TECHNISCHE UNIVERSITÄT DRESDEN

TIME VIRTUALIZATION IN NOVA AND VANCOUVER

GROSSER BELEG

Author:
Meike ZEHLIKE

Matrikel: 3300332
meike.zehlike@mailbox.tu-dresden.de

Supervisors:
Dipl.-Inf. Julian STECKLINA
Dipl.-Inf. Benjamin ENGEL
M.Sc. Nils ASMUSSEN
Prof. Hermann HÄRTIG

February 22, 2013

ERKLÄRUNG

Hiermit erkläre ich an Eides statt, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 22. Februar 2013

Meike Zehlike

Aufgabenstellung

Zeitvirtualisierung

In einem virtualisierten System bietet der Virtual Machine Monitor (VMM) einer virtuellen Maschine (VM) virtuelle Geräte und virtuelle Resourcen, wie Speicher. Der Microhypervisor NOVA und der dazugehörige Userland-VMM virtualisieren jedoch Zeit nicht, d.h. eine VM sieht die Zeit des Hostsystems.

Wenn mehrere VMs sich einen physischen Prozessor teilen, wird z.B. eine Preemption bzw. das Ausführen einer anderen VM als Zeitsprung wahrgenommen. Bei Codepfaden, die darauf nicht vorbereitet sind, wie z.B. Timerkalibrierungsschleifen, kann das zu unerwünschten Ergebnissen führen. Von Zeitvirtualisierung spricht man, wenn jede VM trotz Unterbrechungen Zeit kontinuierlich und fortlaufend wahrnimmt, die virtuelle Zeit aber nicht beliebig weit von der physischen Zeit driftet.

Die Aufgabe des Studenten ist es in Linux Code zu identifizieren, der sich mit unvirtualisierter Zeit fehlverhalten kann und diesen stark vereinfacht als Testkernel nachzubilden. Anhand dieses Testkernels soll das Problem der fehlenden Zeitvirtualisierung auf NOVA/Vancouver veranschaulicht werden. Aufbauend auf diesem Wissen soll das Konzept einer virtuellen Uhr für NOVA/Vancouver entworfen werden, die unter beliebigen Lastsituationen des Hostsystems Zeit effektiv virtualisiert. Dieses Konzept soll prototypisch in Vancouver implementiert werden.

Contents

1 Abstract

Time in computing represents a resource with quite particular characteristics. It has a unique, irreversible direction and passes regularly, meaning that its state changes without interference. Today's systems usually consist of several time sources, which measure passage of time since system boot independently from each other. The attempt to virtualize any of those must not contradict the characteristics of time. Yet no sophisticated hardware or software solutions are available for virtualization environments. The issue here is that a virtual machine can be preempted at any point in time. This work presents an approach to virtualize time in a manner that does not invalidate what an operating system assumes. In spite of preemptions a virtual machine shall observe time passing with the same characteristics as it would when running on bare metal.

2 Introduction

Time in computing represents a resource with quite particular characteristics. As everywhere it refers to the experience of duration and is used to describe a sequence of events but in contrast to other resources, like memory, its state changes constantly by itself and almost at a steady rate. This means that time in a computer has a unique, irreversible direction and passes regularly. An operating system always assumes these characteristics for its calculations. Several devices that measure passage of time in various ways are available in today's systems.

In our virtualization architecture the NOVA Micro Hypervisor and Vancouver, the appropriate virtual machine monitor (VMM), provide virtual devices and resources to a guest operating system, i.e. a system that runs within a virtual machine (VM). However this does not include time yet, so that this guest has to use the time of the host, i.e. the machine it is running on.

The problem arises from the fact that the VM can be preempted at any time without a possibility for the guest to control or even notice that. The preemption and resume will be recognized as a jump in time because the guest's time is lacking the span for which it was suspended. In a multiprocessor system (SMP) time might even go backwards after a migration, if the time sources of the cores are not perfectly in sync, which is the common case. Most per CPU time sources start from the point when their CPU was powered on and this is very likely not to happen at exactly the same moment. These effects clearly contradict the assumptions of the virtualized operating system: As it is not aware of its virtualization it still takes time moving irreversibly and regularly in one direction for granted and it might lead into undesirable behavior, if that is not true anymore. If, for example, a jump in time occurs during the execution of a timer calibration loop the guest will calibrate the timer to run at a higher rate than it actually should do.

The first and most obvious alternative is to stop passage of time within the VM while it is preempted and thus let time pass continuously. The problem here is that time will gradually drift away from the host's clock. Residing within a network the guest's clock will be constantly out of sync with the others, which invalidates sessions and may provoke processes to crash or show incorrect behaviour. [4]

So far the most popular virtualization environments, like KVM or VMware, do not provide a combined solution for these issues. They have either the problem of jumps or of drifts. The goal of this work is therefore to propose a concept of virtualized time that does not have either problem. We work on x86 and focus on the virtualization of the time stamp counter (TSC), which is the most precise time source available from user space [5]. In spite of preemptions a virtual machine shall on the one hand perceive continuous passage of time and its clock shall on the other hand not drift arbitrarily far from physical time, assuming that a preemption can not be arbitrarily long.

3 Background

3.1 A short Introduction in Virtualization

In computing virtualization refers to the attempt to emulate a virtual hardware or software resource rather than providing a real one. This work concentrates on hardware virtualization, where a virtual machine is created that acts like a real computer to an operating system or other software. The physical machine and the software that establishes the VM is called host, the software running on top is called VM or guest. The host's software consists of two components:

The Hypervisor: The lowest software layer that runs in ring 0, the most privileged protection domain. It can be seen as an operating system kernel with features to provide virtualization. In our environment NOVA represents the hypervisor.

The Virtual Machine Monitor: The part that actually emulates the hardware to a guest. If clearly separated from the hypervisor the latter runs in ring 3, the least privileged protection domain. If the VMM is part of the hypervisor it runs as well in ring 0. Vancouver is our VMM.

In many popular virtualization tools, like KVM, the hypervisor and the VMM are not clearly distinguished from each other and the terms are often used synonymously. However within NOVA and Vancouver we have a clean separation between the hypervisor and the VMM. Thus the VMM runs in ring 3, which is an important fact.
Different types of hardware virtualization include [8]:

Full Virtualization: A complete simulation of the real hardware, so that the guest operating system does not notice to be virtualized.

Paravirtualization: The real hardware is not simulated but each guest is executed in its own isolation domain, as if they are running on a separate system. The guest has to be modified to run in such an environment and is therefore virtualization aware.

3.2 Time in a Computer System

Today's systems usually consist of several time sources, which measure passage of time since system boot independently from each other in one of two possible ways: In the first possibility the operating system programs a hardware timer to periodically interrupt at a known rate, then handles these ticks and increments a counter to keep track of how much time has passed. Examples for such timer devices on a PC platform are the programmable interval timer (PIT), the local advanced programmable interrupt controller (LAPIC) timers and the high precision event timer (HPET). In the second possibility a hardware device increments its internal counter as time units pass and the OS asks to read that counter whenever needed. The advantage of tickless timekeeping is certainly that it does not waste CPU performance, yet it is only effective if a suitable hardware counter is available that runs on a constant rate and does not overflow unnoticed. Examples for such timer devices are the time stamp counter (TSC) and the real time clock (RTC). [7]

3.3 Time in a Virtual Machine

Time in a real system usually meets the requirements assumed by the operating system: it moves regularly and irreversibly into one direction. In a virtualized OS this might no longer be true:
If several guests run on one CPU, they have to share the available execution time but in contrast to a non-virtualized OS a guest has no possibility to control if and when it gets suspended. The time spent in preemption is irreversibly lost for the guest. It is actually not even aware of a preemption, as it assumes to be the only OS an a real machine. [2] If the guest OS is allowed to read time directly from the hardware device it will observe a large and unexpected jump in time after being resumed from suspend (Fig. 1).

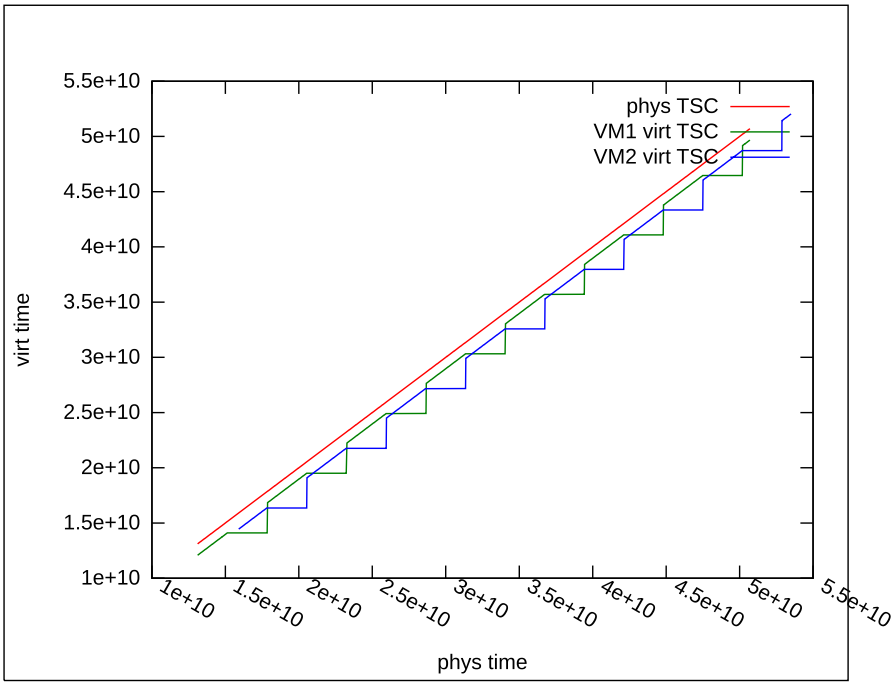

Figure 1: VM 1 and VM 2 read time directly from the hardware device, e.g. the TSC. They are preempted alternately but physical time still passes by so that they observe a large gap between their last read before the preemption and their first read after the resume.

However if passage of time is simply stopped while the guest is suspended virtual time will constantly drift away from the actual physical time (Fig. 2).
To avoid such problems a concept of virtual time is needed in the first place. A widely used software approach is to read physical time and add the value of a certain function to it. This function usually describes how far virtual time is behind physical by an adjustable offset. [2, 3]. Intel and AMD even provide hardware support for this concept (described in detail in Chapter 4.1). In common virtualization tools virtual time is usually initialized on the creation of a virtual CPU in order to contain the host's time at the point when the virtual machine booted. Whenever a guest requests current time the host traps that attempt, reads physical time from whatever hardware time source is available and returns physical time plus the function value to the guest.

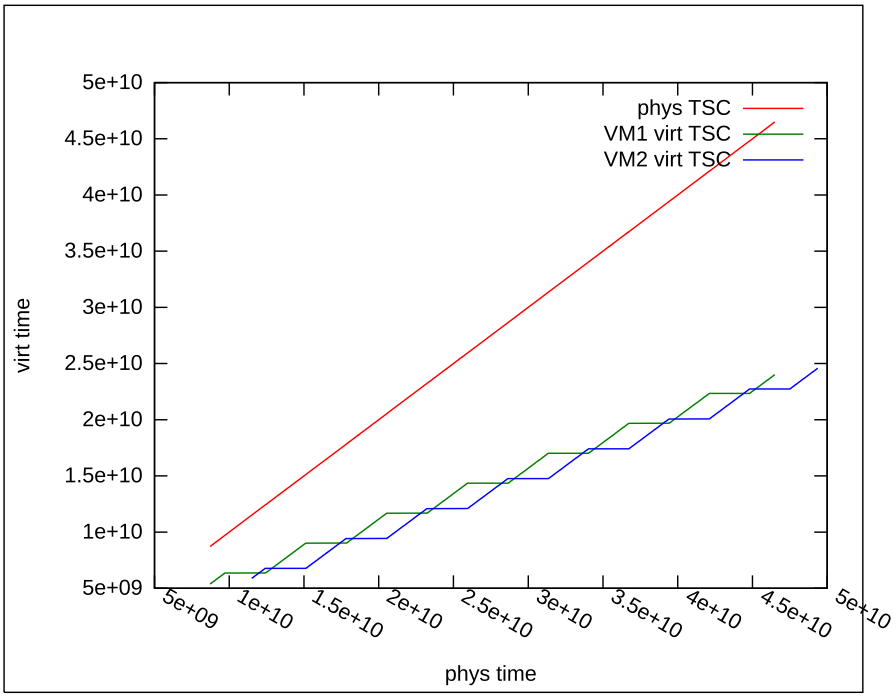

Figure 2: The clocks of VM 1 and VM 2 are offset by a function that accumulates the time they did not run since the host was booted. Time spent in suspend is not visible in the guest and no gap appears between the last read before the preemption the first read after the resume. However the guest's time constantly drifts away from the host's time.

4 Related Work

In this chapter we compare today's hardware and software attempts to virtualize time with the focus on the problem of jumps and of drift. All the presented ideas except VMWare's are based on virtualizing time by offsetting physical time, though in different manners.

4.1 Time Virtualization Support in Hardware

Hardware virtualization allows multiple operating systems to share a processor and its resources in a safe and efficient way. Intel and AMD provide extension fields for x86 architectures in their hardware that allow the guest's visible time to be offset by a constant, typically initialized with the host's system time when the guest OS was booted. [9] However these extension fields are not a solution for time virtualization but only a mechanism to support and improve software solutions. The function that adjusts the constant in the extension field is still to be implemented and otherwise the extension would suffer from the problem of jumps as well.

4.1.1 Intel VT-X

Intel VT-X are hardware features built in especially to facilitate virtualization. Regarding the need to virtualize time, VT-X provides trapping of the RDTSC and RDTSCP instructions. When a guest executes these instructions they cause a VM exit, the hypervisor or the VMM takes over to handle them and then resumes the guest. Virtual time is kept by adding an offset specified in a field in the VT-X control block (VMCS) to the host's time. [2] VMCS can only be accessed by privileged instructions but the hypervisor offers an API to allow safe and secure access to the control block (and thus to the offset) from user space.

4.1.2 AMD-V

AMD-V provides similar features as VT-X. The RDTSC and RDTSCP instructions can be trapped and an offset field is available in the AMD-V control block (VMCB). [2]
Moreover AMD allows the hypervisor to control the guest's view of the time stamp counter by writing to the TSC Ratio MSR. With its content the CPU scales the TSC value when it is read via the RDTSC or RDTSCP instruction inside a virtual machine. This facility is intended to let the TSC appear consistent to a guest that is migrated to another core having a different P_0 frequency (P_0 stands for the highest performance state of a CPU). The TSC value read by the guest is computed using the TSC Ratio MSR along with the TSC_OFFSET field from the VMCB so that the actual value returned is:

$$\textit{TSC\_VALUE\_GUEST} = P_0 \cdot \textit{TSCRatio} \cdot \textit{CURRENT\_TIME} + \textit{VMCB.TSC\_OFFSET}$$

This feature can be used to scale passage of time in the guest system whenever virtual time lags behind physical time and thus allow to synchronize virtual and physical time already in hardware without the problems mentioned at the beginning. Such a component is yet only available in certain AMD architectures. [1, p. 506]

4.2 Examples of Time Virtualization in VM Software

In section 4.1 we learned that even nowadays the time virtualization issue has to be solved in software in order to provide a time concept that neither allows time to jump nor to drift, as no sufficient hardware solution is yet available. Below we have a closer look on today's software solutions and examine how two popular hypervisors deal with the guest's time.

4.2.1 KVM

KVM virtualizes time by providing guests a paravirtualized clock device that lets the guest read the host's wall clock time. It shall be only mentioned casually that a similar algorithm is implemented in Xen [6, p. 8]. The guests register a memory page in their address space to contain kvmclock data. This page has to be present throughout a guest's whole life and contains the following structure:

```
struct pvclock_vcpu_time_info {
      u32   version;
      u32   pad0;
      u64   tsc_timestamp;            //the guest's TSC at time t
                                        //(result of rdtsc + tsc_offset)
      u64   system_time;              //the host's wall clock time at time t
      u32   tsc_to_system_mul;        //scale to multiply tsc_delta with
                                        //in order to obtain nanoseconds
      s8    tsc_shift;
      u8    flags;
      u8    pad[2];
} __attribute__((__packed__));
```

The hypervisor updates it until it is explicitly disabled or the guest is turned off. The guest on its part then calculates its current time the following:

$$\texttt{time\_in\_nsec} = \texttt{system\_time} + (\,\texttt{rdtsc()} - \texttt{tsc\_timestamp}\,) \cdot \texttt{tsc\_to\_system\_mul}$$

The hypervisor is free to change the field `tsc_to_system_mul` in face of events like CPU frequency change, migration, etc. This offers the guest a cheap way to access wall clock time but does not yet prevent it from observing jumps. The documentation does not describe any attempt to support continuous virtual time. [3]

4.2.2 VMWare

VMware virtualizes all timing devices available in a computer using several techniques to keep differences in timing performance as small as possible. Their solution "[...] allows the many timer devices in a virtual machine to fall behind real time and catch up as needed while remaining sufficiently consistent with one another so that software running in the virtual machine is not disrupted by anomalous time readings." [7, p. 4]
VMware's TSC shows virtual time and keeps up with the other timing devices in the virtual machine. It can nevertheless fall behind real time: If time is handled via interrupt counting, a backlog of interrupt delivery accumulates whenever the guest is suspended. To catch up interrupts are served at a higher rate until the backlog is cleared. If the backlog grows larger than 60 it is simply reset to zero. At this point VMWare accepts that virtual time falls behind physical time because missed timeouts are not delivered anymore. Anyhow the virtual TSC advances even when the virtual CPU is not running as it does not count instruction cycles on the virtual CPU. When a virtual machine is booted its virtual TSC is set to increment at the same rate as the host's TSC. This rate is kept throughout the guest's life, but one can force it to a specific value and hence scale passage of time as well. [7]

5 Design

In Chapter 4 we described common mechanisms for time in a VM: either it is not virtualized at all and the VM reads time directly from the timing device or passage of time is stopped whenever the VM does not run. The first approach has the problem of jumps in time after a preemption (Fig. 1) the second has the problem of time drifting arbitrarily far from the host's time (Fig. 2). Our goal is to provide a solution that creates the impression of guest time passing continuously without drifting too far from the host's time. What we need is therefore a solution that combines the two possibilities named above: A time concept that speeds up passage of virtual time after the guest was suspended until its time caught up with physical time.
As virtual time is determined by offsetting physical time, the function to be implemented has to decrement this offset stepwise to zero. Ideally this results in a linear approximation because any virtualization unaware operating system expects time to pass in that manner, as explained in Chapter 3.2.
On closer examination a speed up results into small jumps in time as well: It takes already a few time units to read a time source, so even if a thread runs alone on bare metal and does nothing else but reading a time source it will never see two exactly consecutive time units. Hence if we speed up passage of time, it means to let the time source appear to run at a higher rate than it usually does, for example via an acceleration factor. As a result the time gap in between two time units enlarges. If we want virtual time to catch up with physical time we have to live with that. In future work it has to be found out how large these time jumps can grow without affecting the correctness of the work of the guest.
To fully virtualize time we also need to consider the timer programming issue. If the VMM uses physical time to deliver a virtual timeout it might be triggered too early, because the guest can get preempted in between the programming and the expected delivery. Details are explained in Chapter 5.5.

5.1 The Naive Approach and Why It Does Not Work

The first idea is to obtain the current host time p_0, the current guest time v_0 and the point p_e when the guest's time slot ends. With them we can determine the time that is leftover to the guest in his current time slot ($p_e - p_0$). Furthermore we can calculate the acceleration factor m that determines the speed up, so that virtual time has caught up with physical time at point p_e (Fig. 3). When the guest later tries to read a timer device the VMM handles the request: it reads current physical time p_c , computes the corresponding virtual time v_c using the previously calculated acceleration factor and changes the guest's virtual time respectively. Therefor VMM computes $v_c = m \cdot (p_c - p_0)$.

Unfortunately this does not work. The approach assumes the time that is left to the guest until its time slot is over but this information is not available. Even the kernel does not know in advance when a guest's time is up, because a higher prioritized job might arrive and pretend the guest. The VMM, located in userland, does not know anything about time slots and possible preemptions at all. The only information the VMM can access is the current physical and the current virtual time.

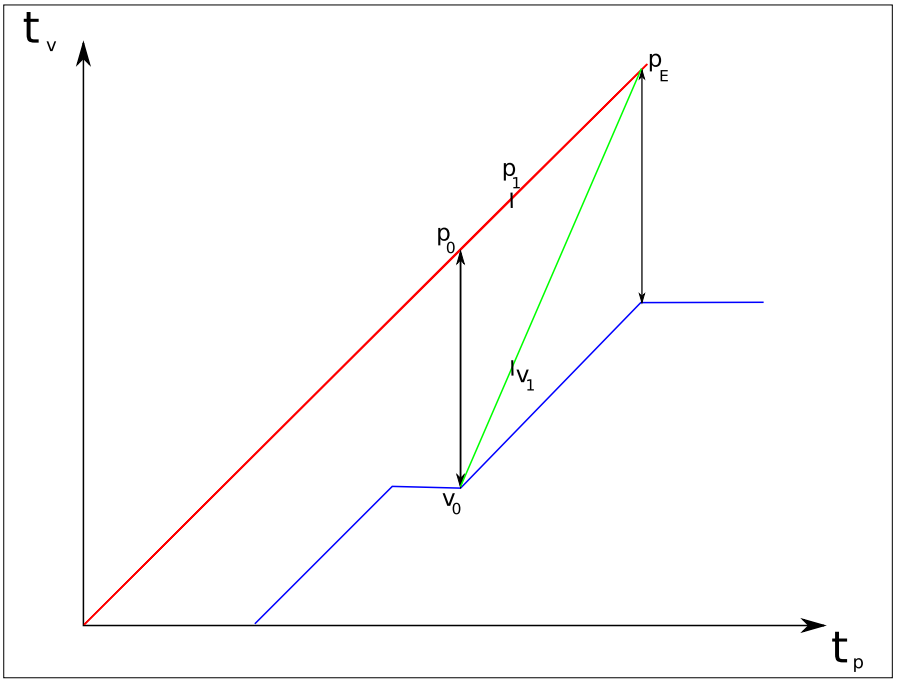

Figure 3: In this idea the acceleration factor m is calculated with information we do not have: p_e, the point in time when the guest's time slot expires.

Thus our solution has to be able to manage with the given circumstances, i.e. the little information we have a priori available.

5.2 What Information is actually Available in Userland?

Our first idea in Chapter 5.1 failed due to information that are not available. The question is hence: What information is actually available in userland?
The only data the VMM can certainly rely on is the current physical time p_0 and the corresponding virtual time v_0 or in other words the current offset. Every other knowledge is based on estimates which in turn are based on monitoring the system's behavior.

5.3 Our Solution

In our approach we decided to use as much information we can certainly rely on as possible and to avoid internal states as much as possible. Reliable information is the current physical time p_0 and the current virtual time v_0. Because the VMM does not know the point in physical time when virtual time should have caught up (which is the end of the guest's running time slot p_e in the naive approach) we define a period x in which we want the VMM to adjust virtual to physical time. Furthermore we need a value n to determine in how many steps the VMM shall adjust. This is indicated by the dashed line in Figure 4. With these values we can establish an initial adjustment size s_0 which has to be adapted by the acceleration factor.

$$s_0 = \frac{x}{n}$$

This factor depends on the difference of p_0 and v_0.

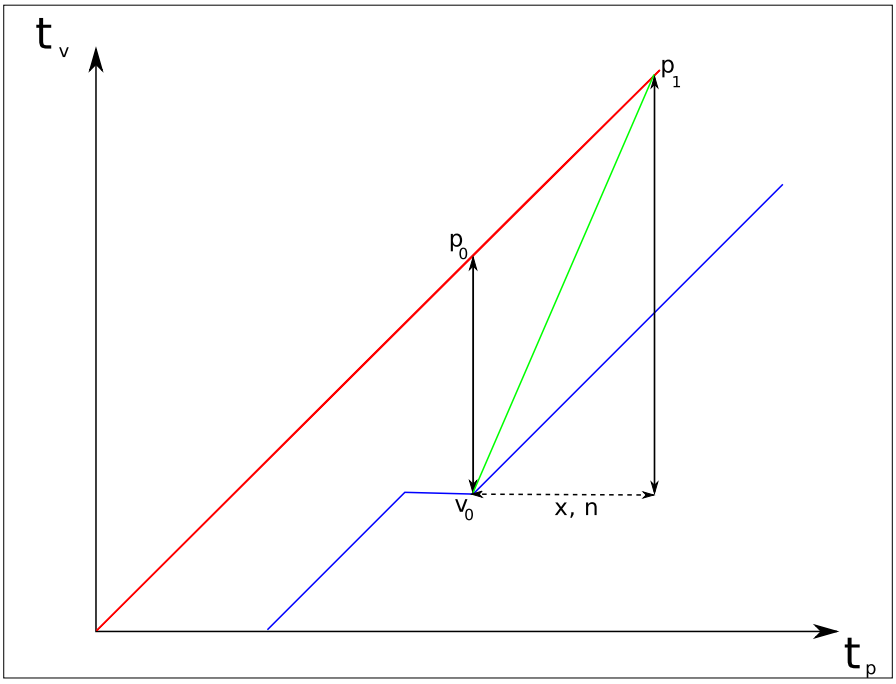

Figure 4: We define a period x and a number of decreasing steps n to circumvent the lack of p_e. Together with p_0 and v_0 we can calculate the acceleration factor and the actual adjustment size s. This adjustment is added to the current virtual time and thus produces a small jump.

Now the VMM can calculate the point p_1 when virtual time should have caught up with physical time.

$$p_1 = p_0 + x$$

In contrast to the first approach p_1 is not necessarily the end of the time slice meaning it does not have to equal p_e. At any point in time our second approach adjusts the offset towards x time units in the future. Consequently the VMM does not need knowledge anymore about its time slice. Having computed p_1 we can now calculate m (Fig. 4).

$$m = \frac{p_1 - v_0}{x}$$

To compute the actual adjustment size we multiply the initial adjustment size by the current

acceleration factor.

$$
\begin{aligned}
s &= s_0 \cdot m \\
s &= \frac{x}{n} \cdot \frac{p_1 - v_0}{x} \\
s &= \frac{p_1 - v_0}{n}
\end{aligned}
$$

It is important to understand that s does not affect the tick rate of the time source, i.e. the hardware device itself. It is a correction done in software, which affects how time is visible in the guest. For that reason virtual time itself passes after our calculation as fast as it did before but the difference between physical and virtual time is decreased that way (Fig. 5). This results into small jumps, because we add s to v_0 in order to catch up.

So far we made a small mistake in our calculation of s: We forgot that x time units only pass once. In our calculation x time units pass in physical and in virtual time, which is not correct because in our concept, virtual time is determined by just offsetting physical time. Thus we have to correct our equation:

$$
\begin{aligned}
p_1 &= v_0 + x + n * s \\
p_1 &= p_0 + x \\
p_0 + x &= v_0 + x + n * s \\
s &= \frac{p_0 - v_0}{n}
\end{aligned}
$$

After that we can adjust virtual time by s (Fig. 5). As explained already s has to be small enough so that the adjustment can happen without the guest noticing it.
Yet no internal state is kept so that the calculation of the next s is independent of its last value. By that we do not have to care whether the guest was preempted between two steps. The adjustment algorithm only considers the current circumstances. Thus the computation of the next step does not differ from the last one at all. The algorithm takes current physical time as p_0 and current virtual time as v_0 and determines a new s (Fig. 6).
The expected approximation curve is shown in Figure 7.

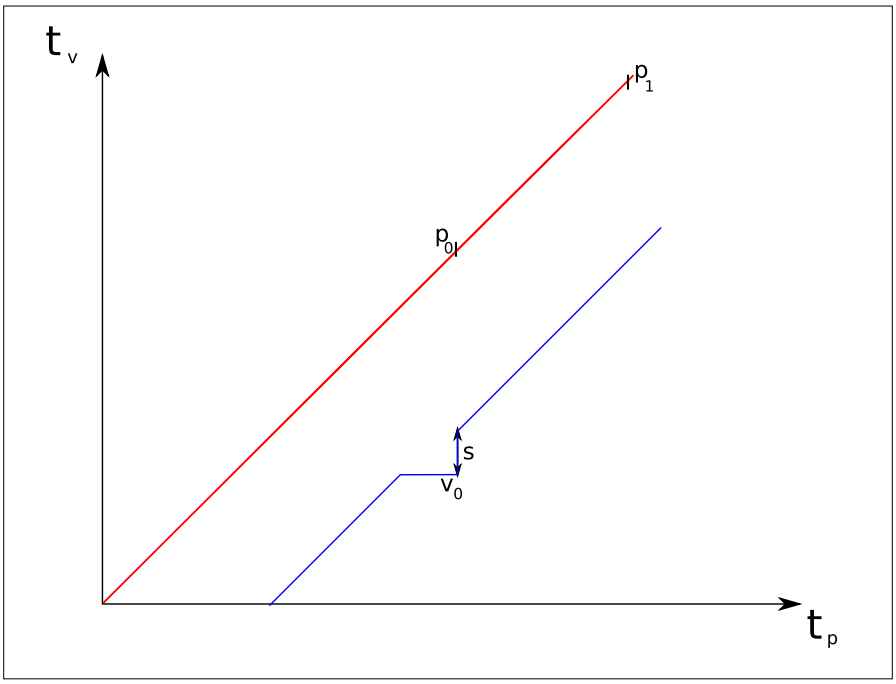

Figure 5: The difference between p_0 and v_0 is decreased by s. If that is done whenever the guest wants to read time, its virtual time will eventually have caught up with physical time.

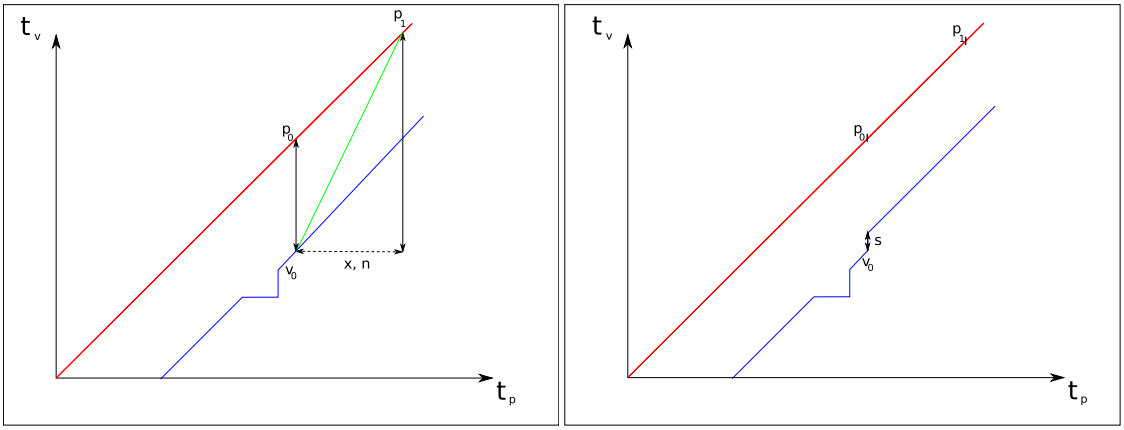

Figure 6: This shows an adjustment calculation after one step has already been made. The current calculation of s is independent of the past, it only considers the current p_0 and v_0.

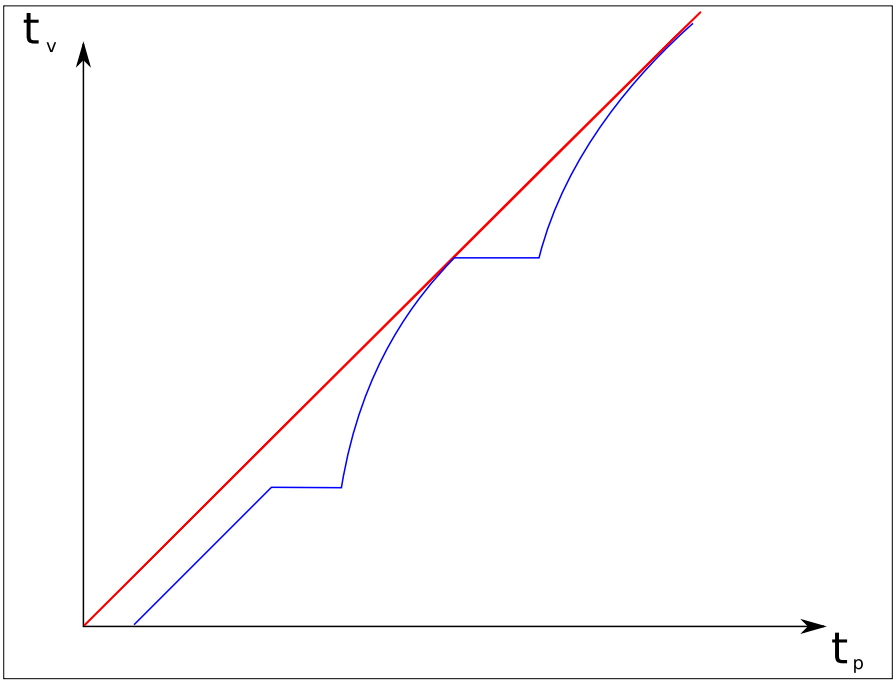

Figure 7: The expected development of virtual time in our approach is parabolic. Because the difference between p_0 and v_0 is decreased in every step but we still assume to have n steps left until virtual time shall have caught up, the adjustment size shrinks with every calculation as well. Thus the difference of p_0 and v_0 is decreased by smaller and smaller values.

5.3.1 How to choose the period x and the number of decreasing steps n?

The adjustment size s, which depends on n, represents the base of our approach and should be chosen to match the conditions at runtime. s should neither be too large because this would lead into larger jumps than necessary, nor should it be too small because then we would never achieve virtual time to catch up. To adapt s at runtime we have to modify n dynamically.
Given that our algorithm is executed only if the guest tries to consult a time source a reasonable value for n would be a count on the guest's time requests within one time slot, so that virtual time has caught up every time right before its slot expires. Without the knowledge of time slots existing however we run into the same problem as in the naive approach. The VMM does not notice if and when it gets interrupted. Anyhow it can determine the interval in which x physical time units have passed and can count the attempts to read time within this span. We call this interval x_i . The number of attempts n_i seen in period x_i is used during the period x_{i+1} to represent n. By that we get a rough prediction of how many times the guest wants to read a time source in the current period and can accommodate n if necessary. The disadvantage of a dynamic n is though that we have to keep an internal state and that such an n is based on estimates.
Because the final adjustment size highly depends on n and because n is the only control variable in our algorithm it is worth to put effort into finding a sophisticated solution to presume the correct value for n. What happens if we err on this value is explained in Chapter 7.1.1.
However even if we manage to estimate the best value for n large time jumps can still occur. We cannot guarantee that a guest is not kept in preemption for an arbitrarily long period of time. In future work it should be considered to extend the number of steps in which virtual time shall have caught up, if the difference between p_0 and v_0 is currently very large.

5.3.2 Advantages and Disadvantages of our Solution

An important advantage of our algorithm is that the results are based on reliable input, not on estimates. We can foretell how the ideal curve has to look like. Even after introducing a state to determine n at runtime, the results should still match our expected curve. If this is not the case, e.g. because an adjustment exceeds a certain limit, n can be adapted immediately and the adjustment can be recalculated before returning virtual time to the guest. Furthermore n is the only control variable which has several benefits. We know on which part we have to focus in order to develop an algorithm of the best possible approximation. In addition we know instantly if n is too small or too large by just considering the latest adjustment calculation. If the adjustment was too large we have to increase n otherwise we have to decrease it. Even if we choose n too large, virtual time cannot drift arbitrarily far away from physical time, assuming that the guest is not suspended for an arbitrarily long time. The situation will eventually end up in a steady state, because we always adjust a certain percentage of the difference between virtual and physical time. If the adjustment is too small and we do not correct it by changing n, virtual time will drift away until the difference to physical time corresponds to that wrong n. And finally the internal state we have to maintain is a small one.
The main disadvantage of our solution is that we can not achieve linear approximation even on ideal conditions. However this should not matter if the adjustments happen totally unnoticed by

the guest.

5.4 Another Idea – Estimating the Current System Load

By monitoring physical time that passes in relation to the time the VMM actually runs it can estimate how many threads are currently computing on that CPU. If two VMs with the same priority run 100 % busy on one CPU, virtual time within a guest passes only half as fast as physical time (Fig. 2) and has to be sped up by factor 2 to stay in step with physical time. So the central idea is to estimate the number of threads by computing the own CPU share and take this as factor to speed up virtual time. The most important advantage of this approach that we achieve a linear approximation if we estimate correctly. The downside however is that we have to rely on estimations and build an algorithm that maintains an internal state correctly. Furthermore the approach estimates system load by observing a certain duration of the past so that it does not react instantly but delayed to changes of the circumstances.

5.5 Timer Programming

To fully virtualize time we need to consider the timer programming issue. The VMM has to emulate a timer interface as well as the timing device itself. If the guest operating system is asked to trigger a timer interrupt in y time units, the VMM's timer emulation has to work out at which physical point in time this timeout can occur at the earliest (Fig 8). Because virtual time is sped up by our approach, y virtual time units do not necessarily equal y physical time units. The VMM itself then programs a physical timeout to get notified when it is time to deliver the virtual timeout to the guest. When this point p_t is reached the VMM checks if y units of virtual time have actually past in the guest (or if it got suspended in between). If so the interrupt is delivered. If not the VMM has to analyze how many time units of y are left to pass and estimate a new physical time to trigger that interrupt (Fig. 9 and 10).
In order to emulate a timer interface with virtual time we reuse parts of the approach for time virtualization. Again the VMM reads current physical and virtual time (p_0, v_0) and calculates the acceleration factor m as described above (Fig. 4). Then it calculates how many physical time units pass during y virtual time units, taking the m factor into account, and adds that to p_0 to get p_t.

$$p_t = \frac{y}{m} + p_0$$

When p_t is reached the VMM has to determine if y time units have already passed in the guest. If not a new p_t has to be calculated.
Several interesting questions arise here: The acceleration factor m diminishes already while the guest is running so that virtual time does not pass linearly. It might happen that y time units do not pass in the guest until p_t, even though the guest was not preempted. Then the timer would fire too early, as well as if the guest is preempted in-between the programming and v_t. We evaluate that in Chapter 7.3. Another question is what happens if the difference of p_t and v_t is very large and an adjustment calculation is triggered before v_t is reached in the guest. Also it might be that the guest executes `rdtsc()` only a moment before v_t, which results into an adjustment calculation. In both cases it the adjustment s can possibly jump over v_t.

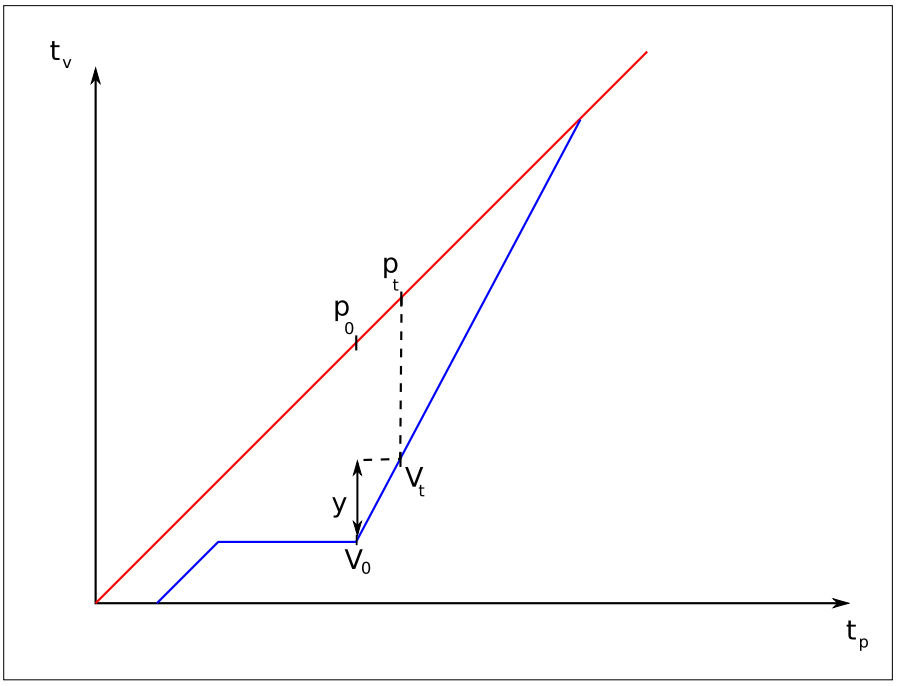

Figure 8: The VMM has to determine which physical time corresponds to when the timer should fire. It programs itself a timer for that moment.

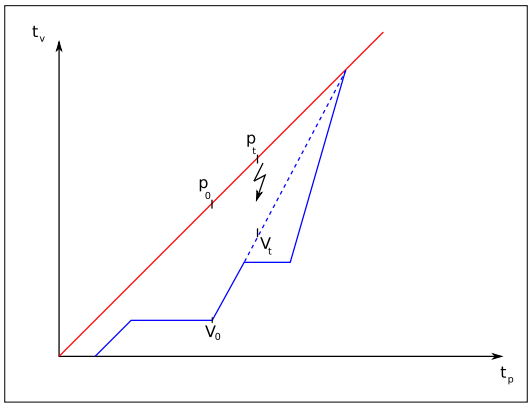

Figure 9: When the physical timer triggers the VMM has to analyze if the guest was suspended since it programmed its virtual timer. Here only a part of y virtual units have passed so far.

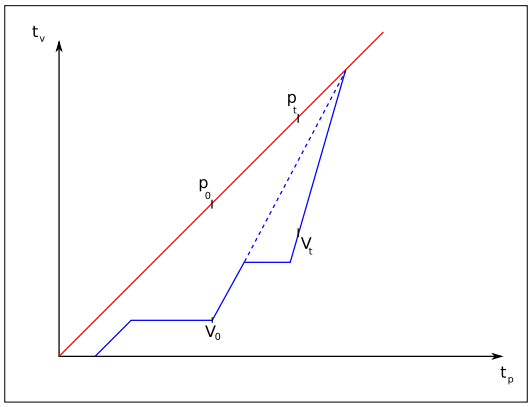

Figure 10: Because the guest was preempted the VMM has to recalculate the physical moment when the virtual timer shall be delivered. Such a situation can occur several times.

6 Implementation

So as to comprehend the implementation details of timing device virtualization in NOVA and Vancouver it is important to understand how time was handled before. Again this work focuses on the virtualization of the TSC being the device that is accessible from user space and has a sufficient precision. Therefore we only describe the TSC implementation. In Chapter 3 we already described the two elementary ways to provide time. The first way is to let the guest use the host's time. The second is to stop passage of time in the guest while it is preempted.
Both NOVA and Vancouver provide one of the two mechanisms but not a combination of them. NOVA stops time while the guest does not run. It maintains a data structure to describe virtual time by offsetting physical time by a constant value. This offset is initialized with the host's TSC value at the moment when the virtual CPU is powered on, so that virtual time starts at zero. When a guest is suspended and resumed, NOVA adds the length of the preemption to the guest's TSC offset and thus time appears to have stopped meanwhile.
In Vancouver the same data structure is accessed to maintain virtual time, which means the concept of virtual time is used in the same way: physical time is offset by a constant. However Vancouver keeps this offset at the host's TSC value on virtual CPU creation. If Vancouver and the guest were suspended, they are resumed by NOVA with an adjusted offset, meaning virtual time seems to pass continuously. However if the guest is now trying to read the TSC, Vancouver resets this offset back to its initial value, i.e. the point when the virtual CPU was created. This means that time in Vancouver starts at zero but the guest will observe a time jump after a preemption. The intention behind that setting and resetting is to allow various implementations in Vancouver. If the hiding of the time jumps was not done initially by the kernel, that would not be possible that easily anymore in userland.
The important difference to an approach that wants virtual time to catch up with physical time is that the TSC offset is not changed, while the guest is running. The offset is only set twice: one time in NOVA, right before a guest is resumed and one time in Vancouver, right after a guest is resumed.

When we said that Vancouver resets the TSC offset back to its initial value only once, this was not quite correct. Actually this code path is executed whenever the guest tries to read the TSC but the offset is always set back to the same value, i.e. the initial one. This is the place where we apply our approach: Instead of always resetting the TSC offset to the point when the virtual CPU was created, we diminish it by letting Vancouver compute the adjustment size s (Chapter 5.3) and subtract it from the offset. As said this is triggered whenever the guest tries to read the time stamp counter. That way we decrease the offset stepwise towards zero. Because the TSC offset only enlarges while a guest is suspended we will arrive at zero, if the estimated n (the number of adjustment steps – Chapter 5.3.1) matches to the guest's attempts to read the TSC. For that reason we observe these attempts for a fixed period x_i, count them and use this value n_i as n for the next period x_{i+1}.

Some NOVA/Vancouver-specific details are worth mentioning here because they contradict the assumptions of our approach and require a corrective:
We assumed that our calculation is only performed when the guest tries to read the TSC though

this is not quite correct. Whenever a guest tries to execute an instruction it is not allowed to, e.g. `rdtsc()`, a VM exit is triggered and different flags are set to inform Vancouver about what it has to do now. This means when the guest calls `rdtsc()` a flag is set that tells Vancouver to adjust the TSC offset. However another flag exists, which sets all flags in the message transfer descriptor including the TSC flag. The flag that sets every other flag is used for example when the guest executes an IO-instruction. The following situation can occur: The guest first reads the TSC, then handles some IO-requests and last reads the TSC again. In all cases the adjustment calculation is performed because the set TSC flag instructs Vancouver to do so and the guest might observe a jump in time between the first and the last `rdtsc()` instruction. The guest does not take note of the adjustments in between because it did not try to read current time. If many small adjustments happen without the guest realizing them, they can possibly accumulate and may become observable for the guest even though each single one would to be small enough to pass unnoticed. It is left for future work to explore under which circumstances this problem occurs and when it starts to get significant.
Another difficulty can occur as follows: When the VM exit is carried out because the guest executed `rdtsc()` the current TSC offset is written to the data structure Vancouver maintains in order to provide virtual time. During the adjustment calculation Vancouver reads the host's TSC and uses it as current physical time p_0. However Vancouver can be preempted between the VM exit and the adjustment calculation so that the TSC offset does not match to current physical time anymore. This becomes a problem if the VMM tries to determine how much virtual time has passed in the guest. The following example illustrates the problem:

host TSC	TSC offset	Action
0	0	initial state on creation of the guest's virtual CPU
1	0	VM exit, VMM reads TSC offset $\mathit{off}_0 = 0$
		VMM is preempted for 8 time units, hypervisor adds to 8 offset
9	8	VMM is resumed and reads the host's TSC $\mathit{tsc}_0 = 9$ and restarts the guest with virtual time $\mathit{virt}_0 = 9$. It does not change the TSC offset, because it uses the one it read before it was preempted off_0.
10	8	VM exit, VMM reads TSC offset $\mathit{off}_1 = 8$ and the host's TSC $\mathit{tsc}_1 = 10$. Then it restarts the guest with virtual time $\mathit{virt}_1 = 2$

If such a situation occurs virtual time in the guest seems to go backwards for the VMM, because the offset grows faster than physical time. However the guest does not notice that so far. The problem becomes visible only when the VMM determines how long the guest ran by calculating $\mathit{virt}_1 - \mathit{virt}_0 = -7$ and uses that negative span to adjust the offset. The miscalculation of the adjustment due to the negative span negates the positive sign and thus the offset is enlarged instead of being reduced. This results into virtual time running backwards within the guest. Though easy to detect, this issue has to be considered during implementation.
Because our approach only considers the current difference between virtual and physical time this does not affect the correctness of the results. Concepts that rely on the offset together with physical time need a time stamp at the moment the VM exit occurred and use that instead of the

host's TSC for adjustment calculation. Also in our implementation for virtual timeouts we have to consider that, because the VMM has to check if y virtual time units have actually past in the guest before delivering the timeout (Chapter 5.5).

7 Evaluation

The following questions have to be answered by our tests:

1. Does virtual time pass in a parabolic curve as we expect it to, if we provide ideal conditions? (Fig. 7)
2. What happens if we choose n constantly too large?
3. How does the expected curve change, if we introduce a dynamic n but still keep ideal conditions?
4. What happens if we change our setting from ideal to real conditions?
5. How often the VMM has to reprogram its physical timeout until the virtual one can be delivered? (Fig. 8, 9 and 10)

We test our algorithm with two different scenarios: First we start two instances of Vancouver on one core with each having a VM on top that behaves ideally for our case, i.e. the guest executes `rdtsc()` in an endless loop. Second we keep the two Vancouver instances on one core but boot a Linux kernel instead. All diagrams, except Figure 17 show virtual and physical time in CPU cycles.

7.1 A VM Reading the TSC in a While Loop

7.1.1 Tests with a static n

For the first two experiments we extended the time slots that each VM has at maximum until it is preempted to 100ms. That way the asymptotic expansion we from theory is clearly visible. The comparison of Figure 7 with Figure 11 approves that our algorithm works as awaited.
We claimed that even if we choose n too large virtual time would not drift arbitrarily far away from physical time, assuming that the is not suspended for an arbitrarily long time. This is shown in Figure 12. Assuming that the guest wants to read the TSC frequently enough the algorithm eventually ends up in a steady state which depends on n (we can see n as the estimated attempts of a guest to read time) and the actual number of attempts to read a time source $\bar{n}$. n and $\bar{n}$ are closely related and should ideally match. If $n/\bar{n} > 1$ virtual time will not catch up with physical time but will stay slightly beyond, depending on the quantity of the quotient. This occurs because our adjustment s is a certain percentage of the difference between p_0 and v_0 depending on the number of steps n that we estimated to perform. If this percentage is too small, because we estimated to perform more steps than we actually did (meaning $n > \bar{n}$), we will not manage to adjust virtual time 100% to physical time. If $n/\bar{n} < 1$ virtual time approximates faster than necessary and the curve slopes upwards quickly.
Obviously it is difficult to estimate a good n at compile time because we do not know in advance how often a real world guest will try to read the TSC. For that reason we extended our algorithm to estimate a reasonable n at runtime as explained in Chapter 5.3.1.

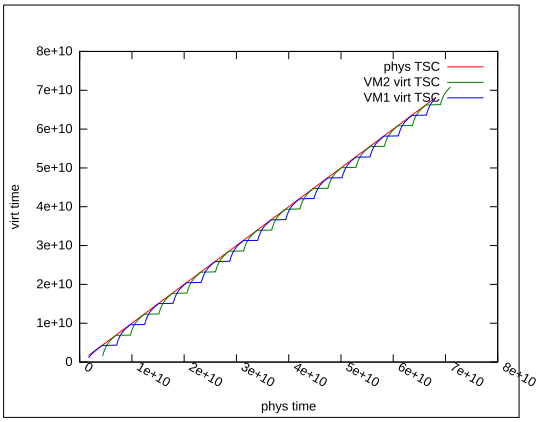

Figure 11: $n = 10$. A guest that does nothing else but reading the TSC provides the best case scenario for our algorithm. We see that the algorithm works as expected.

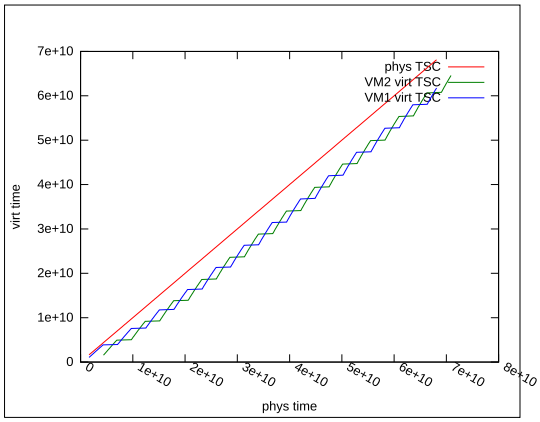

Figure 12: $n = 100$. The guest still loops `rdtsc()`. n is too large and virtual time stays behind. However it cannot drift arbitrarily far from physical time.

7.1.2 Tests with a dynamic n

Figure 13 and 14 show tests with a dynamic n. In Figure 13 the time slot size still amounts to 100ms. In 14 we reset the size back to NOVA's default of 10ms so that the test shows results under the default conditions in NOVA and Vancouver. The n for a period x_{i+1} of the size $4 \cdot$ *time slot* depends on $\bar{n}$, seen during the previous period x_i.

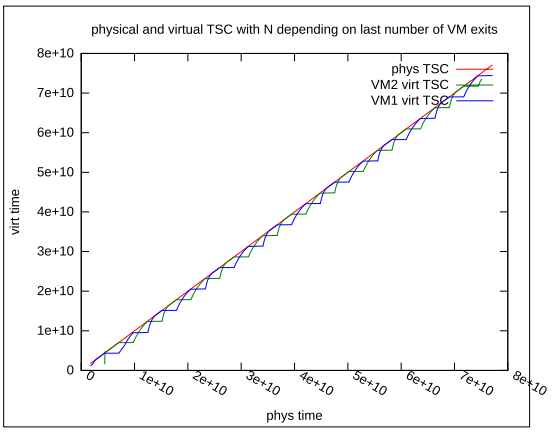

Figure 13: n depends on $\bar{n}$ during the last 400ms. The time slice is still extended.

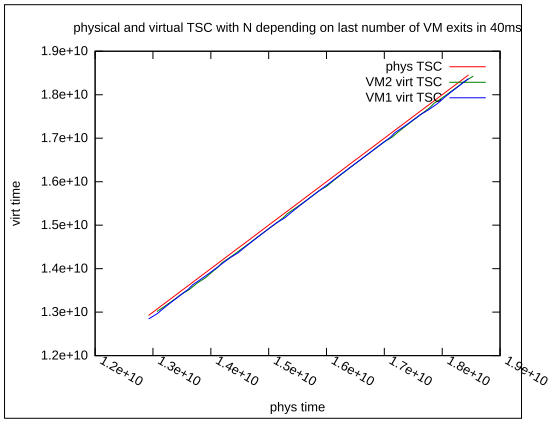

Figure 14: n depends on $\bar{n}$ during the last 40ms. The time slice is set back to default.

7.2 A VM booting a Linux kernel

Instead of letting our guests read the TSC in an endless loop we boot a Linux kernel in our two VMs. We re-extended the time slice to 100ms in order to have a clearly visible result. n is still determined dynamically. Both diagrams show the same setup but in Figure 16 we provided a concept of virtual time, while in Figure 15 we did not. This can be seen as a pre-post comparison. We see that the Linux kernels observe jumps in time after every suspend and resume without virtualization of time. Meanwhile the ones whose virtual time is adjusted stepwise to physical time see time passing faster than physical time in order to catch up.

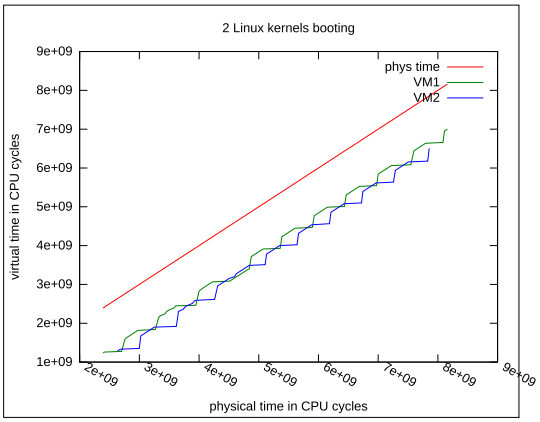

Figure 15: Two Linux kernels boot without virtual time. Their time jumps after every resume and then passes as fast as physical time.

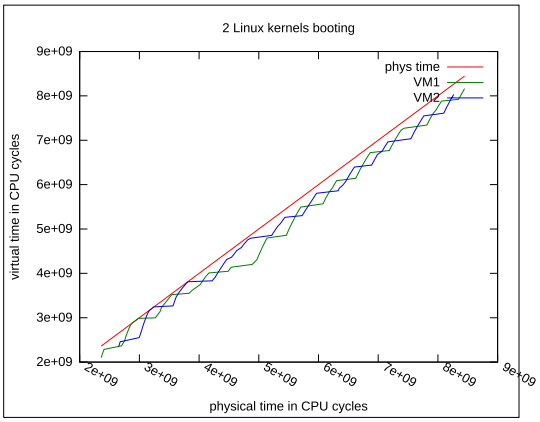

Figure 16: Two Linux kernels boot with virtual time. Their time does not jump but passes faster than physical time.

7.3 Timer Programming

With a virtual timer interface in Vancouver the guest can program itself a timer that corresponds to virtual time not physical and thus behaves like the guest expects it. The background arrangement is done by Vancouver as described in detail in Chapter 5.5. The interesting question is how often the VMM has to reprogram its physical timer so that it can deliver the virtual timeout correctly. In Figure 17 we show that this is almost never the case. A timer that is programmed once can be usually triggered without any reprogramming. The apprehension that the physical timer would have to be reprogrammed several

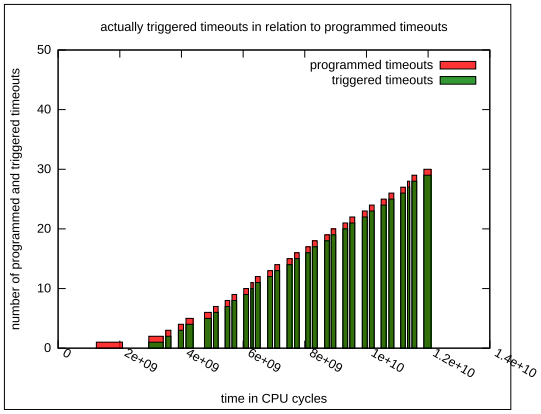

Figure 17: Programmed timers in relation to actually triggered timers. They are in a ratio of 1:1.

times is not confirmed.

7.4 Analysis

With our tests we showed that a simple concept as ours virtualizes time that sufficiently behaves in a way a virtualized guest operating system expects. The only thing we have to put effort into is a precise estimate of n which is based on a close evaluation of its development during the past. However if we guess n wrong we can not drift arbitrarily far from physical time (Fig. 12), in contrast to the approach that interprets system load as speed up factor. Let for example the speed up factor be two so that it doubles the size of one physical time unit in order to obtain a virtual time unit, time will pass twice as fast. If more guests run on that CPU, the speed up factor should however be higher. Without correction virtual time will constantly drift away from physical time in with this approach. Our approach eventually arrives in a steady state even if we do not ever correct n. In future work it would be interesting to compare different algorithms to estimate n and to test the virtual time concept while a guest performs a timer calibration.

8 Conclusion

In this work we provided the first concept of virtual time that combines the two previous solutions: With our approach virtual time neither passes with jumps nor does it drift arbitrarily far from physical time. The idea is easy to understand as we adjust virtual time with a variable percentage of the difference of virtual and physical time. We intially rely on facts not on estimates. The only thing we estimate is n in order to get a better curve development at runtime, if necessary. In addition we are almost independent from events in the past and do not have to maintain a large internal state. That helps us to keep a clear view of the control and data flow and is easier to debug than a solution that evaluates the past to calculate the speed up of virtual time.

As we use only one controllable variable that is furthermore the item on which the whole calculation highly depends, future work can focus on the development of a better algorithm to estimate and regulate that variable. A promising idea is to set n, if $p_0 - v_0$ exceeds a certain limit, to a higher value then $\bar{n}$, meaning n is too large at that moment. With every adjustment step, which decreases $p_0 - v_0$, n can be decreased as well. That might bring us closer to a linear approximation than we are now, but possibly keeps some of our idea's advantages. Also it would be interesting to exchange fixed for variable values, e.g. the length of x_i, and monitor if and how the system's time behavior changes. Furthermore it still has to be found out, how large an adjustment in one step may be in order to pass unnoticed and what happens if adjustments happen behind the guest's back as explained in Chapter 6.

Index of Notations and Abbreviations

$\bar{n}$	the number of steps that were actually performed in period x_i
m	acceleration factor
n	number of steps in which the VMM shall adjust virtual to physical time
n_i	the number of attempts a guest tried to read time during x_i
p_0	current physical time
p_1	physical point in time when virtual time shall have caught up
p_c	a point in physical time after m has been calculated
p_e	physical time when a guest's time slot expires
p_t	physical point in time when the virtual timeout shall be delivered
t_p	physical time
t_v	virtual time
v_0	current virtual time
v_c	the point in virtual time that corresponds to p_c
v_t	the point in virtual time when the guest expects the virtual timeout to fire
x	the period in which we want virtual time to catch up with physical time
x_i	the period in which the VMM counts the guest's attempts to read time
y	a guest programs a timeout in y virtual time units
HPET	High Precision Event Timer
KVM	Kernel-based Virtual Machine – a Linux kernel virtualization infrastructure
LAPIC	Local Advanced Programmable Interrupt Controller
MSR	Machine Specific Register
P_0	The highest performance state of a CPU
PIT	Programmable Interval Timer
RDTSC	CPU instruction to read the time stamp counter
RDTSCP	CPU instruction to read the time stamp counter. Additionally returns the number of the processor that executed it.

RTC	Real Time Clock
SMP	Multiprocessor System
TSC	Time Stamp Counter
VM	Virtual Machine
VMCB	AMD-V control block
VMCS	Intel VT-X control block
VMM	Virtual Machine Monitor
Xen	a native hypervisor developed by the University of Cambridge Computer Laboratory

References

[1] *AMD Architecture Programmer's Manual, Volume 2: System Programming*. Revision: 3.22. AMD. Sept. 2012.

[2] Zachary Amsden. *Timekeeping Virtualization for X86-Based Architectures*. Dec. 2012. URL: http://www.mjmwired.net/kernel/Documentation/virtual/kvm/timekeeping.txt.

[3] Glauber Costa. *kvmclock documentation*. Apr. 15, 2010. URL: http://lkml.org/lkml/2010/4/15/355.

[4] *KVM virtual timing management*. RedHat. URL: https://access.redhat.com/knowledge/docs/en-US/Red_Hat_Enterprise_Virtualization_for_Servers/2.2/html/Administration_Guide/chap-Virtualization-KVM_guest_timing_management.html.

[5] Terry Lambert Sergiu Iordache. *TSC resynchronization*. Google. URL: http://www.chromium.org/chromium-os/how-tos-and-troubleshooting/tsc-resynchronization.

[6] The Xen Team. *Xen – Interface Manual*. University of Cambridge. URL: http://www.xen.org/files/xen_interface.pdf.

[7] *Timekeeping in VMware Virtual Machines*. VMware, Nov. 2011.

[8] Wikipedia. *Virtualization*. Dec. 26, 2012. URL: http://en.wikipedia.org/wiki/Virtualization.

[9] Wikipedia. *x86 Virtualization*. Dec. 26, 2012. URL: http://en.wikipedia.org/wiki/Secure_Virtual_Machine.