Virtualize Everything but Time

Timothy Broomhead Laurence Cremean Julien Ridoux Darryl Veitch
Center for Ultra-Broadband Information Networks (CUBIN)
Department of Electrical & Electronic Engineering, The University of Melbourne, Australia
{*t.broomhead, l.cremean*}*@ugrad.unimelb.edu.au,* {*jridoux, dveitch*}*@unimelb.edu.au*

Abstract

We propose a new timekeeping architecture for virtualized systems, in the context of Xen. Built upon a *feed-forward* based *RADclock* synchronization algorithm, it ensures that the clocks in each OS sharing the hardware derive from a single central clock in a resource effective way, and that this clock is both accurate and robust. A key advantage is simple, seamless VM migration with consistent time. In contrast, the current Xen approach for timekeeping behaves very poorly under live migration, posing a major problem for applications such as financial transactions, gaming, and network measurement, which are critically dependent on reliable timekeeping. We also provide a detailed examination of the HPET and Xen Clocksource counters. Results are validated using a hardware-supported testbed.

1 Introduction

Virtualization represents a major movement in the evolution of computer infrastructure. Its many benefits include allowing the consolidation of server infrastructure onto fewer hardware platforms, resulting in easier management and energy savings. Virtualization enables the seamless migration of running guest operating systems (guest OSs), which reduces reliance on dedicated hardware, and eases maintenance and failure recovery.

Timekeeping is a core service on computing platforms, and accurate and reliable timekeeping is important in many contexts including network measurement and high-speed trading in finance. Other applications where accurate timing is essential to maintain at all times, and where virtualization can be expected to be used either now or in the future, include distributed databases, financial transactions, and gaming servers. The emerging market of outsourced cloud computing also requires accurate timing to manage and correctly bill customers using virtualized systems.

Software clocks are based on local hardware (oscillators), corrected using synchronization algorithms communicating with reference clocks. For cost and convenience reasons, reference clocks are queried over a network.

Since a notion of universally shared absolute time is tied to physics, timekeeping poses particular problems for virtualization, as a tight 'real' connection must be maintained across the OSs sharing the hardware. Both timekeeping and timestamping rely heavily on hardware counters. Virtualization adds an extra layer between the hardware and the OSs, which creates additional resource contention, and increased latencies, that impact performance.

In this paper we propose a new timekeeping architecture for para-virtualized systems, in the context of Xen [1]. Using a hardware-supported testbed, we show how the current approach using the *Network Time Protocol* (NTP) system is inadequate, in particular for VM migration. We explain how the *feed-forward* based synchronization adopted by the *RADclock* [14] allows a *dependent clock* paradigm to be used and ensures that all OSs sharing the hardware share the same (accurate and robust) clock in a resource effective way. This results in robust and seamless live migration because each physical host machine has its own unique clock, with hardware specific state, which never migrates. Only a stateless clock-reading function migrates. We also provide a detailed examination and comparison of the HPET and Xen Clocksource counters.

Neither the idea of a dependent clock, nor the *RADclock* algorithm, are new. The key contribution here is to show how the feed-forward nature, and stateless clock read function, employed by the *RADclock*, are ideally suited to make the dependent clock approach actually work. In what is the first evaluation of *RADclock* in a virtualized context, we show in detail that the resulting solution is orders of magnitude better than the current state of the art in terms of both average and peak error follow-

ing disruptive events, in particular live migration. We have integrated our work into the *RADclock* [14] packages for Linux, which now support the architecture for Xen described here.

After providing necessary background in Section 2, we motivate our work in depth by demonstrating the inadequacies of the status quo in Section 3. Since hardware counters are key to timekeeping in general and to our solution in particular, Section 4 provides a detailed examination of the behavior of counters of key importance to Xen. Section 5 describes and evaluates our proposed timing architecture on a single physical host, and Section 6 deals with migration. We conclude in Section 7.

2 Background

We provide background on Xen, hardware counters, timekeeping, the *RADclock* and NTP clocks, and comparison methodology.

To the best of our knowledge there is no directly relevant peer-reviewed published work on timekeeping in virtualized systems. A valuable resource however is [21].

2.1 Para-Virtualization and Xen

All virtualization techniques rely on a *hypervisor*, which, in the case of Xen [1, 2, 9], is a minimal kernel with exclusive access to hardware devices. The hypervisor provides a layer of abstraction from physical hardware, and manages physical resources on behalf of the guests, ensuring isolation between them. We work within the *para-virtualization* paradigm, whereby **all** guest OS's are modified to have awareness of, and access to, the native hardware via *hypercalls* to the hypervisor, which are similar to a system call. It is more challenging to support accurate timing under the alternative fully hardware virtualized paradigm [2], and we do not consider this here.

Although we work in the context of para-virtualized Xen, the architecture we propose has broader applicability. We focus on Linux OS's as this is the most active platform for Xen currently. In Xen, the guest OSs belong to two distinct categories: Dom0 and DomU. The former is a privileged system which has access to most hardware devices and provides virtual block and network devices for the other, DomU, guests.

2.2 Hardware Counters

The heart of any software clock is local oscillator hardware, accessed via dedicated counters. Counters commonly available today include the *Time Stamp Counter* (TSC) [6] which counts CPU cycles[1], the *Advanced Configuration and Power Interface* (ACPI) [10], and the *High Precision Event Timer* (HPET) [5].

The TSC enjoys high resolution and also very fast access. Care is however needed in architectures where a unique TSC of constant nominal frequency may not exist. This can occur because of multiple processors with unsynchronized TSC's, and/or power management effects resulting in stepped frequency changes and execution interruption. Such problems were endemic in architectures such as Intel Pentium III and IV, but have been resolved in recent architectures such as Intel Nehalem and AMD Barcelona.

In contrast to CPU counters like the TSC, HPET and ACPI are system-wide counters which are unaffected by processor speed issues. They are always on and run at constant nominal rate, unless the entire system is suspended, which we ignore here. HPET is accessed via a data bus and so has much slower access time than the TSC, as well as lower resolution as its nominal frequency is about 14.3MHz. ACPI has even lower resolution with a frequency of only 3.57MHz. It has even slower access, since it is also read via a bus but unlike HPET is not memory mapped.

Beyond the counter ticking itself, power management affects *all* counters through its impact on counter access latency, which naturally requires CPU instructions to be executed. Recent processors can, without stopping execution, move between different *P-States* where the operating frequency and/or voltage are varied to reduce energy consumption. Another strategy is to stop processor execution. Different such idle states, or *C-States* C0, C1, C2. . . are defined, where C0 is normal execution, and the deeper the state, the greater the latency penalty to wake from it [12]. The impact on latency of these strategies is discussed in more detail later.

2.3 Xen Clocksource

The Xen Clocksource is a hardware/software hybrid counter presented to guest OSs by the hypervisor. It aims to combine the reliability of a given *platform timer* (HPET here) with the low access latency of the TSC. It is based on using the TSC to interpolate between HPET readings made on 'ticks' of the periodic interrupt scheduling cycle of the OS (whose period is typically 1ms), and is scaled to a frequency of approximately 1GHz. It is a 64-bit cumulative counter, and is effectively initialized to zero for each guest when they boot (this is implemented by a 'system time' variable they keep) and monotonically increases.

The Xen Clocksource interpolation is a relatively complex mechanism that accounts for lost TSC ticks (it actively overwrites the TSC register) and frequency changes of the TSC due to power management (it main-

[1] TSC is x86 terminology, other architectures use other names.

tains a TSC scaling factor which can be used by guests to scale their TSC readings). Such compensation is needed on some older processors as described in Section 2.2.

2.4 Clock Fundamentals

A distinction must be drawn between the software clock itself, and timestamping. A clock may be perfect, yet timestamps made with it be very inaccurate due to large and/or variable access latency. Such timestamping errors will vary widely depending on context and may erroneously reflect on the clock itself.

By a *raw* timestamp we mean a reading of the underlying counter. For a clock C which reads $C(t)$ at true time t, the final timestamp will be a time in seconds based not only on the raw timestamp, but also the clock parameters set by the synchronization algorithm.

A local (scaled) counter is not a suitable clock and needs to be synchronized because all counters *drift* if left to themselves: their rate, although very close to constant (typically measured to 1 part in 10^6 or less), varies. Drift is primarily influenced by temperature.

Remote clock synchronization over a network is based on a (typically bidirectional) exchange of timing messages from an OS to a time server and back, giving rise to four timestamps: two made by the OS as the timing message (here an NTP packet) leaves then returns, and two made remotely by the time server. Typically, exchanges are made periodically: once every *poll-period*.

There are two key problems faced by remote synchronization. The first is to filter out the variability in the delays to and from the server, which effectively corrupt timestamps. This is the job of the clock synchronization algorithm, and it is judged on its ability to do this well (small error and small error variability) and consistently in real environments (robustness).

The second problem is that of a fundamental ambiguity between clock error and the degree of path asymmetry. Let $A = d^{\uparrow} - d^{\downarrow}$ denote the true path asymmetry, where $d^{\uparrow}$ and $d^{\downarrow}$ are the true minimum one-way delays to and from the server, respectively; and let $r = d^{\uparrow} + d^{\downarrow}$ be the minimal Round Trip Time (RTT). In the absence of any external side-information on A, we must guess a value, and $\hat{A} = 0$ is typically chosen, corresponding to a symmetric path. This allows the clock to be synchronized, but only up to an unknown additive error lying somewhere in the range $[-r, r]$. This ambiguity cannot be circumvented, even in principle, by any algorithm. We explore this further under Experimental Methodology below.

2.5 Synchronization Algorithms

The *ntpd* daemon [11] is the standard clock synchronization algorithm used today. It is a *feedback* based design, in particular since system clock timestamps are used to timestamp the timing packets. The existing kernel system clock, which provides the interface for user and kernel timestamping and which is *ntpd*-oriented, is disciplined by *ntpd*. The final software clock is therefore quite a complex system as the system clock has its own dynamics, which interacts with that of *ntpd* via feedback. On Xen, *ntpd* relies on the Xen Clocksource as its underlying counter.

The *RADclock* [20] (Robust Absolute and Difference Clock) is a recently proposed alternative clock synchronization algorithm based on a *feed-forward* design. Here timing packets are timestamped using raw packet timestamps. The clock error is then estimated based on these and the server timestamps, and subtracted out when the clock is read. This is a feed-forward approach, since errors are corrected based on post-processing outputs, and these are not themselves fed back into the next round of inputs. In other words, the raw timestamps are independent of clock state. The 'system clock' is now stateless, simply returning a function of parameters maintained by the algorithm.

More concretely, the (absolute) *RADclock* is defined as $C_a(t) = N(t) \cdot \bar{p} + K - E(t)$, where $N(t)$ is the raw timestamp made at true time t, $\bar{p}$ is a stable estimate of average counter period, K is a constant which aligns the origin to the required timescale (such as UTC), and $E(t)$ is the current estimate of the error of the 'uncorrected clock' $N(t) \cdot \bar{p} + K$ which is removed when the clock is read. The parameters $\bar{p}$, K and E are maintained by the clock algorithm (see [20] for details of the algorithm itself). The clock reading function simply reads (or is passed) the raw timestamp N for the event of interest, fetches the clock parameters, and returns $C_a(t)$.

The *RADclock* can use any counter which satisfies basic requirements, namely that it be cumulative, does not roll over, and has reasonable stability. In this paper we provide results using both the HPET and Xen Clocksource.

2.6 Experimental Methodology

We give a brief description of the main elements of our methodology for evaluating clock performance. More details can be found in [15].

The basic setup is shown in Figure 1. It incorporates our own Stratum-1 NTP server on the LAN as the reference clock, synchronised via a GPS-corrected atomic clock. NTP timing packets flow between the server and the clocks in the host machine (two OS's each with two

clocks are shown in the figure) for synchronization purposes, typically with a poll period of 16 seconds. For evaluation purposes, a separate box sends and receives a flow of UDP packets (with period 2 seconds) to the host, acting as a set of timestamping 'opportunities' for the clocks under test. In this paper a separate stream was sent to each OS in the host, but for a given OS, all clocks timestamp the same UDP flow.

Based on timestamps of the arrival and departure of the UDP packets, the testbed allows two kinds of comparisons.

External: a specialized packet stamping 'DAG' card [4] timestamps packets just before they enter the NIC of the host machine. These can be compared to the timestamps for the same packets taken by the clocks inside the host. The advantage is an independent assessment; the disadvantage is that there is 'system' lying between the two timestamping events, which adds a 'system noise' to the error measurement.

Internal: clocks inside the same OS timestamp the packets back-to-back (thanks to our kernel modifications), so subtracting these allows the clocks to be compared. The advantage is the elimination of the system noise between the timestamps; the disadvantage is that differences between the clocks cannot be attributed to any specific clock.

The results appearing in this paper all use the external comparison, but internal comparisons were also used as a key tool in the process of investigation and validation.

As far as possible experiments are run concurrently so that clocks to be compared experience close to identical conditions. For example, clocks in the same OS share the very same NTP packets to the time server (and hence in particular, share the same poll period). There are a number of subtle issues we have addressed regarding the equivalence between what the test UDP packets, and the NTP packets actually used by the algorithm, 'see', which depends on details of the relative timestamping locations in the kernel. This topic is discussed further below in relation to 'host asymmetry'.

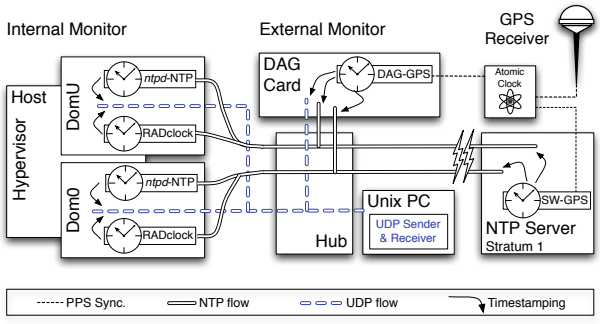

Figure 1: Testbed and clock comparison methodology.

It is essential to note that the delays experienced by timing packets have components both in the network, and in the host itself (namely the NIC + hardware + OS), each with their own minimum RTT and asymmetry values. Whereas the DAG card timestamps enable the network side to be independently measured and corrected, the same is not true of the host side component. Even when using the same server then, a comparison of different clocks, which have different asymmetry induced errors, is problematic. Although the spread of errors can be meaningfully compared, the median errors can only be compared up to some limit imposed by the (good but not perfect) methodology, which is of the order of 1 to 10 µs. Despite these limitations, we believe our methodology to be the best available at this time.

3 Inadequacy of the Status-Quo

The current timekeeping solution for Xen is built on top of the *ntpd* daemon. The single most important thing then regarding the performance of Xen timekeeping is to understand the behavior first of *ntpd* in general, and then in the virtualized environment.

There is no doubt that *ntpd* can perform well under the right conditions. If a good quality nearby time server is available, and if *ntpd* is well configured, then its performance on modern kernels is typically in the tens of microseconds range and can rival that of the *RADclock*. An example in the Xen context is provided in Figure 2, where the server is a Stratum-1 on the same LAN, and both *RADclock* and *ntpd*, running on Dom0 in parallel, are synchronizing to it using the same stream of NTP packets. Here we use the host machine **kultarr**, a 2.13GHz Intel Core 2 Duo. Xen selects a single CPU for use with Xen Clocksource, which is then used by *ntpd*. Power management is disabled in the BIOS.

The errors show a similar spread, with an Inter-Quartile Range (IQR) of around 10 µs for each. Note that here path asymmetry effects have not been accounted for, so that as discussed above the median errors do not reflect the exact median error for either clock.

In this paper our focus is on the right architecture for timing in virtualized systems, in particular such that seamless VM migration becomes simple and reliable, and not any performance appraisal of *ntpd* per se. Accordingly, unless stated otherwise we consistently adopt the configuration which maximizes *ntpd* performance – single nearby statically allocated Stratum-1 server, static and small polling period.

The problem with *ntpd* is the sudden performance degradations which can occur when conditions deviate from 'ideal'. We have detailed these robustness issues of *ntpd* in prior work, including [17, 18]. Simply put, when path delay variability exceeds some threshold, which is a

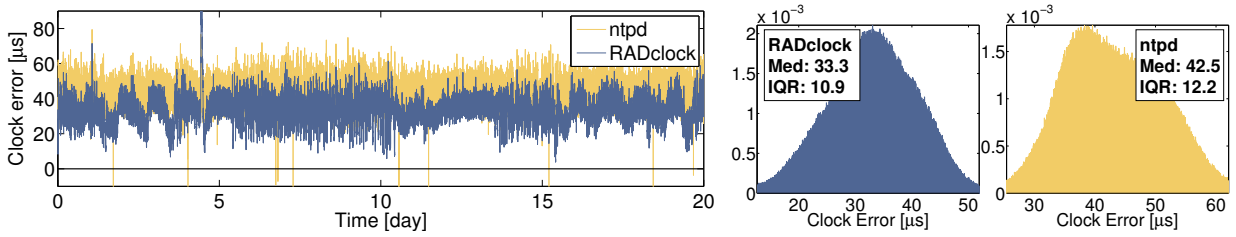

Figure 2: *RADclock* and *ntpd* uncorrected performance on dom0, measured using the external comparison with DAG.

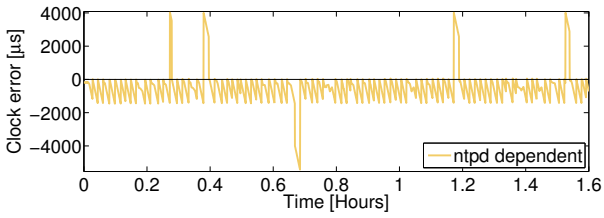

Figure 3: Error in the DomU dependent clock (dependent on the *ntpd* on Dom0 shown in Figure 2), measured using the external comparison with DAG. This 2 hour zoom is representative of an experiment 20 days long.

complex function of parameters, stability of the feedback control is lost, resulting in errors which can be large over small to very long periods. Recovery from such periods is also subject to long convergence times.

Consider now the pitfalls of using *ntpd* for timekeeping in Xen, through the following three scenarios.

Example 1 – Dependent *ntpd* clock In a *dependent* clock paradigm, only Dom0 runs a full clock synchronization algorithm, in this case *ntpd*. Here we use a 2.6.26 kernel, the last one supporting *ntpd* dependent timekeeping.

In the solution detailed in [19], synchronizing timestamps from *ntpd* are communicated to DomU guests via the periodic adjustment of a 'boot time' variable in the hypervisor. Timestamping in DomU is achieved by a modified system clock call which uses Xen Clocksource to extrapolate from the last time this variable was updated forward to the current time. The extrapolation assumes Xen Clocksource to be exactly 1GHz, resulting in a sawtooth shaped clock error which, in the example from our testbed given in Figure 3, is in the millisecond range. This is despite the fact that the Dom0 clock it derives from is the one depicted in Figure 2, which has excellent performance in the 10 µs range.

These large errors are ultimately a result of the way in which *ntpd* interacts with the system clock. With no *ntpd* running on DomU (the whole point of the dependent clock approach), the system clock has no way of intelligently correcting the drift of the underlying counter, in this case Xen Clocksource. The fact that Xen Clocksource is in reality only very approximately 1GHz means that this drift is rapid, indeed appearing to first order as a simple skew, that is a constant error in frequency. This failure of the dependent clock approach using *ntpd* has led to the alternative solution used today, where each guest runs its own independent *ntpd* daemon.

Example 2 – Independent *ntpd* clock In an *independent* clock paradigm, which is used currently in Xen timekeeping, each guest OS (both Dom0 and DomU) independently runs its own synchronization algorithm, in this case *ntpd*, which connects to its own server using its own flow of NTP timing packets. Clearly this solution is not ideal in terms of the frugal use of server, network, NIC and host resources. In terms of performance, the underlying issue is that the additional latencies suffered by guests in the virtual context make it more likely *ntpd* will be pushed into instability. Important examples of such latencies are the descheduling and time-multiplexing of guests across physical cores.

An example is given in Figure 4, where, despite synchronizing to the same high quality server on the LAN as before, stability is lost and errors reach the multiple millisecond range. This was brought about simply by adding a moderate amount of system load (some light churn of DomU guests and some moderate CPU activity on other guests), and allowing NTP to select its own polling period (in fact the default configuration), rather than fixing it to a constant value.

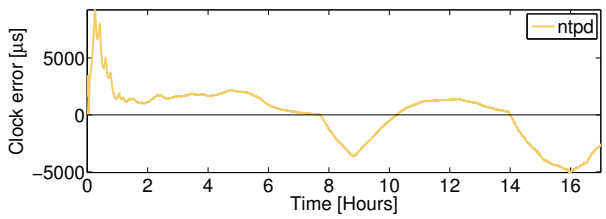

Figure 4: Error in the *ntpd* independent clock on DomU synchronizing to a Stratum-1 server on the LAN, with polling period set by *ntpd*. Additional guests are created and destroyed over time.

Example 3 – Migrating independent *ntpd* clock This example considers the impact of migration on the synchronization of the *ntpd* independent clock (the current solution) of a migrating guest. Since migration is treated in detail in Section 6, we restrict ourselves here to pointing to Figure 11, where extreme disruption – of the order of seconds – is seen following migration events. This is not a function of this particular example but is a generic result of the design of *ntpd* in conjunction with the independent clock paradigm. A dependent *ntpd* clock solution would not exhibit such behavior under migration, however it suffers from other problems as detailed above.

In summary, there are compelling design and robustness reasons for why *ntpd* is ill suited to timekeeping in virtual environments. The *RADclock* solution does not suffer from any of the drawbacks detailed above. It is highly robust to disruptions in general as described for example in [20, 15, 16], is naturally suited to a dependent clock paradigm as detailed in Section 5, as well as to migration (Section 6).

4 Performance of Xen Clocksource

The Xen Clocksource hybrid counter is a central component of the current timing solution under Xen. In this section we examine its access latency under different conditions. We also compare it to that of HPET, both because HPET is a core component of Xen Clocksource, so this enables us to better understand how the latter is performing, and because HPET is a good choice of counter, being widely available and uninfluenced by power management. This section also provides the detailed background necessary for subsequent discussions on network noise.

Access latency, which impacts directly on timekeeping, depends on the access mechanism. Since the timing architecture we propose in Section 5 is based on a feed-forward paradigm, to be of relevance our latency measurements must be of access mechanisms that are adequate to support feed-forward based synchronization. The fundamental requirement is that a counter be defined, which is cumulative, wide enough to not wrap between reboots (we use 64-bit counters which take 585 years to roll over on a 1GHz processor), and accessible from both kernel and user context. The existing *ntpd*-oriented software clock mechanisms do not satisfy these conditions. We describe below the alternatives we implement.

4.1 Baseline Latencies

In this section we use the host machine **kultarr**, a 2.13GHz Intel Core 2 Duo, and measure access latencies by counting the number of elapsed CPU cycles. For this purpose we use the *rdtsc()* function, a wrapper for the x86 RDTSC instruction to read the relevant register(s) containing the TSC value and to return it as a 64-bit integer. This provides direct access to the TSC with very low overhead from both user and kernel space. To ensure a unique and reliable TSC, from the BIOS we disable power management (both P-states and C-states), and also disable the second core to avoid any potential failure of TSC synchronization across the cores.

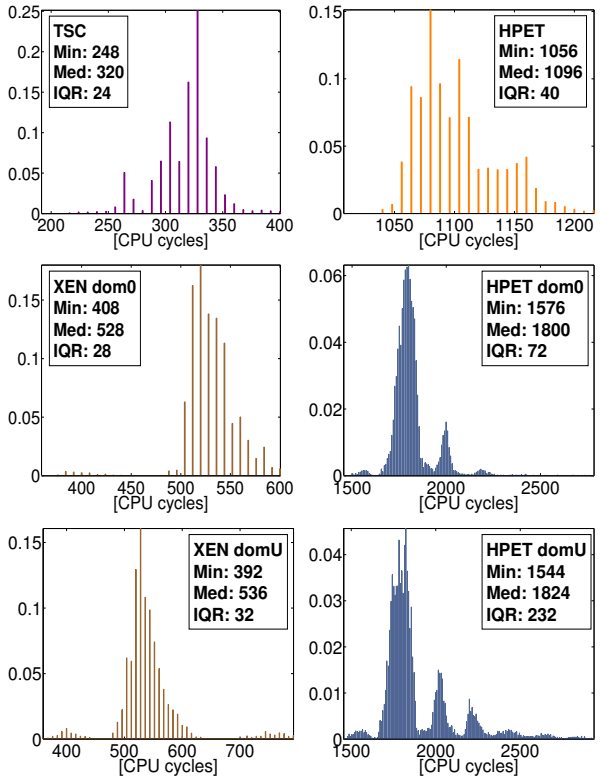

Figure 5: Distribution of counter latencies. Top: TSC and HPET in an unvirtualized system using feed-forward compatible access; Middle: Xen Clocksource and HPET from Dom0; Bottom: from DomU.

We begin by providing a benchmark result for HPET on a non-virtualized system. The top right plot in Figure 5 gives its latency histogram measured from within kernel context, using the access mechanism described in [3]. This mechanism augments the Linux *clocksource* code (which supports a choice among available hardware counters), to expose a 64-bit cumulative version of the selected counter. For comparison, in the top left plot we give the latency of the TSC accessed in the same way – it is much smaller as expected, but both counters have low variability. Note (see [3]) that the latency of TSC accessed directly via *rdtsc()* is only around 80 cycles, so that the feed-forward friendly access mechanism entails an overhead of around 240 cycles on this system.

Now consider latency in a Xen system. To provide the feed-forward compatible access required for each of HPET and Xen Clocksource, we first modified the Xen Hypervisor (4.0) and the Linux kernel 2.6.31.13 (Xen pvops branch) to expose the HPET to Dom0 and DomU. We added a new hypercall entry that retrieves the current raw platform timer value, HPET in this case. Like Xen Clocksource, this is a 64-bit counter which satisfies the cumulative requirement. We then added a system call to enable each counter to be accessed from user context. Finally, for our purposes here we added an additional system call that measures the latency of either HPET or Xen Clocksource from within kernel context using *rdtsc()*.

The middle row in Figure 5 shows the latency of Xen Clocksource and HPET from Dom0's kernel point of view. The Xen Clocksource interpolation mechanism adds an extra 200 CPU cycles compared to accessing the TSC alone in the manner seen above, for a total of 250 ns at this CPU frequency. The HPET latency suffers from the penalty created by the hypercall needed to access it, lifting its median value by 800 cycles for a total latency of 740 ns. More importantly, we see that the hypercall also adds more variability, with an IQR that increases from 40 to 72 CPU cycles, and the appearance of a multi-modal distribution which we speculate arises from some form of contention among hypercalls. The Xen Clocksource in comparison has an IQR only slightly larger than that of the TSC counter.

The bottom row in Figure 5 shows the latency of Xen Clocksource and HPET from DomU's kernel point of view. The Xen Clocksource shows the same performance as in the Dom0 case. HPET is affected more, with an increase in the number of modes and the mass within them, resulting in a considerably increased IQR.

In conclusion, Xen Clocksource performs well despite the overhead of its software interpolation scheme. In particular, although its latency is almost double that of a simple TSC access (and 7 times a native TSC access), it does not add a significant latency variability even when accessed from DomU. On the other hand however, the simple feed-forward compatible way of accessing HPET used here is only four times slower than the much more complicated Xen Clocksource and is still under 1 µs. This performance could certainly be improved, for example by replacing the hypercall by a dedicated mechanism such as a read-only memory-mapped interface.

4.2 Impact of Power Management

Power management is one of the key mechanisms potentially affecting timekeeping. The Xen Clocksource is designed to compensate for its effects in some respects. Here we examine its ultimate success in terms of latency.

In this section we use the host machine **sarigue**, a 3GHz Intel Core 2 Duo E8400. Since we do not use *rdtsc()* for latency measurement in this section, we do not disable the second core as we did before. Instead we measure time differences in seconds, to sub-nanosecond precision, using the *RADclock* difference clock [20] with HPET as the underlying counter[2]. P-states are disabled in the BIOS, but C-states are enabled.

Ideally one would like to directly measure Xen Clocksource's interpolation mechanism and so evaluate it in detail. However, the Xen Clocksource recalibrates the HPET interpolation on every change in P-State (frequency changes) or C-State (execution interruption), as well as once per second. Since oscillations for example between C-States occur hundreds of times per second [8], it is not possible to reliably timestamp these events, forcing us to look at coarser characterizations of performance.

From the timestamping perspective, the main problem is the obvious additional latency due to the computer being idle in a C-State when an event to timestamp occurs. For example, returning from C-State C3 to execution C0 takes about 20 µs [8].

For the purpose of synchronization over the network, the timestamping of outgoing and incoming synchronization packets is of particular interest. A useful measure of these is the Round-Trip-Time (RTT) of a request-reply exchange, however since this includes network queuing and delays at the time server as well as delays in the host, it is of limited use in isolating the latter.

To observe the host latency we introduce a metric we call the *RTThost*, which is roughly speaking the component of the RTT that lies within the host. More precisely, for a given request-reply packet pair, the RTThost is the sum of the two one-way delays from the host to the DAG card, and from the DAG card to the host. It is not possible to reliably measure these one-way delays individually in our testbed. However, the RTThost can be reliably measured as the difference of the RTT seen by the host and that seen by the DAG card. The RTThost is a measure of 'system noise' with a specific focus on packet timestamping. The smaller RTThost, the less noisy the host, and the higher the quality of packet timestamps.

Figure 6 shows RTThost values measured over 80 hours on two DomU guests on the same host. The capture starts with only the C-State C0 enabled, that is with power management functions disabled. Over the period of the capture, deeper C-States are progressively enabled and we observe the impact on RTThost. At each stage the CPU moves between the active state C0 and the idle states enabled at the time. Table 1 gives a breakdown of

[2]This level of precision relates to the difference clock itself, when measuring time differences of size of the order of 100 µs as here. It does not take into account the separate issue of timestamping errors, such as the (much larger!) counter access latencies studied above.

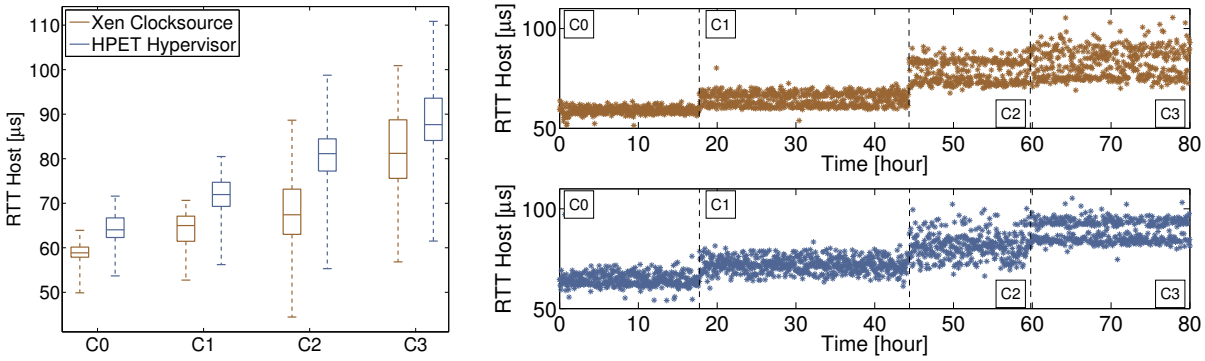

Figure 6: System noise as a function of the deepest enabled C-State for Xen Clocksource (upper time series and left box plots) and HPET (lower time series and right box plots). Time series plots have been sampled for clarity.

time spend in different states. It shows that typically the CPU will rest in the deepest allowed C-state unless there is a task to perform.

The left plot in Figure 6 is a compact representation of the distribution of RTThost values for Xen Clocksource and HPET, for each section of the corresponding time series presented on the right of the figure. Here whiskers show the minimum and 99th percentile values, the lower and upper sides of the box give the 25th and 75th percentiles, while the internal horizontal line marks the median.

The main observation is that, for each counter, RTThost generally increases with the number of C-States enabled, although it is slightly higher for HPET. The increase in median RTThost from C0 to C3 is about 20 µs, a value consistent with [8]. The minimum value is largely unaffected however, consistent with the fact that if a packet (which of course is sent when the host is in C0), is also received when it is in C0, then it would see the RTThost corresponding to C0, even if it went idle in between.

We saw earlier that the access latencies of HPET and Xen Clocksource differ by less than 1 µs, and so this cannot explain the differences in their RTThost median values seen here for each given C-State. These are in fact due to the slightly different packet processing in the two DomU systems.

	C0	C1	C2	C3
C0 enabled	100%	–	–	–
C1 enabled	2.17%	97.83%	–	–
C2 enabled	2.85%	0.00%	97.15%	–
C3 enabled	2.45%	0.00%	1.84%	95.71%

Table 1: Residency time in different C-States. Here "Cn enabled" denotes that all states from C0 up to Cn are enabled.

We conclude that Xen Clocksource, and HPET using our proof of concept access mechanism, are affected by power management when it comes to details of timestamping latency. These translate into timestamping errors, which will impact both clock reading and potentially clock synchronization itself. The final size of such errors however is also crucially dependent on the asymmetry value associated to RTThost, which is unknown. Thus the RTTHost measurements effectively place a bound on the system noise affecting timestamping, but do not determine it.

5 New Architecture for Virtualized Clocks

In this section we examine the performance and detail the benefits of the *RADclock* algorithm in the Xen environment, describe important packet timestamping issues which directly impact clock performance, and finally propose a new feed-forward based clock architecture for para-virtualized systems.

In Section 5.1 we use **sarigue**, and in Section 5.2 **kultarr**, with the same BIOS and power management settings described earlier.

5.1 Independent *RADclock* Performance

We begin with a look at the performance of the *RADclock* in a Xen environment. Figure 7 shows the final error of two independent *RADclock*s, one using HPET and the other Xen Clocksource, running concurrently in two different DomU guests. Separate NTP packet streams are used to the same Stratum-1 server on the LAN with a poll period of 16 seconds. The clock error for each clock has been corrected for path asymmetry, in order to reveal the underlying performance of the algorithm as a delay variability filter (this is possible in our testbed, but impossible for the clocks in normal operation). The difference of median values between the two clocks is

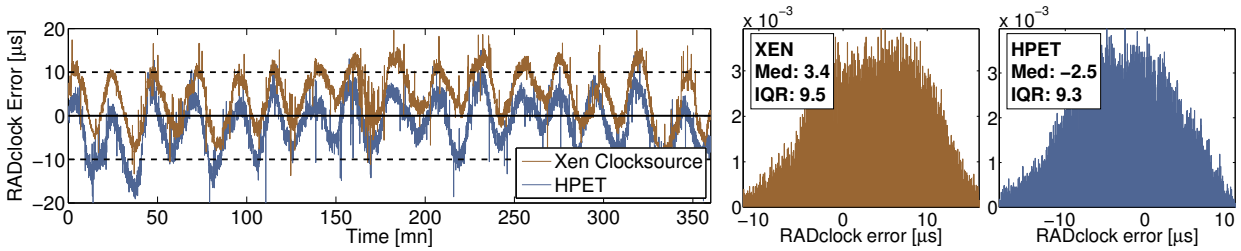

Figure 7: *RADclock* performance in state C0 using Xen Clocksource and HPET, running in parallel, each in a separate DomU guest.

extremely small, and below the detection level of our methodology. We conclude that the clocks essentially have identical median performance.

In terms of clock stability, as measured by the IQR of the clock errors, the two *RADclock* instances are again extremely similar, which reinforces our earlier observations that the difference in stability of the Xen Clocksource and HPET is very small (below the level of detection in our testbed), and that *RADclock* works well with any appropriate counter. The low frequency oscillation present in the time series here is due to the periodic cycle of the air conditioning system in the machine room, and affects both clocks in a similar manner consistent with previous results [3]. It is clearly responsible for the bulk of the *RADclock* error in this and other experiments shown in this paper.

Power management is also an important factor that may impact performance. Figure 8 shows the distribution of clock errors of the *RADclock*, again using HPET and the Xen Clocksource separately but concurrently as above, with different C-State levels enabled. In this case the median of each distribution has simply been shifted to zero to ease the stability (IQR) comparison. For each of the C-State levels shown, the stability of the *RADclock* is essentially unaffected by the choice of counter.

As shown in Figure 6, power management creates additional delays of higher variability when timestamping timing packets exchanged with the reference clock. The near indifference of the IQR given in Figure 8 to C-State shows that the *RADclock* filtering is robust enough to see through this extra noise.

Power management also has an impact on the asymmetry error all synchronization algorithms must face. In an excellent example of systematic observation bias, in a bidirectional paradigm a packet send by an OS would not be delayed by the power management strategy, because the OS chooses to enter an idle state only when it has nothing to do. On the other hand, over the time interval defined by the RTT of a time request, it is likely the host will choose to stop its execution and enter an idle state (perhaps a great many times) and the returning packet may find the system in such a state. Consequently, only the timestamping of received packets is likely to be affected by power management, which translates into a bias towards an extra path asymmetry, in the sense of 'most but not all packets', in the receiving direction. This bias is difficult to measure independently and authoritatively. The measurement of the RTThost shown in Figure 6 gives however a direct estimate of an upper bound for it.

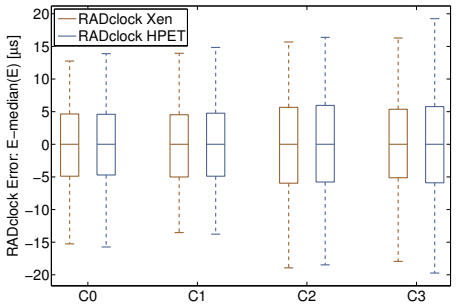

Figure 8: Compact centred distributions of *RADclock* performance as a function of the deepest C-State enabled (whiskers give 1st to 99th percentile).

5.2 Sharing the Network Card

The quality of the timestamping of network packets is crucial to the accuracy the synchronization algorithm can achieve. The networking in Xen relies on a firewall and networking bridge managed by Dom0. In Figure 9 we observe the impact of system load on the performance of this mechanism.

The top plot shows the RTThost time series, as seen from the Dom0 perspective, as we add more DomU guests to the host. Starting with Dom0 only, we add an additional guest every 12 hours. None of the OSs run any CPU or networking intensive tasks. The middle plot gives the box plots of the time series above, where the increase in median and IQR values is more clearly seen. For reference the 'native' RTThost of a non-virtualized system is also plotted. The jump from this distribution

to the one labeled 'Dom0' represents the cost of the networking bridge implemented in Xen.

The last plot in figure 9 shows the distribution of RTThost values from each guest's perspective. All guests have much worse performance than Dom0, but performance degrades by a similar amount as Dom0 as a function of the number of guests. For a given guest load level, the performance of each guest clock seems essentially the same, though with small systematic differences which may point to scheduling policies.

The observations above call for the design of a timestamping system under a dependent clock paradigm where Dom0 has an even higher priority in terms of networking, so that it can optimize its timestamping quality and thereby minimize the error in the central Dom0 clock, to the benefit of all clocks on the system. Further, DomU packet timestamping should be designed to minimize any differences between DomU guests, and reduce as much as possible the difference in host asymmetry between Dom0 and DomU guests, to help make the timestamping performance across the whole system more uniform.

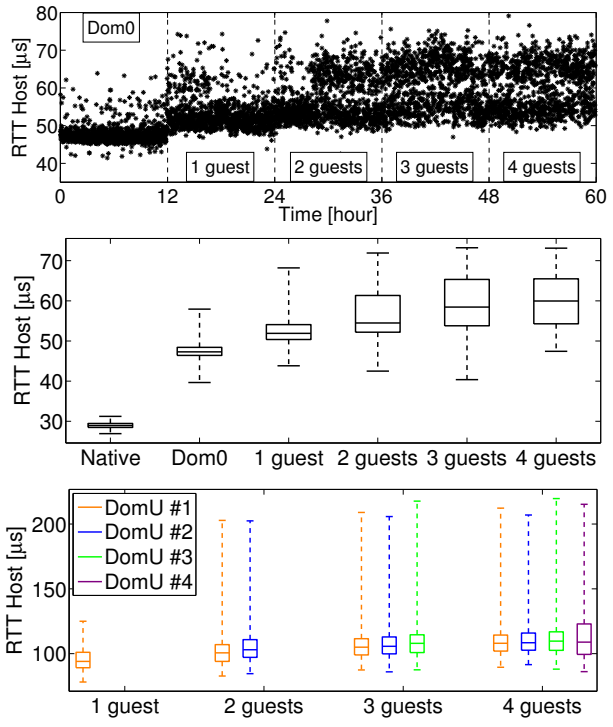

Figure 9: **kultarr**: RTThost (a.k.a. system noise) as a function of the number of active guests. Top: RTThost timeseries seen by Dom0; Middle: corresponding distribution summaries (with native non-Xen case added on the left for comparison); Bottom: as seen by each DomU. Whiskers show the minimum and 99th percentile.

5.3 A Feed-Forward Architecture

As described in Section 2.5, the feed-forward approach used by the *RADclock* has the advantage of cleanly separating timestamping (performed as a raw timestamp in the kernel or user space as needed), which is stateless, and the clock synchronization algorithm itself, which operates asynchronously in user space. The algorithm updates clock parameters and makes them available through the OS, where any authorized clock reading function (a kind of almost trivial stateless 'system clock') can pick them up and use them either to compose an absolute timestamp, or robustly calculate a time difference [20].

The *RADclock* is then naturally suited for the dependent clock paradigm and can be implemented in Xen as a simple read/write stateless operation using the *XenStore*, a file system that can be used as an inter-OS communication channel. After processing synchronization information received from its time server, the *RADclock* running on Dom0 writes its new clock parameters to the XenStore. On DomU, a process reads the updated clock parameters upon request and serves them to any application that needs to timestamp events. The application timestamps the event(s) of interest. These raw timestamps can then be easily converted either into a wallclock time or a time difference measured in seconds (this can even be done later off-line).

Unlike with *ntpd* and its coupled relationship to the (non-trivial) incumbent system clock code, no adjustment is passed to another dynamic mechanism, which ensures that only a single clock, clearly defined in a single module, provides universal time across Dom0 and all DomU guests.

With the above architecture, there is only one way in which a guest clock can not be strictly identical with the central Dom0 clock. The read/write operation on the XenStore is not instantaneous and it is possible that the update of clock parameters, which is slightly delayed after the processing of a new synchronization input to the *RADclock*, will result in different parameters being used to timestamp some event. In other words, the time across OSs may appear different for a short time if a timestamping function in a DomU converts a raw timestamp with outdated data. However, this is a minor issue since clock parameters change slowly, and using out of date values has the same impact as the synchronization input simply being lost, to which the clock is already robust.

In Figure 10 we measured the time required to write to the XenStore using the *RADclock* difference clock which has an accuracy well below 1 µs [20]. We present results obtained on 2 host machines with slightly different hardware architectures, namely **kultarr** (2.13 GHz Intel Core 2 Duo) and **tastiger** (3.40 GHz Intel Pentium D), that

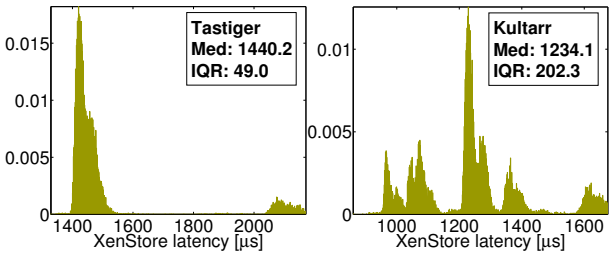

Figure 10: Distribution of clock update latency through the xenstore, **tastiger** (left, Pentium D, 3.4GHz) and **kultarr** (right, Core 2 Duo, 2.13GHz).

show respective median delays of 1.2 and 1.4 ms. Assuming a 16 s poll period, this corresponds to 1 chance out of 11,500 that the clocks would (potentially) disagree if read at some random time.

The dependent *RADclock* is ideally suited for time keeping on Xen DomU. It is a simple, stateless, standard read/write operation that is robust as it avoids the dangerous dynamics of feedback approaches, ensures that the clocks of all guests agree, and is robust to system load and power management effects. As a dependent clock solution, it saves both host and network resources and is inherently scalable. Thanks to a simple timestamping function it provides the same level of final timekeeping accuracy to all OSs.

6 A Migration-Friendly Architecture

Seamlessly migrating a running system from one physical machine to another is a key innovation of virtualization [13, 7]. However this operation becomes far from seamless with respect to timing when using *ntpd*. As mentioned in Section 3, *ntpd*'s design requires each DomU to run its own instance of the *ntpd* daemon, which is fundamentally unsuited to migration, as we now explain.

The synchronization algorithm embodied in the *ntpd* daemon is stateful. In particular it maintains a time varying estimate of the Xen Clocksource's rate-of-drift and current clock error, which in turn is defined by the characteristics of the oscillator driving the platform counter. After migration, the characteristics seen by *ntpd* change dramatically since no two oscillators drift in the same way. Although the Xen Clocksource counters on each machine nominally share the same frequency (1GHz), in practice this is only true very approximately. The temperature environment of the machine DomU migrates to can be very different from the previous one which can have a large impact, but even worse, the platform timer may be of a different nature, HPET originally and ACPI after migration for example. Furthermore, *ntpd* will also inevitably suffer from an inability to account for the time during which DomU has been halted during the migration. When DomU restarts, the reference wallclock time and last Xen Clocksource value maintained by its system clock will be quite inconsistent with the new ones, leading to extreme oscillator rate estimates. In summary, the sudden change in status of *ntpd*'s state information, from valid to almost arbitrary, will, at best, deliver a huge error immediately after migration, which we expect to decay only slowly according to *ntpd*'s usual slow convergence. At worst, the 'shock' of migration may push *ntpd* into an unstable regime from which it may never recover.

In contrast, by decomposing the time information into raw timestamps and clock parameters, as described in Section 5, the *RADclock* allows the daemon running on DomU to be stateless within an efficient dependent clock strategy. The migration then becomes trivial from a timekeeping point of view. Once migrated, DomU timestamps events of interests with its chosen counter and retrieves the *RADclock* clock parameters maintained by the new Dom0 to convert them into absolute time. DomU immediately benefits from the accuracy of the dedicated *RADclock* running on Dom0 – the convergence time is effectively zero.

The plots in Figure 11 confirm the claims above and illustrate a number of important points. In this experiment, each of **tastiger** and **kultarr** run an independent *RADclock* in Dom0. The clock error for these is remarkably similar, with an IQR below 10 µs as seen in the top plot (measured using the DAG external comparison). Here for clarity the error time series for the two Dom0 clocks have been corrected for asymmetry error, thereby allowing their almost zero inherent median error, and almost identical behavior (the air-conditioning generated oscillations overlay almost perfectly), to be clearly seen.

For the migration experiment, a single DomU OS is started on **tastiger**, and two clocks launched on it: a dependent *RADclock*, and an independent *ntpd* clock. A few hours of warm up are then given (not shown) to allow *ntpd* to fully converge. The experiment proper then begins. At the 30 minute mark DomU is migrated to **kultarr**, it migrates back to **tastiger** after 2 hours then back again after another 2, followed by further migrations with a smaller period of 30 minutes.

The resulting errors of the two migrating DomU clocks are shown in the top plot, and in a zoomed out version in the middle plot, as measured using the external comparison. Before the results, a methodological point. The dependent *RADclock* running on DomU is by construction identical to the *RADclock* running on Dom0, and so the two time series (if asymmetry corrected) would superimpose almost perfectly, with small differences owing to the different errors in the times-

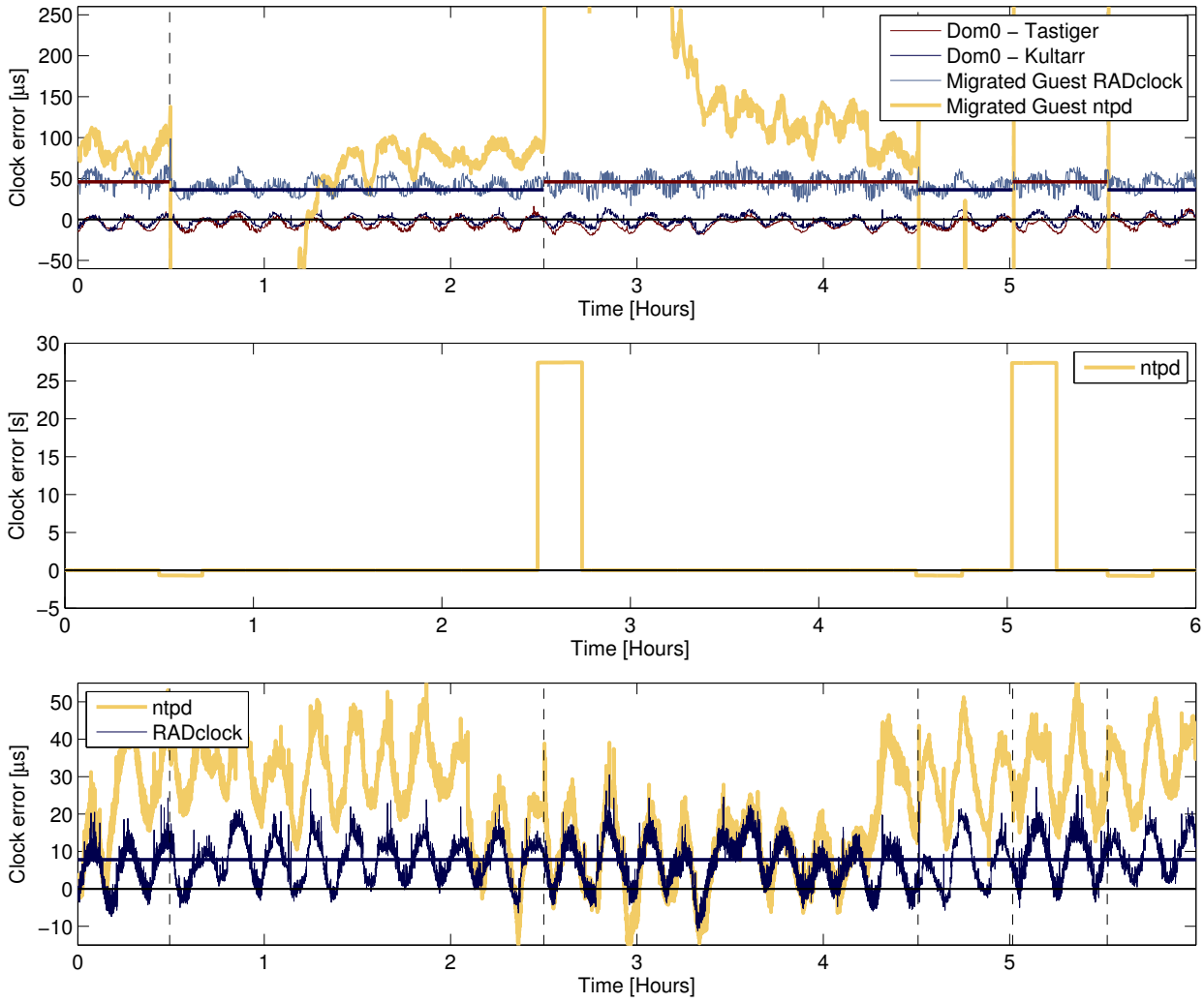

Figure 11: Clock errors under migration. Top: asymmetry corrected unmigrated *RADclock* Dom0 clocks, and (uncorrected) migrated clocks on DomU; Middle: zoom out on top plot revealing the huge size of the migration 'shock' on *ntpd*; Bottom: effect of migration load on Dom0 clocks on **kultarr**.

tamping of the separate UDP packet streams. We choose however, in the interests of fairness and simplicity of comparison, not to apply the asymmetry correction in this case, since it is not possible to apply an analogous correction to the *ntpd* error time series. As a substitute, we instead draw horizontal lines over the migrating *RADclock* time series representing the correction which *would* have been applied. No such lines can be drawn in the *ntpd* case.

Now to the results. As expected, and from the very first migration, *ntpd* exhibits extremely large errors (from -1 to 27 s!) for periods exceeding 15 minutes (see zoom in middle plot) and needs at least another hour to converge to a reasonable error level. The dependent *RADclock* on the other hand shows seamless performance with respect to the horizontal lines representing the expected jumps due to asymmetry changes as just described. These jumps are in any case small, of the order of a few microseconds. Note that these corrections are a function both of RTThost and asymmetry that are both different between **tastiger** and **kultarr**.

Finally, we present a load test comparison. The bottom plot in Figure 11 compares in detail the performance of the independent *RADclock* running on Dom0 on **kultarr**, and an independent *ntpd* clock, also running on Dom0 during the experiment (not shown previously). Whereas the *RADclock* is barely affected by the changes in network traffic and system load associated with the migrations of the DomU guest, *ntpd* shows significant deviation. In summary, not only is *ntpd* in an independent

clock paradigm incompatible with clock migration, it is also, regardless of paradigm, affected by migration occurring around it.

One could also consider the performance of an independent *RADclock* paradigm under migration. However, we expect that the associated 'migration shock' would be severe as the *RADclock* is not designed to accommodate radical changes in the underlying counter. Since the dependent solution is clearly superior from this and many other points of view, we do not present results for the independent case under migration.

7 Conclusion

Virtualization of operating systems and accurate computer based timing are two areas set to increase in importance in the future. Using Xen para-virtualization as a concrete framework, we highlighted the weaknesses of the existing timing solution, which uses independent *ntpd* synchronization algorithms (coupled to stateful software clock code) for each guest operating system. In particular, we showed that this solution is fundamentally unsuitable for the important problem of live VM migration, using both arguments founded on the design of *ntpd*, as well as detailed experiments in a hardware-validated testbed.

We reviewed the architecture of the *RADclock* algorithm, in particular its underlying feed-forward basis, the clean separation between its timestamping and synchronization aspects, and its high robustness to network and system noise (latency variability). We argued that these features make it ideal as a dependent clock solution, particularly since the clock is already set up to be read through combining a raw hardware counter timestamp with clock parameters sourced from a central algorithm which owns all the synchronization intelligence, via a commonly accessible data structure. We supported our claims by detailed experiments and side-by-side comparisons with the status quo. For the same reasons, the *RADclock* approach enables seamless and simple migration, which we also demonstrated in benchmarked experiments. The enabling of a dependent clock approach entails considerable scalability advantages and suggests further improvements through optimizing the timestamping performance of the central clock in Dom0.

As part of an examination of timestamping and counter suitability for timekeeping in general and the feed-forward paradigm in particular, we provided a detailed evaluation of the latency and accuracy of the Xen Clocksource counter, and compared it to HPET. We concluded that it works well as intended, however note that it is a complex solution created to solve a problem which will soon disappear as reliable TSC counters again become ubiquitous. The *RADclock* is suitable for use with any counter satisfying basic properties, and we showed its performance using HPET or Xen Clocksource was indistinguishable.

The *RADclock* [14] packages for Linux now support a streamlined version of the architecture for Xen described here using Xen Clocksource as the hardware counter. With the special code allowing system instrumentation and HPET access removed, no modifications to the hypervisor are finally required.

8 Acknowledgments

The *RADclock* project is partially supported under Australian Research Council's Discovery Projects funding scheme (project number DP0985673) and a Google Research Award.

We thank the anonymous reviewers and our shepherd for their valuable feedback.

9 Availability

RADclock packages for Linux and FreeBSD, software and papers, can be found at http://www.cubinlab.ee.unimelb.edu.au/radclock/.

References

[1] Xen.org History. `http://www.xen.org/community/xenhistory.html`.

[2] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), ACM, pp. 164–177.

[3] BROOMHEAD, T., RIDOUX, J., AND VEITCH, D. Counter Availability and Characteristics for Feed-forward Based Synchronization. In *Int. IEEE Symp. Precision Clock Synchronization for Measurement, Control and Communication (ISPCS'09)* (Brescia, Italy, Oct. 12-16 2009), IEEE Piscataway, pp. 29–34.

[4] ENDACE. Endace Measurement Systems. DAG series PCI and PCI-X cards. http://www.endace.com/networkMCards.htm.

[5] INTEL CORPORATION. IA-PC HPET (High Precision Event Timers) Specification (revision 1.0a). `http://www.intel.com/hardwaredesign/hpetspec_1.pdf`, Oct. 2004.

[6] KAMP, P. H. Timecounters: Efficient and precise timekeeping in SMP kernels. In *Proceedings of the BSDCon Europe 2002* (Amsterdam, The Netherlands, 15-17 November 2002).

[7] KEIR, C. C., CLARK, C., FRASER, K., H, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live Migration of Virtual Machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2005), pp. 273–286.

[8] KIDD, T. Intel Software Network Blogs. `http://software.intel.com/en-us/blogs/author/taylor-kidd/`.

[9] MENON, A., SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G. J., AND ZWAENEPOEL, W. Diagnosing performance overheads in the xen virtual machine environment. In *VEE '05:*

Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments (New York, NY, USA, 2005), ACM, pp. 13–23.

[10] MICROSOFT CORPORATION. Guidelines For Providing Multimedia Timer Support. Tech. rep., Microsoft Corporation, Sep. 2002. `http://www.microsoft.com/whdc/system/sysinternals/mm-timer.mspx`.

[11] MILLS, D. L. *Computer Network Time Synchronization: The Network Time Protocol*. CRC Press, Inc., Boca Raton, FL, USA, 2006.

[12] MOGUL, J., MILLS, D., BRITTENSON, J., STONE, J., AND WINDL, U. Pulse-Per-Second API for UNIX-like Operating Systems, Version 1.0. Tech. rep., IETF, 2000.

[13] NELSON, M., HONG LIM, B., AND HUTCHINS, G. Fast transparent migration for virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (2005), USENIX Association.

[14] RIDOUX, J., AND VEITCH, D. RADclock Project webpage.

[15] RIDOUX, J., AND VEITCH, D. A Methodology for Clock Benchmarking. In *Tridentcom* (Orlando, FL, USA, May 21-23 2007), IEEE Comp. Soc.

[16] RIDOUX, J., AND VEITCH, D. The Cost of Variability. In *Int. IEEE Symp. Precision Clock Synchronization for Measurement, Control and Communication (ISPCS'08)* (Ann Arbor, Michigan, USA, Sep. 24-26 2008), pp. 29–32.

[17] RIDOUX, J., AND VEITCH, D. Ten Microseconds Over LAN, for Free (Extended). *IEEE Trans. Instrumentation and Measurement (TIM) 58*, 6 (June 2009), 1841–1848.

[18] RIDOUX, J., AND VEITCH, D. Principles of Robust Timing Over the Internet. *ACM Queue, Communications of the ACM 53*, 5 (May 2010), 54–61.

[19] THE XEN TEAM. Xen Documentation. `http://www.xen.org/files/xen_interface.pdf`.

[20] VEITCH, D., RIDOUX, J., AND KORADA, S. B. Robust Synchronization of Absolute and Difference Clocks over Networks. *IEEE/ACM Transactions on Networking 17*, 2 (April 2009), 417–430.

[21] VMWARE. Timekeeping in VMware Virtual Machines. Tech. rep., VMware, May 2010. `http://www.vmware.com/files/pdf/Timekeeping-In-VirtualMachines.pdf`.