Navigating the x86 TSC/kvmclock maze

Clocks on x86

TIME STAMP COUNTER (TSC)

Setting ancient history aside, time on x86 is derived from the Time Stamp Counter (TSC); a counter which is readable directly from the CPU as a Machine Specific Register (MSR).

There have been a number of issues with the TSC on historical x86 platforms, where it would either stop counting or change frequency according to the behaviour of the CPU itself, or where it *(and even its frequency)* might differ from one CPU to another. These issues should no longer exist on modern hardware, but KVM still has code to work around them and present a sane-ish environment to guests.

A number of CPUID or hypervisor 'features' exist to advertise the lack of certain problems:

```
#define X86_FEATURE_CONSTANT_TSC        ( 3*32+ 8) /* TSC ticks at a constant rate */
#define X86_FEATURE_TSC_RELIABLE        ( 3*32+23) /* TSC is known to be reliable */
#define X86_FEATURE_NONSTOP_TSC         ( 3*32+24) /* TSC does not stop in C states */
#define X86_FEATURE_NONSTOP_TSC_S3      ( 3*32+30) /* TSC doesn't stop in S3 state */
#define X86_FEATURE_TSC_KNOWN_FREQ      ( 3*32+31) /* TSC has known frequency */
```

The handling of the guest TSC is fairly simple. Modern CPUs allow the host to provide both an offset and a scaling factor to be applied by hardware while running in guest vCPU mode. Even where scaling isn't supported, KVM will allow a guest vCPU to run *faster* than the host, by artificially advancing the offset each time the vCPU is entered. During a given period of the vCPU running in guest mode, its TSC will run slower than it should, but will catch up after the next VMExit.

KVM CLOCK (AND XEN CLOCK)

Both KVM and Xen expose an identical clock to guests, indicating the time in nanoseconds which the guest has been running for. By means of a virtual MSR or hypercall, the guest asks the hypervisor to populate the following structure in guest memory:

```
struct pvclock_vcpu_time_info {
        u32   version;
        u32   pad0;
        u64   tsc_timestamp;
        u64   system_time;
        u32   tsc_to_system_mul;
        s8    tsc_shift;
        u8    flags;
        u8    pad[2];
} __attribute__((__packed__)); /* 32 bytes */
```

This struct is per-vCPU because, in theory, each vCPU's TSC could run at a different rate, and/or have a different offset. The values of `system_time` and `tsc_timestamp` represent the guest TSC value and the VM uptime in nanoseconds at an arbitrary point in time, while `tsc_to_system_mul` and `tsc_shift` provide a conversion from the guest TSC ticks to nanoseconds. The uptime, calculated by the __pvclock_read_cycles() function in Linux's `<asm/pvclock.h>`, is:

kvmclock_ns = `system_time` + (((`rdtsc()` - `tsc_timestamp`) * `tsc_to_system_mul`) >> `tsc_shift`)

In an ideal world where the TSC is reliable and synchronized across vCPUs, the contents of this structure should be identical for

all vCPUs and should never change. However, if the host's TSC does suffer discontinuities, KVM will modify the contents of the structure for each vCPU accordingly.

KVM Userspace Clock APIs

KVM_SET_TSC_KHZ

This ioctl allows userspace to set the TSC frequency for a given vCPU. When when invoked with the KVM file descriptor, sets the default TSC frequency for all subsequently created vCPUs. Modern CPUs support hardware TSC scaling, presenting an apparent TSC frequency to the guest which differs from the true host TSC frequency.

However, the precise effect will differ between AMD and Intel hardware, because the scaling uses a hard-coded shift; Intel CPUs multiply the host TSC by a given factor and then shift right by 48 bits, while AMD multiplies by the factor and then shifts by 32 bits. Without hardware support for TSC scaling, KVM artificially advances the TSC each time it enters the guest (and the TSC runs 'too slow' for the period while inside the guest).

KVM_SET_MSRS(MSR_IA32_TSC)

As an MSR, the TSC is directly *writable* by either the guest or the host. Writing to the TSC sets its value at the specific moment that the write takes effect, which is fundamentally imprecise. To compensate for this, KVM has a nasty hack. It keeps track of the last TSC value that was written for any vCPU, and the *(host CLOCK_MONOTONIC_RAW)* time at which it was written.
When *userspace* subsequently writes a TSC value on any vCPU which is within a second's worth of where that previously-written TSC would now be, KVM 'snaps' the result to be precisely the same, by using the same *offset* for host→guest translation.

KVM_SET_DEVICE_ATTR(KVM_VCPU_TSC_OFFSET)

This is a more useful way of setting the guest TSC for a given vCPU, as an *offset* from the host's TSC. With TSC scaling enabled, this is an offset from the *scaled* TSC (that is, this offset is applied *after* scaling).

For live update, this API allows for a cycle-accurate restoration of the guest TSC because the *host's* TSC will have continued to tick at precisely the same rate.

KVM_SET_CLOCK

This ioctl allows for two modes of setting the clock. The simplest invocation allows userspace to set the clock at the moment the ioctl takes effect. As with directly writing a TSC value, this is fundamentally imprecise.

A more subtly broken version also exists, using a `KVM_CLOCK_REALTIME` flag. In theory, this allows userspace to save and restore a pair of { real time, kvmclock } values at a given moment in time. However, in interpreting these values, KVM still introduces errors by performing two *separate* clock reads at different times, for the 'real time' and the monotonic time, and then assuming they were simultaneous. Furthermore, this API uses CLOCK_REALTIME (UTC) as its reference instead of CLOCK_TAI, leading to errors if migration occurs in the vicinity of a leap second.

Inaccuracies due to arithmetic precision

The KVM clock is bound to the host's CLOCK_MONOTONIC_RAW. This differs from CLOCK_MONOTONIC in that it is not subject to frequency adjustment from NTP. Therefore it should be a precise function of the host's TSC, just as the KVM clock is. So theoretically, if KVM were to periodically update the `tsc_timestamp` and `system_time` with values from a new read of CLOCK_MONOTONIC_RAW, it should have no effect on the result... right?

Sadly, that doesn't turn out to be true because the precise method of converting ticks to nanoseconds differs between CLOCK_MONOTONIC_RAW *(which has been refined for precision)*, and the KVM clock *(which is limited to what can be*

expressed in the ABI structure above). This means that there is a systemic *drift* between CLOCK_MONOTONIC_RAW and the KVM clock as observed by the guest. If the kernel ever recalculates the contents of the KVM clock structure, it can cause time to jump in the guest, and even to jump *backwards*. KVM used to perform such a recalculation on vCPU hotplug, but that was fixed in commit c52ffadc65e2 ("KVM: x86: Don't unnecessarily force masterclock update on vCPU hotplug").

Another source of systemic drift occurs between the `__get_kvmclock()` function and the time observed by the guest, because the kernel function converts directly from *host* TSC cycles while the guest necessarily uses a TSC value which is already scaled to the *guest* TSC frequency. This caused errors in the delivery of Xen timers until it was fixed (just for Xen timers) in commit 451a707813ae ("KVM: x86/xen: improve accuracy of Xen timers"). The `__get_kvmclock()` function has not yet been fixed for the general case.

Inaccuracies due to KVM_REQ_MASTERCLOCK_UPDATE

When generating the pvclock data structure, the `kvm_guest_time_update()` function uses a `kvmclock_offset` field in the KVM structure. It calculates the value of CLOCK_MONOTONIC_RAW at a given moment along with the host TSC at that same moment, then applies the `kvmclock_offset` to that, and fills in the structure accordingly.

On a system with a reliable TSC, the moment advertised to the guest as the reference point for the KVM clock should not change. It is stored in the `master_cycle_now` and `master_kernel_ns` fields of the KVM structure. Only in the pathological case does `kvm_guest_time_update()` actually use CLOCK_MONOTONIC_RAW (via `get_kvmclock_base_ns()` to calculate the values to expose to the guest.

Ultimately, this means that the KVM clock is bi-modal. In the sane case (when the `use_master_clock` flag is set), it is defined by `master_cycle_now` and `master_kernel_ns`, and counts at the rate of the guest TSC. When the master clock is *not* being used, because the host TSC is unreliable or because the guest TSCs are out of sync, the KVM clock is defined by the host's CLOCK_MONOTONIC_RAW and `kvmclock_offset`. So while operating in master clock mode, the `kvmclock_offset` value should actually *vary* as the clock calculated by the guest TSC drifts from the host's `CLOCK_MONOTONIC_RAW`.

In `use_master_clock` mode, the discontinuity is only exposed when a `KVM_REQ_MASTERCLOCK_UPDATE` request occurs. Even with a sane TSC, this *used* to happen on a vCPU hotplug, but that was fixed in commit c52ffadc65e2 ("KVM: x86: Don't unnecessarily force masterclock update on vCPU hotplug"). For reasons explained in that commit, the masterclock update does still happen when a guest TSC is written to a value which isn't in sync with the other vCPUs, but that should not be a problem in normal operation.

There is a special case here for the old MSR_KVM_SYSTEM_TIME, which historically was moved to MSR_KVM_SYSTEM_TIME_NEW purely because it sat in the wrong number range. But an old SUSE 2.6.16 kernel had a bug where if the reference point in the pvclock information was too far in the past (as happens in master clock mode), it would fail to boot. This was worked around in commit 54750f2cf042 ("KVM: x86: workaround SuSE's 2.6.16 pvclock vs masterclock issue") by treating the old MSR_KVM_SYSTEM_TIME the same as the unreliable-TSC case, and disabling masterclock mode support.

When setting shared_info page, the Xen support in KVM also triggers a `KVM_REQ_MASTERCLOCK_UPDATE`. This is a bug; it shouldn't. There are other cases in KVM which look wrong too, and should be audited.

Migration

For the TSC this is fairly simple. Live update can be performed in a cycle-accurate way as long as the *offset* method is used to save and restore the guest TSC across the update.

Preserving the TSC across live *migration* is harder. Fundamentally, the accuracy of any such migration is limited to the accuracy of the source and destination droplets' clocks, as synchronized by NTP or other methods. But KVM makes it harder than it needs to be. In fact the KVM_GET_CLOCK ioctl does return a full set of *{ real time, kvmclock, host tsc }* at a given moment. We can

ignore the kvmclock part of that and just use it to know the host TSC at a given time. The KVM documentation describes an algorithm to migrate guest TSCs by using this, although it doesn't take TSC frequency scaling or leap seconds into account.

The KVM clock, on the other hand, *ought* to be simple. If the TSC and its frequency are preserved correctly, then the `tsc_timestamp` and `system_time` fields of the KVM clock structure should be *identical*. Sadly, that isn't how KVM does things. All we have is the KVM_SET_CLOCK ioctl, which sets the KVM clock to a certain value at a given moment in *wallclock* time, but wallclock time is skewed by NTP and ambiguous due to leap seconds.

Previous attempts at fixes

- [PATCH v2] KVM: x86: add KVM_VCPU_TSC_VALUE attribute (sveith@amazon.com, 2023-02-02):
  This attempts to address the TSC migration problem. It adds a way to set the guest TSC at a given moment in KVM clock time.
- [RFC] KVM: x86: Add KVM_VCPU_TSC_SCALE and fix the documentation on TSC migration (dwmw2@infradead.org, 2023-09-13):
  This was a straw man proposal, highlighting the ugliness of the 'how to migrate TSC' documentation if we do it this way. I've since changed my mind, as the KVM_VCPU_TSC_VALUE proposal to set TSC based on KVM clock misses the fact that the KVM clock is derived f*rom* the TSC in the first place. And we need KVM_VCPU_TSC_SCALE to calculate and migrate the KVM clock precisely anyway.

Fixing the mess

- Add KVM unit test to validate that KVM clock does not change when provoked *(including by simulated live update).*
  It's OK for the reference point at `{ tsc_timestamp, system_time }` in the pvclock structure to change, but only such that it gives the *same* results for a given guest TSC — that is, if `system_time` changes, then `tsc_timestamp` must change by a delta which precisely corresponds in terms of the advertised guest TSC frequency. Perhaps allow a slop of 1ns for rounding, but no more.
- Audit and fix (i.e. remove) `KVM_REQ_MASTERCLOCK_UPDATE` usage, starting with `kvm_xen_shared_info_init()`. And work out whether it should be sent to *all* vCPUs, as some call sites do, or just one?
- Add KVM_VCPU_TSC_SCALE attribute to allow userspace to know the precise host→guest TSC scaling.
- Expose guest's view of KVM clock to userspace via `KVM_GET_CLOCK_GUEST` ioctl. Perhaps also a memory-mapped version, as the gfn_to_pfn_cache allows writing to userspace HVAs. With this, userspace has fast and accurate way to calculate the KVM clock at any given moment in time. (Currently, userspace calls the `KVM_GET_CLOCK` ioctl which is slow and returns inaccurate results). Then userspace can base other things like PIT and HPET emulation on the KVM clock and simplify timekeeping over migration for those too.
- Add a `KVM_SET_CLOCK_GUEST` ioctl which consumes the pvclock information back again. This should not only set the `kvmclock_offset` field, but *also* set the reference point `{ master_cycle_now, master_kernel_ns }` as follows:
  - Sample the kernel's CLOCK_MONOTONIC_RAW to create a new `master_kernel_ns` and `master_cycle_now`.
  - Convert the new `master_cycle_now` to a guest TSC.
  - Calculate the *intended* KVM clock with that guest TSC from the provided pvclock information.
  - Calculate the *current* KVM clock with that guest TSC using the *new* `master_cycle_now` and `master_kernel_ns` and `kvmclock_offset` as usual.
  - Adjust `kvmclock_offset` to correct for the delta between *current* and *intended* values.
  - Raise `KVM_REQ_CLOCK_UPDATE` on all vCPUs.

- Fix the broken `__get_kvmclock()` function to scale via the guest's TSC frequency as it should. There isn't necessarily a vCPU to use for this, so it's OK for this to work only when the frequency has been set of the whole VM rather than only for individual vCPUs. Likewise `kvm_get_wall_clock_epoch()` which has the same bug.
- Fix all other cases where KVM reads the time in two places *separately* and then treats them as simultaneous.
- Fix the discontinuities in `KVM_REQ_MASTERCLOCK_UPDATE` by allowing `kvmclock_offset` to vary while the VM is running in master clock mode. Perhaps every call to `pvclock_update_vm_gtod_copy()` which starts in master clock mode should follow the same process as the proposed `KVM_SET_CLOCK_GUEST` to adjust the `kvmclock_offset` value which corresponds with the new reference point. As long as we don't break in the case where something weird (host hibernation, etc.) happened to the TSC, and we actually *want* to trust `kvmclock_offset`. Maybe we should have a periodic work queue which keeps `kvmclock_offset` in sync with the KVM clock while the VM is in master clock mode?
- Correct the KVM documentation for TSC migration to take TSC scaling into account. Something like...
  - (SOURCE)
    - Sample both TAI and the (source) host TSC at an arbitrary time we shall call ***Tsrc***:
      - Use adjtimex() to obtain `tai_offset`.
      - Use `KVM_GET_CLOCK` to read UTC time and host TSC (ignoring the actual kvm clock). These represent time ***Tsrc***.
      - Use adjtimex() to obtain `tai_offset` again, looping back to the beginning if it changes.
      - Convert the UTC time to TAI by adding the `tai_offset`.
    - ∀ vCPU:
      - Read the scaling information with the `KVM_CPU_TSC_SCALE` attribute.
      - Read the offset with the `KVM_CPU_TSC_OFFSET` attribute.
      - Calculate this vCPU's TSC at the moment of `KVM_GET_CLOCK`, from the host TSC value.
    - Use `KVM_GET_CLOCK_GUEST` to read the KVM clock (on vCPU0).
  - (DESTINATION)`K`
    - Sample both TAI and the (destination) host TSC at a time we shall call ***Tdst***:
      - Use adjtimex() to obtain `tai_offset`.
      - Use `KVM_GET_CLOCK` to read UTC time and host TSC.
      - Use adjtimex() to obtain `tai_offset` again, looping back to the beginning if it changes.
      - Convert the UTC time to TAI by adding the `tai_offset`.
    - Calculate the time (in the TAI clock) elapsed between ***Tsrc*** and ***Tdst***. Call this ***ΔT***.
    - ∀ vCPU:
      - Calculate this vCPU's *intended* TSC value at time ***Tdst***:
        - Given this vCPU's TSC frequency, calculate the number of TSC ticks correponding to ***ΔT***.
        - Add this to the vCPU TSC value calculated on the source
      - Read the scaling information on the *current* host with the `KVM_CPU_TSC_SCALE` attribute
      - Calculate this vCPU's scaled TSC value corresponding to the host TSC at time ***Tdst*** *without* taking offsetting into account.
      - Set `KVM_CPU_TSC_OFFSET` to the delta between that and the *intended* TSC value.
    - Use `KVM_SET_CLOCK_GUEST` to set the KVM clock (on vCPU0).