amount that the CPU spent performing user related tasks. So everything that happens outside user related tasks should be excluded from the overall measurement process. Achieving that could be complicated especially in virtual environments. While performing some user related tasks the VM can unexpectedly migrate, causing migration issues for timekeeping, in that case the system needs to perform an Automatic Scaling process if the new hardware that the VM had migrated to runs faster or slower than the original one for the timer to show accurate results or the migration process can be halted while the user is performing some tasks. As with the previous need any unrelated system call or scheduling event that has occurred outside user related tasks should be excluded from the final result, but if any of those processes are invoked by the user, then they should be included in the bill as they perform user related tasks. As an example VMMs often allocate memory using the "ballooning" technique. Virtual memory ballooning is a computer memory reclamation technique used by a hypervisor to allow the physical host system to retrieve unused memory from certain guest VMs and share it with others. In that case if the virtual system has generated a page fault as a consequence of ballooning it should not be included in the final bill, but if there was a system interrupt generated by the user to request additional memory for some tasks that the user performs, then it should be billed.

CHAPTER 4

HARDWARE PROPOSED SOLUTION

For different use cases described in the previous section there should be a unique solution to each of the issues that occur during the usage of the TSC.

The way that the "CAVIAR" system works is that it has multiple secondary TSCs each linked to a TSC Control module that is responsible for deciding when should the secondary TSC stop or start counting. It does that by receiving commands from the system that flip the bits inside the control module that represent different scenarios of TSC usage as well as different issues that can impact the measurement. After receiving the command based on the use case it enables or disables the counting process of the secondary TSC.

There are not many issues linked with wall-clock time measurements, unless there is a need to be precise with when working with microseconds or nanoseconds. So in order to be accurate and consistent there needs to be a Primary TSC that behaves like a traditional constant rate TSC. That way we will get accurate and consistent results measuring or timing wall-clock time. In Virtual Environments, when performing the same tasks and faced with migration problems the resulting values should be exactly scaled based on exact frequency of the processor.

Based on the figure represented above 4.1 the solution to further needs can be provided.

For the purpose of measuring the clock cycles spent by user code for performance monitoring or benchmarking the proposed solution is able to stop the TSC from counting in certain

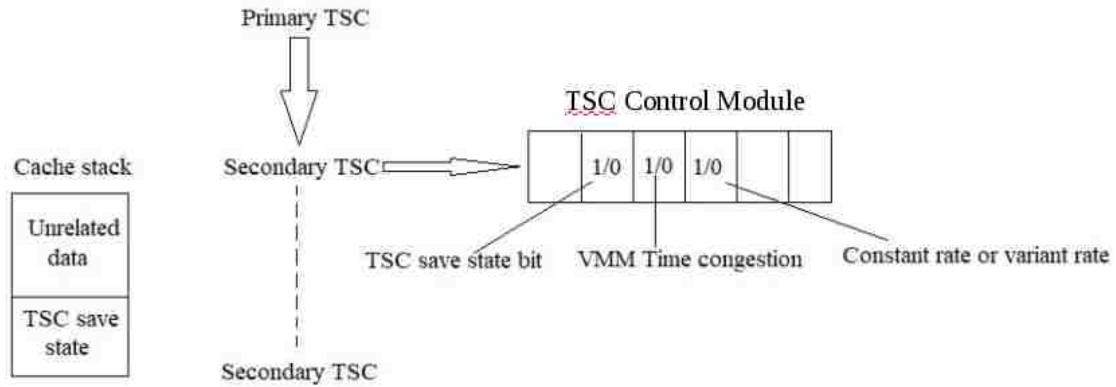

Figure 4.1: Sketch of Hardware proposed Solution

situations. If any interrupts arise, or any scheduling event is present that can impact the resulting measurement, one of the bits in the TSC Control Module is flipped in order to save the current state of TSC in Cache/Memory. After everything is clear and user code is able to resume its execution, the bit is flipped back in order to continue counting. Not only that, but the flexibility of the TSC Control Module may insure accurate and consistent results without any fluctuations. That way any measurement is ensured to reflect the exact time spent on code execution.

The same may be applied for billing purposes but with minor modifications in order to meet certain requirements. When any interrupts or scheduling takes place and the one causing it is the system itself, then the timer has flipped bits that stops counting for the time being. But if any interrupts or page faults are caused by the user as a result of code execution or something else, then the timer should continue counting for this to appear in the final bill.

CHAPTER 5

USAGE OVER HARDWARE PROPOSED SOLUTION

Usage cases of the proposed hardware solution are many.

```
1  int main()
2  {
3      clock t1, t2;
4      t1 = tsc_wc.now();
5
6      Sample_project_execution();
7
8      t2 = tsc_wc.now();
9      Wall_clock_time_elapsed = t2 - t1;
10     cout << Wall_clock_time_elapsed;
11 }
```

Figure 5.1: Caviar usage for wall-clock time

In listing 5.1 some user function "Sample project execution" is being timed by using the improved wall-clock time functions. By timing the execution using the provided method, the results are consistent and accurate for the designated task.

```
1  int main()
2  {
3      pin_this_vm();
4      clock t1, t2;
5      t1 = tsc_cycle.now();
6
7      Sample_project_execution();
8
9      t2 = tsc_cycle.now();
10     Cycle_spent_by_user_code = t2 - t1;
11     cout << Cycle_spent_by_user_code;
12 }
```

Figure 5.2: Caviar usage for CPU cycle counting

In listing 5.2 the same function "Sample project execution" is being executed by the user, and this time it is necessary to know the amount of clock cycles that have passed while executing relevant code only. first of all it is necessary to pin the Guest Virtual Machine to avoid unnecessary migrations that could cause problems by forcing the system to scale the results of counting on one frequency, to match the same results on another frequency. The `tsc_cycle.now()` function starts and ends the timing process, it also creates a background process that monitors any interruptions that could potentially harm the final results of the measurement.

```
1  int main()
2  {
3      pin_this_vm();
4      clock t1, t2;
5      t1 = tsc_billing.now();
6
7      Sample_project_execution();
8
9      t2 = tsc_billing.now();
10     Billed_time = t2 - t1;
11     cout << Billed_time;
12 }
```

Figure 5.3: Caviar usage billing

In listing 5.3 the process of measurement is the same as in the previous one, but the `tsc_billing.now()` creates a background process that acts as a controller. The purpose of this function is to stop counting whenever there is an interruption from the system side and resume or continue the counting process when interruption is caused by some user related actions, whether it is code related executions, or system call related interrupt that is related to user code.

CHAPTER 6

EXPERIMENTAL RESULT

In this section we will see how does the Time Stamp Counter performs in different conditions.

The conducted experiments were based on the code that was used previously in Section 2 illustrated in figure 2.1. In order to achieve the most accurate results, no libraries were used, to exclude the possibility of receiving unnecessary time spent on certain library executions. Therefore The ”rdtsc” instruction used in the code, was written as an inline assembly code. The user function was written in such way, that it could be modified to execute the same amount of computations over a certain period of time, that period is configurable by the user.

Based on that, the first experiment that was conducted included a simple execution of the code with an expected measurement of each iteration of 23ms in an environment without any load. That means that the system was not performing any heavy computational tasks, and the processors had been in an idle state. As we see from the figure 6.1 the results seem mostly stable in such an environment. The initial high results of the first iteration indicate the ”cold start” of a program, meaning that at that start of a process, memory has to allocated for that execution as well as the CPU itself has to get out of the idle state in order to start performing this task.

In the next figure 6.2 everything staid the same, but now the system is under heavy load. This can occur while a machine is running several VMs at the same time and each of them are

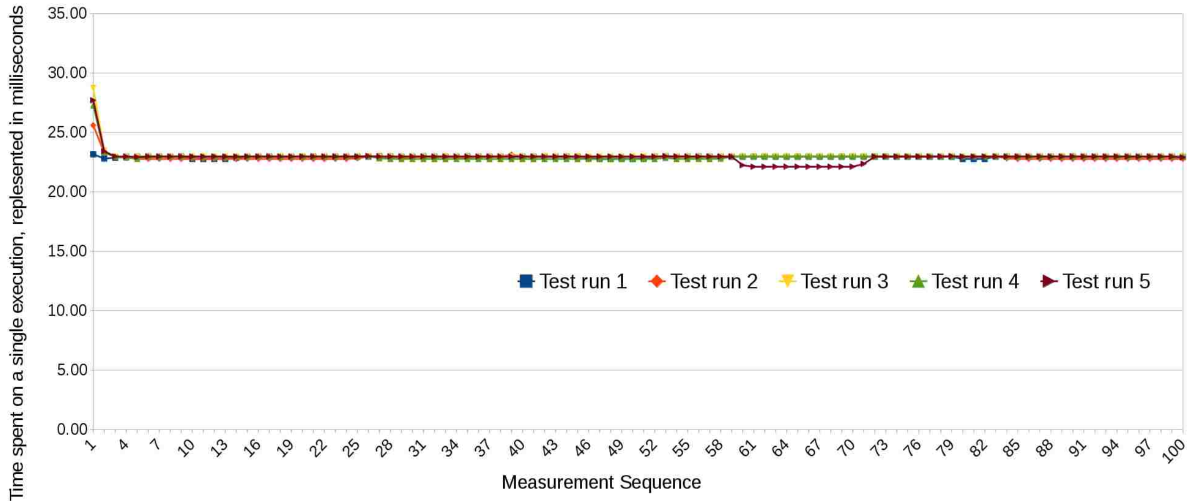

Figure 6.1: Measurements in a system without any load, expected runtime per iteration 23ms. X-axis - Spent time on 1 iteration, Y-axis - Number of iterations

performing some computations, using up processing power of the CPU. Or it can also occur in a non virtualized system, when the system is loaded with the complex computations of the code itself. As we see from the results, the difference between the two are apparent. Each spike represents a scheduling event or a paging fault. in order to clearly see and distinguish between them, we need to perform the same experiment but faster.

The second experiment was conducted using the same code but modified to run faster in order to clearly see and distinguish between different spikes and what caused them. The expected results from each iteration were 2.2ms. In figure 6.3 we can clearly distinguish the "cold start" after which the results smooth out and continue to be consistent throughout the test. However it does not exclude a possibility of a scheduling event occurring if this code were to run for a longer period of time.

In the following figure 6.4 the experiment was run with the same 2.2ms expectation but this time the system was under heavy load. As we see the resulting spikes that are caused by

Figure 6.2: Measurements in a system under heavy load, expected runtime per iteration 23ms. X-axis - Spent time on 1 iteration. Y-axis - Number of iteration

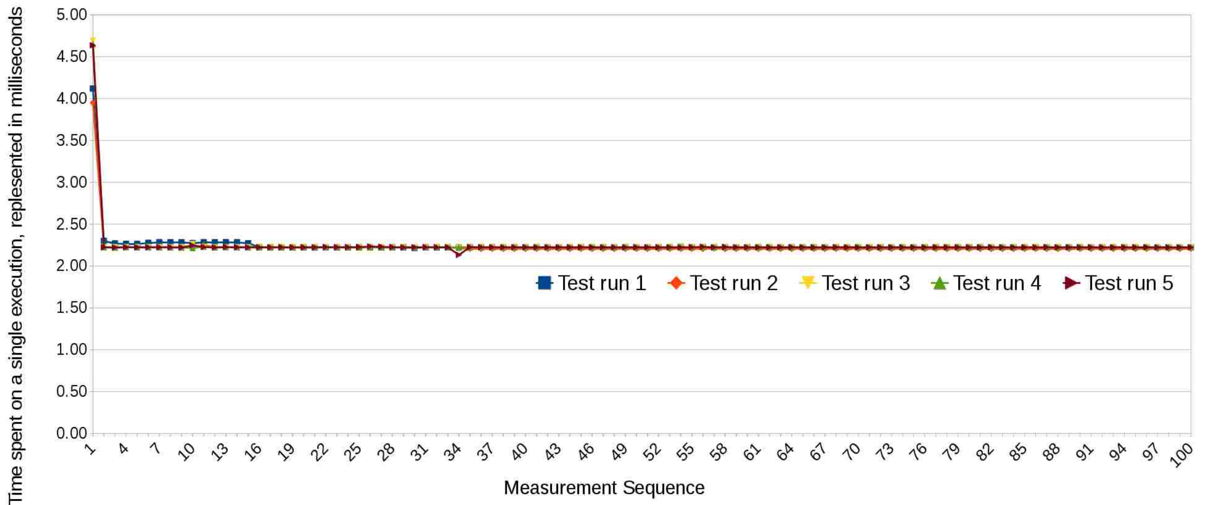

Figure 6.3: Measurements in a system without any load, expected runtime per iteration 2.2ms. X-axis - Spent time on 1 iteration, Y-axis - Number of iteration

scheduling events and paging faults, are seen more clearly, but it is still hard to distinguish which spike is caused by what process. So the next step and the next experiment should be executed even faster so that each iteration would measure less than 1ms. That way we will see if an iteration took more than 1ms to execute, then we have a scheduling event, that usually takes 1-2ms.

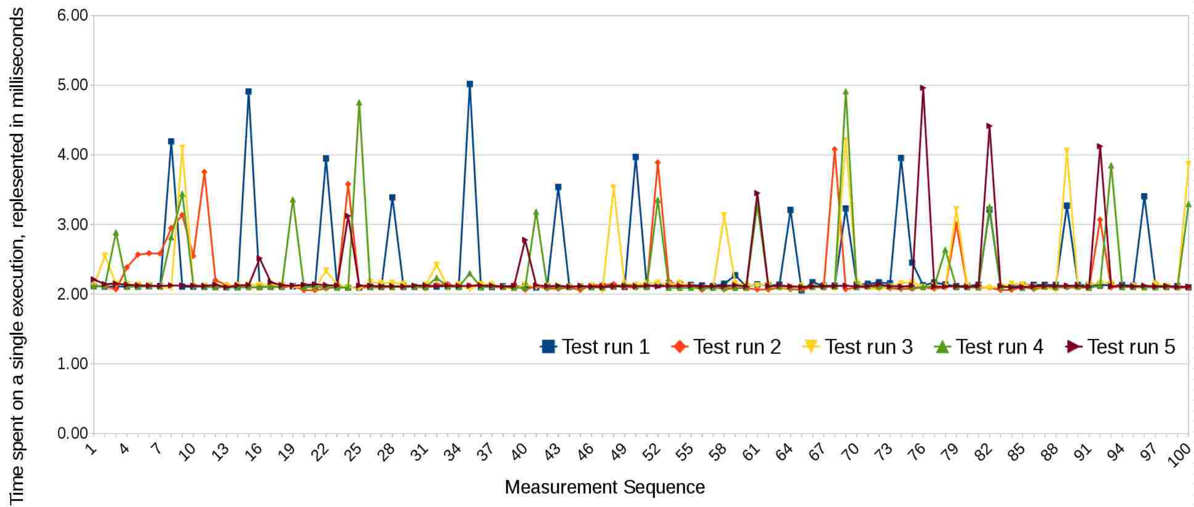

Figure 6.4: Measurements in a system under heavy load, expected runtime per iteration 2.2ms. X-axis - Spent time on 1 iteration, Y-axis - Number of iteration

Following the said above the third experiment was conducted with the same code executing even faster, with an expected run time of each iteration of 0.6ms. As seen in figure 6.5 apart from the "cold start" we can now see that with some executions the frequency of the CPU was throttling for a short period of time that was smoothed out as it went on.

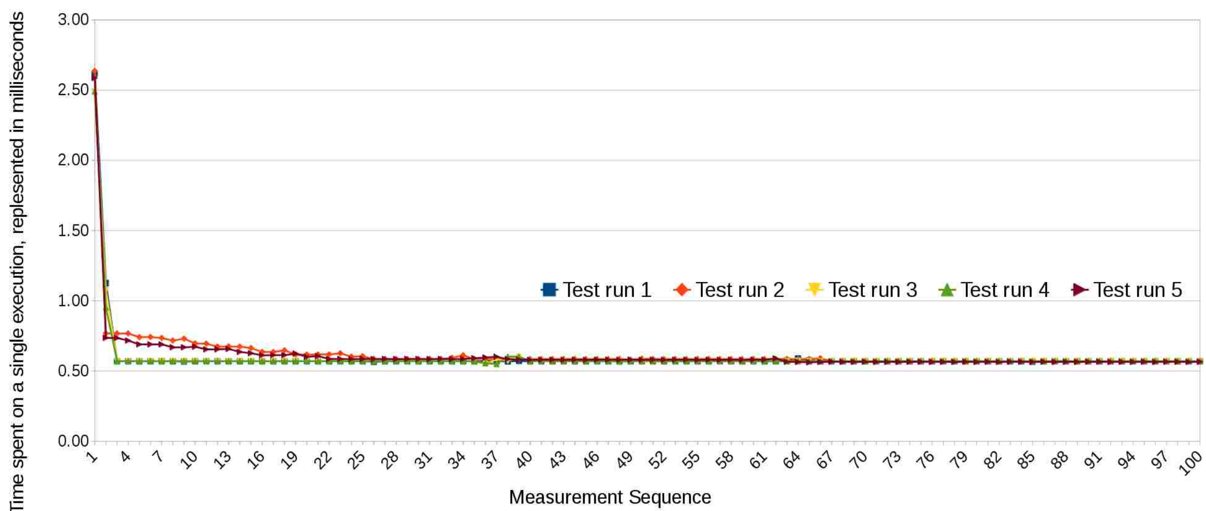

Figure 6.5: Measurements in a system without any load, expected runtime per iteration 0.6ms. X-axis - Spent time on 1 iteration, Y-axis - Number of iteration

In figure 6.6 we can now clearly distinguish each spike that occurred during this experiment and link each of them to different issues that caused them.

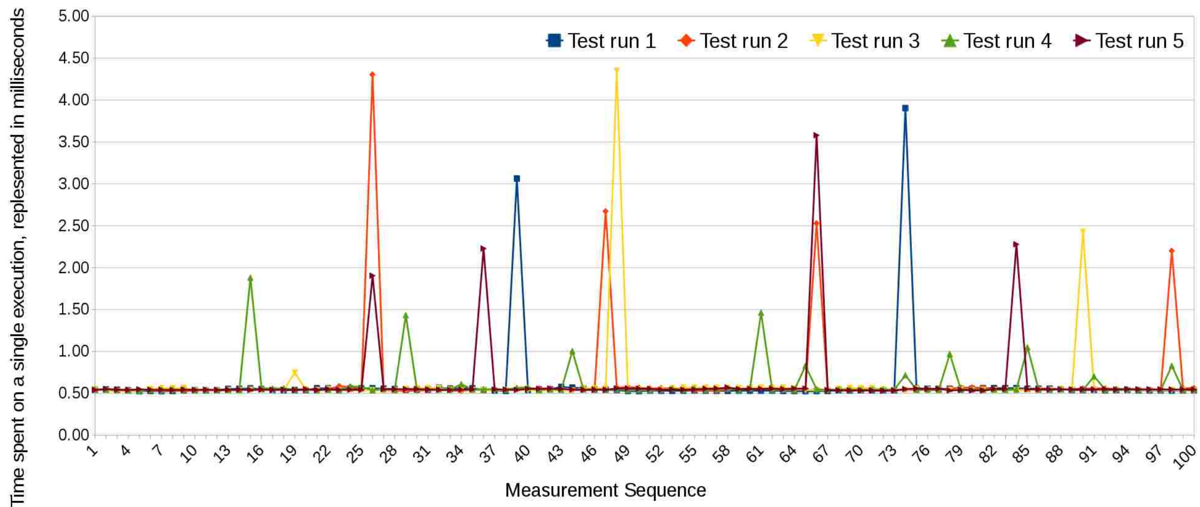

Figure 6.6: Measurements in a system under heavy load, expected runtime per iteration 0.6ms. X-axis - Spent time on 1 iteration, Y-axis - Number of iteration

After these experiments it has become apparent that the results of TSC are very inconsistent and not accurate especially if a system is loaded with different complex executions or processes. But those executions were only in a bare metal system where we have control over the environment in which we measure time. What will happen if we perform the same measurement but in a virtualized environment?

In figure 6.7 are the results of a measurement on an Amazon EC2 Server. The code that was executed is the same as the one executed in the bare metal system and the frequency of the processor in the Amazon machine is also the same as the one in the bare metal experiment. As we can clearly see, when measuring something in virtual environment the stability that we seen in previous experiment is no more. These are the results of system without any load.

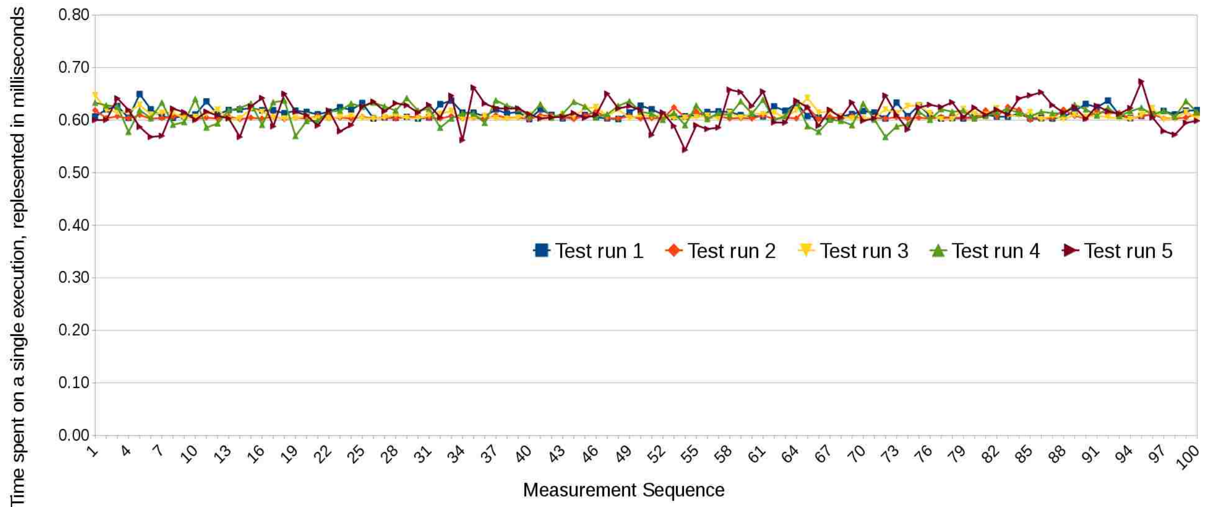

Figure 6.7: Measurements on an Amazon EC2 Server without any load, expected runtime per iteration 0.6ms. X-axis - Spent time on 1 iteration, Y-axis - Number of iteration

The results presented in figure 6.8 are from a virtual environment that is under heavy load. As we can clearly see, the results are all over the place and there is no was to extract any accurate data from such measurements.

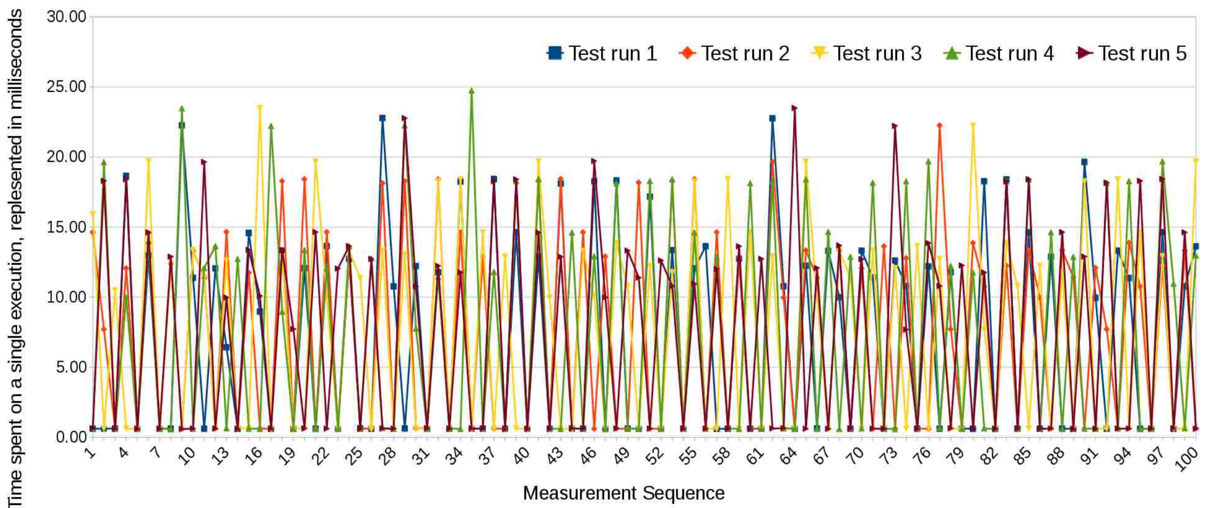

Figure 6.8: Measurements on an Amazon EC2 Server under heavy load, expected runtime per iteration 0.6ms. X-axis - Spent time on 1 iteration, Y-axis - Number of iteration

CHAPTER 7

SURVEY OF TIME MEASUREMENTS AND TSC

This Chapter is dedicated to the background overview of Hardware, Timers and Virtual Machines.

Hardware Timers

There are several timekeeping hardware sources in a typical x86/PC system, but user applications rarely use them directly. Instead, they typically use system API that is part of operating system system call or run time library routine that comes with programming language. This is because most timekeeping hardware is treated as I/O devices, access to which is privileged. Operating system therefore manages the hardware and then provides user a set of abstract timer/alarm operations via system call interface. However, one glaring exception to this privileged nature of timekeeping hardware is CPU timestamp counter (TSC), which, in many cases, is allowed to be accessed directly by user application via execution of a single machine instruction. The Operating System (OS) itself also needs to know the time and have the ability to measure it for scheduling the work of user threads, accounting for the resources consumed by them, performance profiling, power management, etc. At the same time the OS works directly with the interfaces that are presented by devices. As there are a lot of different timers modern OS can select, one "central" device is used in the beginning of the loading process. Time Stamp Counter (TSC) is based on the idea of using the processors

clock as the source of time. The TSC register in processor core stores the current clock number counted from system boot. To read this register, a program can just execute an instruction (i.e., `RDTSC`). Depending on system configuration, this instruction can be used directly by user-level programs without raising a fault. As TSC can provide both high resolution and convenience, programs may use TSC at runtime to quickly measure the time spent during the execution of a small segment of code. Despite the utility of TSC, the development of PC's TSC hasn't been smooth. Originally a TSC was created with each logical processor and it was a good way to get CPU timing information. But with the creation of multi-core processors and the technology that it brought with it, for example the simple hibernation, the TSC cannot be relied upon to provide accurate results. If, for example, a program needs to measure time with TSC in a multi-core system, the TSC counters need not only to be synchronized but also to equalize tick rates of all the cores. In virtual environments, TSC can be virtualized. The virtual TSC falls behind when there is a backlog of timer interrupts and catches up as it clears. The virtual TSC does not count the virtual CPU cycles it continues even when the guest operating system is turned off, so the guest TSC values do not match the host TSC values. One reason why that is happening is that when the guest operating system first been installed on a host system, the TSC values are identical, but if that guest would have been moved to a different system the original TSC values stay and if the new host system has a different CPU speed from the previous one there bound to be mismatches in TSC values along the way.

Different Kinds of TSC

There are several types of TSC, each of which can roughly be distinguished by different instruction defined.

Read Time Stamp Counter (`RDTSC`) is the original PC TSC instruction, first appeared in Intel Pentium in 2003. Upon execution, the current 64-bit counter value – the clock cycles counted from the last time of a power disconnection or a reboot – is stored in a pair of registers (EDX:EAX). Unlike other timekeeping hardware devices whose access is privileged, `RDTSC` instruction can, by default, be executed on any level of privileges. The OS of course can disable the `RDTSC` in user mode. In such case, an exception is raised if user code executes `RDTSC`. The same TSC counter is exposed as Model-Specific-Register (MSR) interface, which allows privileged code to read/write TSC value. To update the TSC counter, an operating system can therefore execute `WRMSR` (write to MSR) instruction, specifying new TSC value.

Being an instruction, `RDTSC` is originally subjected to instruction reordering found in most modern out-or-order micro architecture. Hence, care must be taken if one wants to obtain TSC with instruction level precision. One way to enforce program order is to execute memory barrier (serializing instruction) along with the `RDTSCP`. Reading the TSC is a single instruction and on real hardware it's very fast, but in a virtual machine, that instruction faces a lot of overhead, and thus runs slower resulting in false readings.

Read Time Stamp Counter and Processor ID (`RDTSCP`) is an upgrade of `RDTSC` and is now supported in all recent x86 systems. Out of order execution is when a processor can execute system instructions in an order that is different from the order that is written in the program code. This means that the `RDTSC` execution can be halted or on the contrary executed faster than expected. So it means that `RDTSC` cannot reliably measure the part of the executable code

as a result there are no guarantees of a monotonic readings. in the later computer architecture `RDTSCP` was created, it is an instruction that partially serializes the flow of execution, so it does not need additional barriers. It is not affected by power options of the processor. And the multi-core problem that persisted in the `RDTSC` instruction, where it could not synchronize the clock readings between different cores can also be handled by the `RDTSCP` instruction. In order to deal with those problems `RDTSCP` has a single signal source and it can detect the migration of the process between multiple cores.

Timers in Virtual Environment

Like the real system, the guest system can periodically access all of the devices that provide the time: RTC, PIT, APIC timer, ACPI PM-timer, HPET, TSC. The first five of this list are external to the CPU of the device, so the approaches to their virtualization are similar. Work with peripheral devices is through the programming of their registers. The registers are either available through the port space (PIO, programmable input / output), then IN/OUT instructions are used, or at specified physical memory addresses (MMIO, memory-mapped input / output), and then they are operated with ordinary MOV instructions. In both cases, VT-x technology allows you to configure what will happen when you try to access devices from within the VM - monitor output or ordinary access. In the first case, the tasks of the monitor include emulation of interaction with the software model of the corresponding device, exactly as it would be in purely software solutions that do not use hardware acceleration. In this case, the processing time of each access can be greater than when accessing a real device. However, almost always the frequency of calls to the registers of timers is small, so the overhead of virtualization is reasonably small. In practice, there might be some exceptions

- some operating systems, found in the HPET system, begin to read it often and persistent. This causes a noticeable slowdown in the simulation. For example if a programmer wanted to allow direct access to the timers, it would be rarely possible. Firstly, a real timer device can already be used by the monitor for its own needs, and it is inadmissible to allow a guest to interfere with the operation of the monitor. Secondly, one real timer can not be divided into parts, and after all in one system there can be several VMs, and each one needs a copy of the device.

TSC Virtualization

Unlike other timer devices, this counter is located directly on the processor, and access to it is through the instructions `RDTSC`, `RDTSCP` and `RDMSR`.

As with other timers, there are two approaches to TSC virtualization:

- Interception of all calls followed by pure software emulation.
- Permission to read the value directly based on the TSC in the guest.

In the Intel VT-x architecture, the `RDTSC` exiting bit of the VMCS control structure corresponds to the behavior of the `RDTSC` within the guest mode, and the `RDTSCP` behavior is another bit, "RDTSCP enable". That is, both instructions (as well as RDMSR, interpreted as a variant of `RDTSC`), can be intercepted and software emulated. This method is rather slow: just reading TSC takes a dozen cycles, whereas a full cycle of going out of the monitor, emulation and return is thousands of cycles. For a number of scenarios that do not use `RDTSC` often, the effect of slowing down from emulation is invisible. However, other scenarios can try to "learn the time" with `RDTSC` every few hundred instructions which, of course, leads to a slowdown. In the case when the direct execution of the corresponding instructions for reading the TSC is allowed, the monitor can set the offset of the return value relative to the real one using the "TSC offset" field of the VMCS and thereby compensate for the time during which each guest was frozen. In this case, the guest will get the value of TSC plus TSC OFFSET.

Unfortunately, allowing direct execution of `RDTSC` has a lot of complications. One of them is the the complexity of fixing the exact moment when the monitor finishes and the guest starts – after all, the TSC "ticks" constantly, and the processes of transitions between the processor modes are instantaneous and have an unknown variable duration. One way or the

other, this difficulty introduces TSC uncertainty or errors that depend on particular VM implementation. A certain border zone arises, for which it is not clear which"worlds" to attribute to the measures carried out in it. As a result, it is very difficult to understand which TSC values the guest could see, and this creates an error of several thousand clock cycles per guest-monitor-guest switch over. Such a mistake can quickly accumulate and manifest in a very strange way. In fact, for normal implementation, there is not enough "atomically-instantaneous" exchange of the TSC values of the guest and the host.

The second problem is essential for VM monitors in cloud environments. Although with TSC-OFFSET we can set the initial value for TSC when entering guest mode, the rate of TSC change after that, can not be changed. This will create problems when the guest OS is hotly migrated from one host machine to another with a different TSC frequency. Since timer calibration is usually performed only on initial boot, after such a move, the guest OS will not correctly schedule events. As a result, we can say that the current state of hardware accelerated virtualization technology does not contain methods for efficient virtualization of the TSC counter. Or reality in one way or another "squeezes" into a virtual environment, or everything works very slowly. Of course, not all applications are so sensitive as to break down inside a guest with direct execution of `RDTSCP`. Especially if you write programs so that they take into account the possibility of running inside the simulator. And yet many virtualization solutions have moved to using TSC software emulation by default - although it is slower, but more reliable. The user must enable the direct execution mode himself if TSC creates performance problems, and if he is willing to investigate strange incidents associated with other times in his scenarios.

Current state of TSC-based measurement

The current approaches of solving the named above problems vary in rage from simple to very complicated and as practice shows the simpler ones do not provide the full scale solution to the desired problems. Some users experience difficulty in setting up a custom timer every time they need to measure time on a different task as each problem needs to be approached uniquely. So as a result the user sacrifices accuracy and consistency of measurements and presents an average result, validity of which he can not approve nor deny. That validity is solely dependant on the system it has been run on. More complicated solutions create even more bottlenecks for themselves as the more complicated the solution is the more system calls and functions it will have to call in order to execute itself. Thus creating an even bigger margin for inaccuracy to take place as it will potentially raise even more exceptions and invoke even more system calls that will impact the resulting measurement.

CHAPTER 8

RELATED WORKS

The evolution of Time Stamp Counter (TSC) has been dictated by changes in x86 instruction set architecture [3]. The very first implementation of x86 TSC was just a clock cycle counter; architecturally, TSC was supposed to count the cycles, so if processor changes its operating frequency, the rate of TSC also changes. Soon, there was increasing demand that TSC to be operating at constant frequency, which is necessary if TSC is to replace conventional wall-clock timer, so the semantics of x86 TSC changed in such a way that TSC always operates at the nominal frequency. At the same time, people begun to use TSC to profile code performance. The original RDTSC instruction was not serializing, meaning that CPU can reorder instructions around RDTSC. This meant RDTSC lacks precision when used for code profiling, so x86 added a serializing version of the instruction called RDTSCP. As x86 virtualization became mainstream in mid-2000, x86 architecture gradually added hardware support for virtualization such as VT-x. However, with virtualization, maintaining TSC semantics in virtualized environment became an issue [1]. A few fixes in x86 have then been added to address virtual machine migration, such as TSC offsetting and scaling, as part of VT-x specific features [3, 4]. Our proposed TSC solution is more generic and flexible, so if adopted, it will streamline x86 TSC measurement in both bare-metal systems and virtualized systems.

Hardware virtualization has a long history with its root on IBM VM370 in the late 60s.

However, x86 virtualization has become mainstream in early 2000 when VMware first successfully commercialized x86 hardware [11, 14]. The emergence of x86/PC based virtualization was also fueled by Xen hypervisor [2], which was developed open-source. Xen then became one of the important virtualization kernel in the industry, currently serving as the hypervisor of Amazon's EC2 cloud. Timekeeping in virtualization has been an issue, but it has so far been dealt with in-house solutions specific to individual virtualization [1, 10], partly due to the lack of consistent hardware TSC behaviour.

Descendants of UNIX operating system, such as Linux, keeps three different per-process CPU usage time: real-time, user-time, and sys-time. Real-time measures the wall-clock time passed since the process has launched. User-time measures the CPU time when the process has been actually executing in the user-mode. Sys-time measures the CPU time the process spent in the kernel-mode, excluding I/O wait. Operating system kernel measures these metrics using hardware time source, such as TSC. However, when a system is running in a virtualized environment, these measures can become unreliable. Also, these metrics do not accurately account for special types of CPU usage such as fast I/O operations done in synchronous fashion [16, 17] or involve lengthy in-kernel computations such as memory compression [15]. In many cases, users can only perform time measurement in-context in the user-level. Our proposed solution can be applied to address CPU accounting problem in these modern systems, producing more detailed usage statistics.

Time synchronization has been an important domain of research in computing as well as in networking. Invention of Lamport clock [6] is considered as one of the foundation of distributed computing. In the field of computer networking, numerous technical articles, specifications, and protocols have been produced. For example, Tian *et. al* [12] is a recent

example in which time synchronization issues in networking has been investigated. Although time synchronization is an important issue, it is not directly related to our work. If hardware TSC needs to synchronize with external clock, the system can use existing synchronization algorithms.

Early works on high-resolution hardware timer includes research of Ong textitet. al. [8]. In it, they presented a high resolution performance monitoring software on the Pentium.

CHAPTER 9

CONCLUSION

To support the Time Stamp Counter usage for different purposes like: wall-clock time, Benchmarking and billing, we have presented a solution named "Caviar" that has multiple secondary TSC that are controlled by the TSC Control Module, that allows the timer to start or stop the counting process as opposed to a traditional TSC that does not have the ability to do so. We have also confirmed through tests, that the fluctuations of measurements when using a traditional TSC are indeed impacting the accuracy and consistency of the results. As a result, this solution offers a more reliable and stable way to measure time in different and unique situations by eliminating the possibility of any system related activity to be counted towards the final result of a reading. For users with the need for accurate time measurements this solution will provide them with easy and flexible system to use, without the need to set up an external hardware and without the need of global satellite time synchronization that can be very costly.

In future work, our solution needs to be physically implemented in a live system to ensure the ability to properly work under said circumstances. After that this solutions needs to be implemented as a part of modern system architecture, for everyone to use.

BIBLIOGRAPHY

[1] Zachary Amsden. Timekeeping virtualization for x86-based architectures. `https://github.com/torvalds/linux/blob/master/Documentation/virtual/kvm/time-keeping.txt`. Accessed: 2019-3-8.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[3] Intel Corporation. Intel 64 and IA-32 architectures software developers manual. `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`. Accessed: 2019-4-1.

[4] Intel Corporation. Timestamp counter scaling for virtualization white paper. `https://www.intel.com/content/www/us/en/processors/timestamp-counter-scaling-virtualization-white-paper.html`, September 2015. Accessed: 2019-3-10.

[5] Elliott Hauser. Unix time, utc, and datetime: Jussivity, prolepsis, and incorrigibility in modern timekeeping. *Proceedings of the Association for Information Science and Technology*, 55(1):161–170, 2018.

[6] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[7] Microsoft. Acquiring high-resolution time stamps. `https://docs.microsoft.com/en-us/windows/desktop/SysInfo/acquiring-high-resolution-time-stamps`, May 2018. Accessed: 2019-3-1.

[8] Ong, Cheng Soon, Wong, Fadhli, Hasan, Mohd, Weng Kin, and Lai. A high resolution performance monitoring software on the pentium, 2000.

[9] Gabriele Paoloni. How to benchmark code execution times on Intel IA-32 and Intel 64 instruction set architectures. In *Intel Embedded*, September 2010.

[10] RedHat. RedHat enterprise linux KVM guest timing management. `https://access.redhat.com/documentation/en-us/redhatenterpriselinux/6/html/virtualization_host_configuration_and_guestinstallation_guide/chapvirtualization_host_configuration_and_guest_installation_guide-kvm_guest_timing_management`. Accessed: 2019-3-10.

[11] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference (ATC)*, pages 1–14. USENIX Association, 2001.

[12] Guo-Song Tian, Yu-Chu Tian, and Colin J. Fidge. High-precision relative clock synchronization using time stamp counters. In *13th IEEE International Conference on Engineering of Complex Computer Systems*, Belfast, UK, 4 April 2008.

[13] VMware. Timekeeping in vmware virtual machines, vmware information guide. `https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf`, November 2011. Accessed: 2019-3-10.

[14] Carl A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI '02, pages 181–194, Boston, MA, 2002. USENIX Association.

[15] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[16] Jisoo Yang, David Minturn, and Frank Hady. When poll is better than interrupt. In *Proceedings of the 2012 USENIX/ACM File and Storage Technology (FAST)*, Santa Clara, CA, February 2012.

[17] Jisoo Yang and Julian Seymour. Pmbench: A micro-benchmark for profiling paging performance on a system with low-latency SSDs. In *Proceedings of the 2017 International Conference on Information Technology: Next Generation (ITNG)*, Las Vegas, NV, February 2017.

Curriculum Vitae

Alexander Tabatadze, B.S.

Email : tabatadzealexander@gmail.com

Education:

Plekhanov - Russian University of Economics
B.S., Programming in computer systems, July 2015

Russian New University
B.S., Applied Mathematics and Computer Science, June 2016

University of Nevada, Las Vegas
M.A., Computer Science, May 2019