Towards Linux as a Real-Time Hypervisor

Jan Kiszka
Siemens AG, Corporate Technology, CT SE 2
Otto-Hahn-Ring 6, D-81739 Munich
jan.kiszka@siemens.com

Abstract

Combining virtualization and real-time is important for an increasing amount of use cases, from embedded system to enterprise computing. In this paper, we will analyze the real-time capabilities of Linux as a hypervisor when using KVM and QEMU. We will furthermore introduce and evaluate a paravirtual scheduling interface that helps resolving priority inversion problems in embedded virtualization scenarios.

1 Introduction

Combining virtualization and real-time requirements is so far primarily a topic in embedded systems. In this domain, a general purpose operating system (GPOS) often needs to be executed aside a real-time operating system (RTOS). A hypervisor may then be used to manage shared resources and isolate the OSes from each other. Such architectures can be found in industrial control systems where the RTOS takes over the time-critical control of a machine while the GPOS runs, for example, the visualization software. Futher examples are single-processor smartphones where an RTOS is used to manage critical tasks of the radio communication while a GPOS hosts the typical set of mobile phone applications.

With the increasing importance of real-time in enterprise computing scenarios, the overlap with virtualization becomes larger as well. In the absence of highly demanding real-time requirements, virtualization is already used for server consolidation and load balancing. Thus, improving real-time capabilities of virtualization solutions will extend their usability also into the real-time enterprise domain.

But the gap between embedded real-time and enterprise virtualization is still large. In many embedded scenarios, specialized hypervisors are used that have a reduced feature set, are mostly statically configured, but can provide better performance and require less resources than full-featured versions. Hypervisors from the enterprise domain focus on good average performance as well as resource control at a comparatively high level. They manage significantly larger resource sizes and include mechanisms to dynamically shift load between physical nodes.

Linux as an operating system has already demonstrated that it can adopt to a very broad range of use cases: from low-end embedded up to large-scale machines with thousands of CPU cores [1]. In the real-time domain, Linux is currently approaching more demanding use cases with the PREEMPT-RT patch [2, 3]. And Linux includes built-in virtualization support with the KVM subsystem [4, 5]. Consequently, one question is what quality of service could already be achieved by combining these technologies–which furthermore include the advantages of open source, thus are also easily adaptable to specific requirements whenever needed. A realistic question is not if Linux can solve everthing, but how large the share in use cases is that it can cover and how it can be gradually extended so that all Linux users benefit.

2 Linux as Hypervisor

Using the Linux OS itself as a hypervisor requires two components: A kernel driver and a user space application. Both will be briefly introduced in the following.

2.1 Kernel-Based Virtual Machine

The Kernel-Based Virtual Machine (KVM) is a Linux device driver whose main task is to manage unprivileged access to virtualization features that can only be used directly by the privileged kernel. It was originally developed to grant this access to Intel's and AMD's x86 hardware-based virtualization extensions, but meanwhile provides essentially the identical interface on ia64, s390, and PowerPC hosts–even though not all of them provide comparable hardware virtualization support.

Another important task of KVM is to provide fast virtualization for frequently accessed guest devices. So far, interrupt and timer controllers are available as kernel-hosted models. The advantage is that no consultation of the controlling user space process is required if a guest accesses any of these devices, which reduces the virtualization overhead. In-kernel support for paravirtualized network adapters is currently under development.

KVM is a so-called type-2 hypervisor [6], that is it makes use of the host OS for tasks like setup/cleanup, scheduling, native interrupt and memory management etc. For this reason, the execution model of KVM is very simple: A Linux user space process acting as a hypervisor can request to create and configure a virtual machine (VM) by opening the KVM character device and issuing a series of IOCTLs. This VM shares the host memory map with its controlling process, while the hypervisor can freely configure the mapping between the VM's physical and its own virtual memory.

The hypervisor process can furthermore request to create virtual CPUs within the VM. Given a guest CPU state, they can then execute arbitrary guest code in a confined environment. Any guest activity that cannot be handled by the host CPU's virtualization extension will trigger an exit from the virtual CPU (VCPU) execution environment. If the KVM kernel code is not able to handle the exit internally, for example by swapping in a host memory page that the guest has requested by accessing a virtual page, the exit event is propagated to the user space hypervisor.

The execution context of the VCPU is implicitly defined by the user space hypervisor: It invokes the VCPU execution request from within one of its threads, and KVM preserves this scheduling context while running the guest code. Thus, if the Linux host scheduler decides to switch out a hypervisor thread, it is the corresponding guest CPU which is effectively scheduled out. The hypervisor process and/or the system administrator can therefore apply standard means at the level of threads, processes, users or control groups to influence the scheduling of virtual machines as well as further host services.

Guest exits also take place on asynchronous host events, foremostly interrupts. As soon as KVM has restored the host OS context, it re-allows host interrupt processing, thus the corresponding Linux interrupt handler is able to run. If the interrupt handler creates the need to reschedule, KVM detects this and invoke the corresponding service that possibly switches to a different host task.

2.2 QEMU

QEMU [7] is a fast CPU and peripheral hardware emulator. Its emulation does not depend on specific host features, that is it also works across architectures. The KVM project chose QEMU as the foundation for its user space hypervisor as it already came with most of the additionally required infrastructure like I/O device emulation.

KVM's QEMU version basically replaces the CPU emulation by the hardware-assisted execution provided by the kernel driver. It furthermore splits the single-threaded execution model of QEMU into VCPU threads and a main I/O processing loop, enabling more scalable SMP virtualization. As the QEMU core code is not yet prepared for multi-threaded execution, a global lock protects any access to the device emulation layer. Thus, only one VCPU or the I/O main loop may query or modify emulated devices of a virtual machine at the same time.

The KVM project also adds asynchronous I/O (AIO) support to QEMU. As Linux's native AIO implementation performs worse than a pthread-based approach, QEMU gained AIO via a thread-pool: for each AIO request a user space thread is spawned or obtained from the pool that executes the request and then signals the completion to the main I/O thread.

Although the upstream QEMU version meanwhile includes KVM support, we developed our modifications only for KVM's QEMU branch [8, commit f2593a0] as it includes some required features that have not yet been merged into upstream.

3 Prioritizing QEMU/KVM

A straightforward approach to improve guest responsiveness over QEMU/KVM is to raise the priorities of QEMU's threads and lift them into a real-time scheduling class. This comes with the risk of starving host services that the guest may indirectly busy-wait upon. For example, the guest could issue an I/O request that is handled asynchronously by the host, but requires completion within the context of non-real-time service like the kernel's event or AIO thread.

This risk can mitigated by avoiding CPU overcommitment, that is running less VCPUs than real processor cores exist. Furthermore, Linux comes with a real-time throttling mechanism [9] that restricts the CPU slice of all real-time tasks in the host system. The latter approach, however, can result in very low performance as the guest system consumes a lot of CPU time without making relevant progress.

3.1 Realization

While QEMU's main I/O thread and its VCPU threads could also be adjusted after start-up via `chrt`, its AIO thread pool grows and shrinks dynamically. Therefore a QEMU extension was required to establish a stable prioritization of all involved threads. We defined a new command line switch for QEMU:

```
-rt maxprio=prio[,policy=ff|rr|ts]
    [,aioprio=prio][,aiopolicy=ff|rr|ts]
```

If `aioprio` and `aiopolicy` are omitted, the scheduling `policy` is applied to all existing and future threads of the QEMU process. QEMU's main thread and its AIO threads then gain the specified maximum priority, while VCPU threads will use the next lower priority level. This ensures that a spinning guest cannot starve itself from receiving I/O events that are routed through those threads. However, if asynchronous I/O is not considered most important for a guest, the priority and policy of corresponding completion threads can be separately specified. The effect is discussed in Section 3.3.

We furthermore added

```
mlockall(MCL_CURRENT | MCL_FUTURE);
```

in case a real-time scheduling class is selected. This prevents any swapping of QEMU's process memory, including guest pages, and avoids lazy mapping. While `mlockall` and memory overcommitment are generally not compatible, real-time requirements deny the latter in most cases anyway.

A further real-time enhancement for QEMU concerned switching mutexes used for synchronizing I/O handling to the priority inheritance protocol. The impact of this change, however, can be considered minor at this point, as it is pointed out in Section 6.

3.2 Test Setups

Our test setup consisted of a host system with an Intel dual-core processor (Core2 T5500 at 1.6 GHz) with 2 GB of RAM. Host as well as guest systems were based on OpenSUSE 11.1. We started our evaluation with a host kernel built out of KVM's git tree [10, commit 0b972aa], which basically generated a 2.6.31-rc4 kernel. We enabled `CONFIG_PREEMPT` and `CONFIG_PREEMPT_RCU` and disabled `CONFIG_ACPI_PROCESSOR` for best latencies. Tracing facilities were built in, but remained disabled during latency tests (`echo 0 > /proc/sys/kernel/ftrace_enabled`, `echo 0 > /sys/kernel/debug/tracing/tracing_enabled`).

To limit the size of the test matrix, we decided to focus on a single real-time aspect in the following evaluations: the latency of timed high-priority user space threads in the guest system. For measuring this latency, we used cyclictest from the rt-test project [11, v0.50] with the following parameters: `-m -n -p 99 -l 1000000 -h 100000 -q`, that is a 1 ms periodic nanosleep test over 1 million iterations. As our kernel configuration allowed both host and guest system to use the native TSC as clock source, we were able to rely on the measurement results obtained within the guest. Moreover, our interest in the latency distribution was stronger than in a reliable upper bound for the worst case. We therefore ran all tests only for 15 minutes, which is typically not enough to obtain maximum latencies but sufficient to compare distributions.

The guest kernel was kept identical for all tests: 2.6.29.6-rt23 plus paravirtual scheduling patches for Linux guests that are introduced in Section 5. The latter extensions were always enabled as Linux disables them automatically during boot-up if the hypervisor does not provide this feature.

QEMU was started with the parameters

```
qemu-system-x86_64 -m 512 OpenSuse_64.raw \
    -nographic -serial /dev/null -net nic \
    -net user,hostfwd=::2222-:22
```

that is using 512 MB RAM, a raw disk image and no local console. Instead we controlled the guests via a forwarded SSH connection. The reasons for choosing a raw image over QEMU's QCOW2 disk format and avoiding local graphic or even serial console emulations is discussed in Section 6.

The tests on PREEMPT-RT host kernels required us to bind prioritized QEMU to the second core of our host CPU as we otherwise faced livelocks during boot-up. We did not analyze the reason for this in further detail, but instead decided to establish this CPU binding consistently for all tests. The binding to the second core was not only applied to the QEMU process but also to the load applications as well as the host's disk interrupt. A positive side-effect of moving the tests was that we were able to avoid occasional latency peaks of a few hundred microseconds due to system management interrupts (SMI) which apparently only hit the first core on our platform.

We defined two load applications that were running in parallel in endless loops during the tests: bonnie 1.4 (as provided by OpenSUSE 11.1) was executed with `-y -s 2000` to stress the I/O subsystems. Furthermore, the cache calibrator [12] was run with `1660 4M /tmp/calibrator.log`, that is a 4 MB RAM test area to stress memory caches. For all tests, the calibrator load stayed on the host as the intended cache pollution can be triggered from both host and guest domain equally well.

3.3 Evaluation

To evaluate the effect of raising QEMU's thread priorities, we first measured the latencies natively achievable with cyclictest on the host. For this test, the bonnie load ran on the host, and no QEMU instance was started. Figure 1 shows the result for the `CONFIG_PREEMPT` host kernel we used in this first round. Note that the axes in all graphs are logarithmically scaled.

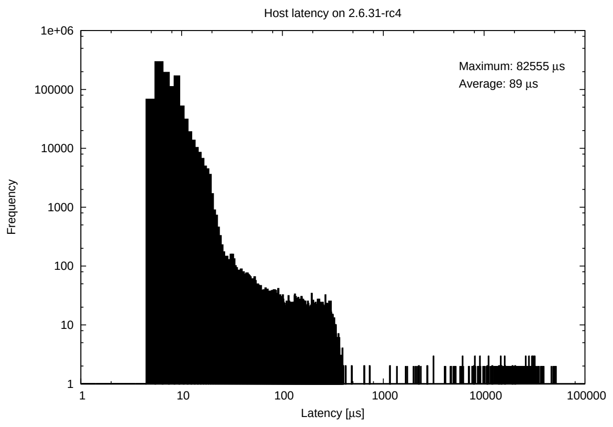

FIGURE 1: *Host Latency*

Next we measured the guest latency without any QEMU prioritization. Bonnie now ran inside the guest. In Figure 2, the difference to the host latency is visible in form of a shifted frequency maximum and a much higher probability of multi-millisecond latencies. In fact, the selected upper bound for the histogram of 100 ms did no catch the maximum latencies. Clearly, this setup is unsuited for latency sensitive guest load.

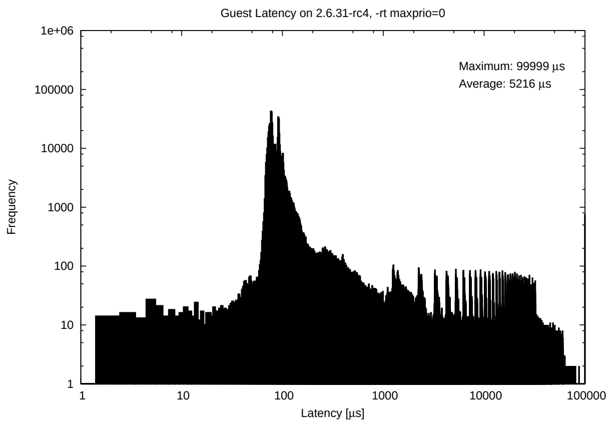

FIGURE 2: *Unprioritized Guest Latency*

We then applied `SCHED_FIFO` with a priority of 98 for I/O and 97 for VCPU threads on QEMU. The result is depicted in Figure 3: the average latency dropped significantly.

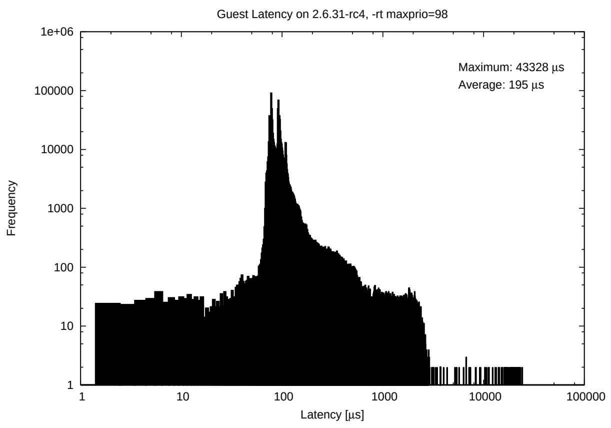

FIGURE 3: *Prioritized Guest Latency*

In the aim to reduce the I/O load impact on our test scenario, we moved QEMU's AIO threads out of the real-time scheduling class again. Figure 4 visualizes that this kept the overall distribution shape but reduced the probability of high or low latencies.

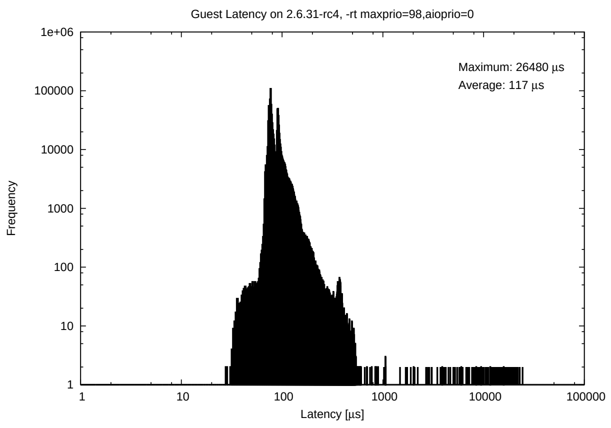

FIGURE 4: *Lowered AIO Priority*

Processing guest originated I/O load, even if handled asynchronously, apparently contributes to excessive latencies in cyclictest. To confirm this assumption, we moved the bonnie test back on the host. This roughly resulted in the same latency distribution as in the previous experiment, see Figure 5
.

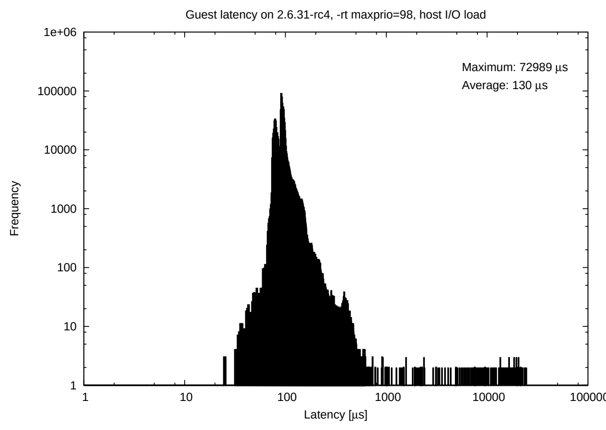

FIGURE 5: *I/O Load Moved to Host*

4 PREEMPT-RT hosting

In the next step, we switched the host to the same PREEMPT-RT kernel that was already in use for the guest system. The aim was to evaluate if the I/O-related priority inversions we found on a standard kernel are caused by QEMU/KVM itself or by the host kernel not respecting priorities properly in all cases.

4.1 Priority Assignment

Selecting the maximum priority of QEMU under PREEMPT-RT has to be done carefully in order to avoid livelocks. A Linux guest, for example, performs a timer check during boot-up, busy-waiting for timer interrupts to arrive. On the host side, these interrupts flow from the timer IRQ handler via the hrtimer kernel thread to QEMU's I/O thread and finally to the VCPU. If the hrtimer thread is not given a higher priority than QEMU, the guest will spin endlessly, preventing any timer IRQ delivery. As our test setups focus on guest timer IRQ latency, we lifted hrtimer processing above any others interrupt threads.

```
# chrt -p -f 99 `pgrep hrtimer/1`
```

4.2 Evaluation

Again we measured the timer latency via cyclictest on the host first, running bonnie and calibrator as load. Figure 6 shows promising numbers. They indicate that PREEMPT-RT is avoiding priority inversions because of low priority I/O load as far as this short test can reveal.

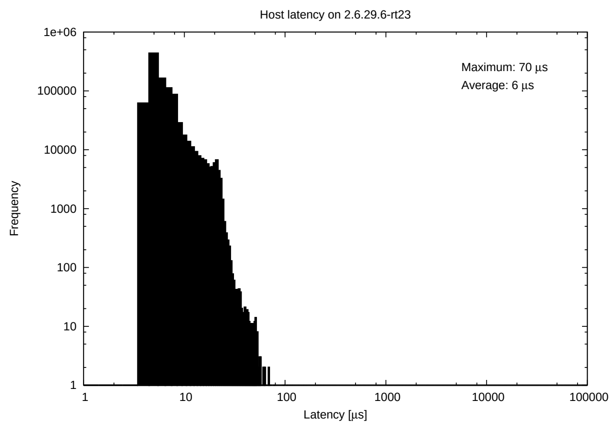

FIGURE 6: *PREEMPT-RT Host Latency*

Running QEMU with raised priority on PREEMPT-RT results in clear worst case latency improvement over a standard PREEMPT kernel. Figure 7 depicts this measurement; it corresponds to Figure 3. On PREEMPT-RT, the latency maximum moved below 100 μs, and the average latency dropped by one third.

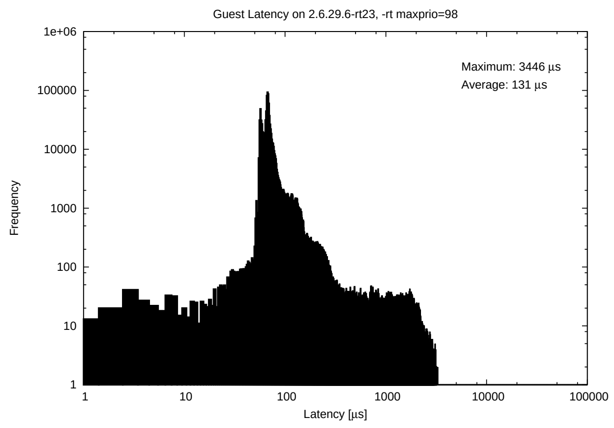

FIGURE 7: *Guest Latency on PREEMPT-RT Host*

Corresponding to the test shown in Figure 4, we next dropped AIO priorities to 0. Figure 8 visualizes the best guest latency distribution we were able to achieve with our test setup. Here the latency average drops to only 75 μs, and a maximum latency of less than 500 μs was observable (again noting that the test only ran for 15 minutes).

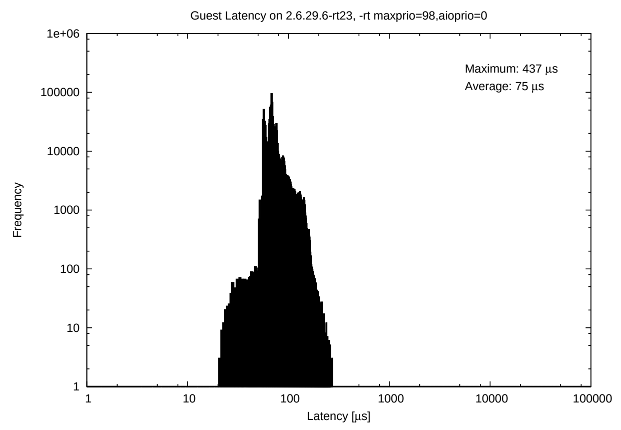

FIGURE 8: *PREEMPT-RT with Lowered AIO*

5 Paravirtualized Scheduling

To fulfill the requirements of real-time tasks running in a VM, the hypervisor may assign guaranteed resource time shares to that VM. All real-time task requirements must then be fulfillable by granting an adequate time slice in a predefined scheduling cycle. Such a static time slice allocation can be impractical in a highly reactive environment.

Alternatively, the hypervisor may prioritize the real-time VM over any other VM. This approach is sufficient as long as the real-time VM only executes tasks that have a higher importance than *any* other task in the system. Otherwise, background jobs in the prioritized VM may use up all CPU resources of the host, starving the other VMs. A straightforward approach to avoid this priority inversion is to give the hypervisor a hint about the internal states of its guests. The hypervisor can then adjust the schedule of the VCPUs accordingly.

Such an approach is already used for resolving efficiency issues related to guest CPUs waiting on contended spinlocks [13]. Consider one VCPU holding a spinlock, then being preempted by another VCPU which starts to spin on the very same lock. Without any knowledge of the guest state, the hypervisor can only keep the second VCPU running until it consumed its time slice. But given some information that the second VCPU is currently spinning on a lock, the hypervisor can instead grant the host CPU to the first VCPU that is able to make progress on its jobs. Moreover, there is a probability that this VCPU will have released the spinlock the next time it is preempted by the second VCPUs. Thus, the overall efficiency increases as less CPU time is burned unproductively.

Today this additional information about spinlock contentions can only be provided by the guest OS itself. Linux introduced paravirtual spinlock operations for this purpose. These replace the native spinlock code when Linux detects that it running over a hypervisor which supports an alternative mechanism. Currently only Xen makes use of them [14]. Upcoming Intel and AMD CPUs will include hardware-assisted detection of spinlock loops, which triggers a guest exit so that the hypervisor can schedule a different VCPU.

However, hardware assistance cannot be expected for solving the scheduling problem of prioritized guest CPUs because task scheduling decisions are done in software on commodity platforms like x86. Therefore, we need a software interface between guest and hypervisor to propagate the required information, and we have to define an execution model which both hypervisor and guest OSes can adopt.

5.1 Execution Model

The least common denominator in task scheduling today is fixed-priority scheduling. We chose POSIX [15] to define the detailed semantics of the supported scheduling policy. The following policies are so far supported by our model:

- `SCHED_OTHER`:
  Default time-sharing policy of the host.
- `SCHED_FIFO`:
  FIFO policy as defined by POSIX.
- `SCHED_RR`:
  Round-robin policy as defined by POSIX.

The hypervisor schedules a VCPU according to the last scheduling policy and priority that have been reported by the guest. The available priority range is $[0, p_{max}]$. p_{max} is a per-VCPU priority limit that the hypervisor can be configured to. The guest-provided priority p_{guest}, confined to the range of $[0, 99]$, is mapped to the host priority p according to

$$p = \left\lfloor p_{guest} \frac{p_{max}}{99} \right\rfloor$$

The guest-provided scheduling parameters apply if no virtual interrupt is waiting to be injected or is currently processed by the guest. As soon as the hypervisor marks an interrupt for injection into a VCPU, it raises the VCPU to p_{max}. If the guest's current scheduling policy is `SCHED_OTHER`, it is changed to `SCHED_FIFO`, otherwise it is left unmodified. The boost is kept until the guest reports that it completed interrupt handling on the corresponding VCPU.

The model also supports one level of interrupt nesting to allow for accurate non-maskable interrupt (NMI) virtualization. If a virtual NMI is marked pending while interrupt boosting is already applied, the NMI boosting is added. In the following, completion for both interrupt types has to be reported by the guest CPU for deboosting it to its base policy and priority.

5.2 Interface

Our current version of a paravirtual scheduling interface requires two new hypercalls, that is paravirtual service calls from the guest to the hypervisor:

- **Set Scheduling Parameters**
  This hypercall informs the hypervisor about a potentially changed scheduling policy and/or priority for the specified VCPU. The hypercall's argument list consists of the target VCPU, the new scheduling policy and the new priority value p_{guest}.

  In order to support SMP guest use case where one VCPU may change the scheduling parameters of another VCPU, the hypercall interface can also be invoked on remote VCPUs. In our implementation for x86, we use the physical ID of the emulated local APICs to specify the target VCPU of this hypercall.

  If the target VCPU is the same as the caller of the hypercall and the VPU currently undergoes interrupt priority boosting, the boost is cleared so that no additional *Interrupt Done* hypercall has to be issued.

- **Interrupt Done**
  This hypercall has to be invoked by a VCPU that finished its interrupt handling but will not invoke *Set Scheduling Parameters* as no reschedule is required. No arguments are passed.

For the integration in QEMU/KVM, we chose KVM's own hypercall interface that is based on vmcall/vmmcall, the vendor-specific x86 instructions to call from a guest into its hypervisor. Like other KVM hypercall interfaces, this extension is advertised to the guest via a flag in KVM's CPUID leaf.

5.3 Host-Side Realization

The modifications to implement our paravirtual scheduling interface in QEMU/KVM focused on KVM's kernel module. Here we had to provide the *Set Scheduling Parameters* hypercall interface which, after checking and mapping the passed parameters as well as potentially deboosting the VCPU, finally invokes `sched_setscheduler()` for the Linux task of the corresponding VCPU.

Standard interrupt handling was also straightforward to implement. We hooked into `kvm_vcpu_kick()` and the IOCTLs to trigger an interrupt or an NMI from user space and applied the required boosting. Futhermore, we added the *Interrupt Done* hypercall interface.

More challenging was the implementation of VCPU boosting on APIC and PIC timer events. To avoid a priority boost gap between the host timer IRQ and the guest IRQ injection, we have to start the boosting as soon as the host timer fires. KVM uses Linux hrtimers for setting up guest timer events. When these timers fire, a KVM-provided handler is invoked, looks up the target VCPU and sets a flag that pending timers have to be considered on next guest entry.[1] It is not yet clear at this point if an IRQ or NMI will finally be injected. Moreover, on non-PREEMPT-RT kernels, the hrtimer handler runs in interrupt context, preventing a direct invocation of `sched_setscheduler()` due to the IRQ-unsafe implementation of that service.

To work around this, we register per-CPU kernel threads, running at highest host priority, forwarding our boost requests from the timer handlers to the Linux scheduler. We additionally introduce an intermediate boosting state, "timer pending", that is kept until KVM decides if the pending timer will raise an IRQ or an NMI. This boost is not affected by any guest-originated clearing.

As we rely on the physical APIC ID for looking up the internal VCPU state during the *Set Scheduling Parameters* hypercall, our implementation is so far limited to the standard case of in-kernel guest IRQ controller emulation. All information we need is at hand here, while it would otherwise reside only in user space.

To configure p_{max} for every VCPU, the new IOCTL `KVM_SET_MAX_PRIO` is now available for KVM's VCPU file descriptors. User space is expected to call it during setup of a new virtual CPU.

The QEMU user space part only had to be extended by the `pvsched=on|off` option for the `-rt` command line switch. It controls whether paravirtual scheduling is advertised to the guest and VCPU maximum priorities are configured to the value defined by `maxprio` minus one.

[1]A plain flag is enough as timers always fire on the same host CPU the associated VCPU may run on. Thus, the guest is either already preempted by the host timer and the flag will simply be evaluated on guest re-entry, or this will happen as soon as the VCPU resumes (i.e. re-enters guest context).

5.4 Guest-Side Realization

To demonstrate the guest-side implementation we chose Linux as well. Linux has the advantage of providing a well-established OS paravirtualization layer called paravirt-ops. It comes with detection and runtime deactivation services that allow to reuse a image in the absence of paravirtual scheduling support without noticeable overhead.

We first introduced a new group of paravirt-ops, `pv_sched_ops`. This group contains two callbacks that directly map on two previously defined hypervisor calls. The wrappers for these calls are;

```
void cpu_set_sched(int cpu, int policy,
                   int priority);
void cpu_interrupt_done(void);
```

`cpu_set_sched()` is called from `schedule()`, equipped with the parameters of the next task. Furthermore, we hook into `__sched_setscheduler` where we call `cpu_set_sched()` if the manipulated thread is the current one on its assigned CPU.

`cpu_interrupt_done()` is inserted into `nmi_exit()`, where it is called unconditionally, and into `irq_exit()`. In the latter case, we only invoke the hook if we left interrupt handling and no reschedule is pending that will deboost the current CPU anyway. The following code extract shows the modified function:

```
void irq_exit(void)
{
    ...
    sub_preempt_count(IRQ_EXIT_OFFSET);
    if (!in_interrupt()) {
        if (local_softirq_pending())
            invoke_softirq();
        if (!need_resched())
            cpu_interrupt_done();
    }
    ...
}
```

While the `pv_sched_ops` declaration and setup is currently only available for x86, our hooks were exclusively added to generic code and, thus, are portable to other architectures. Moreover, no adaption of this paravirtual interface was required to use it in a PREEMPT-RT guest. But we were forced to disable KVM's paravirtual MMU operations as their current implementation is incompatible with PREEMPT-RT's MMU subsystem changes.

5.5 Evaluation

Ideally, enabling paravirtual scheduling should not have any negative impact on our test scenarios. It should not raise the worst case latency nor shift the average. Realistically, paravirtual scheduling introduces overhead, both related to leaving the guest to inform the host about priority changes as well as applying priority changes because of guest activity or interrupt injections.

To evaluate the impact, we repeated the test of running cyclictest with raised maximum but low AIO priority both over the standard as well as the PREEMPT-RT kernel, this time with `pvsched=on`. The results can be found in Figures 9 and 10 that correspond to Figures 4 and 8, respectively. We see a slight increase in the average latency on both host kernels, which is assumed to reflect the paravirtual scheduling overhead. Moveover, we see a higher worst case latency on the standard PREEMPT kernel. But as this increase does not repeat over PREEMPT-RT and the tests may have not revealed the actual worst case, we consider this a random effect.

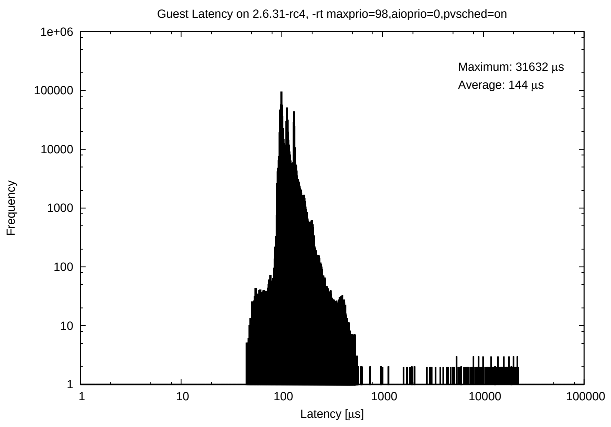

FIGURE 9: *Paravirtual Scheduling on* *`CONFIG_PREEMPT`*

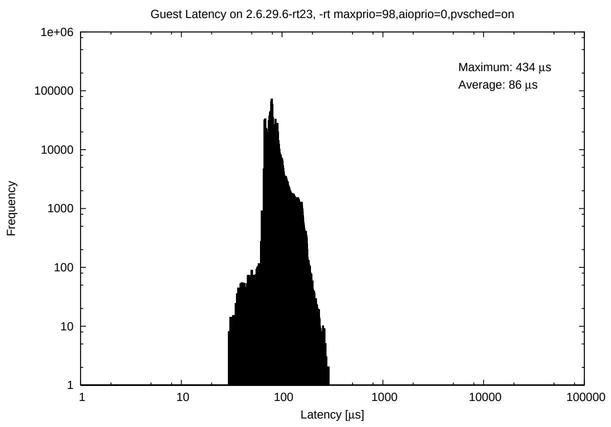

FIGURE 10: *Paravirtual Scheduling on* *`CONFIG_PREEMPT_RT`*

6 Analysis

The measurements of the real-time properties QEMU/KVM can provide for its guest have shown the following:

- With careful tuning, worst case timer-driven guest scheduling latencies below 1 ms and average latencies below 100 μs can be achieved on a PREEMPT-RT host kernel.
- Parallel I/O activity of the guest has significant influence on its overall latency. This can be mitigated by lowering the priority of AIO completion handling.
- Our paravirtual scheduling interface can be implemented in a way that no obvious priority inversions are introduced, while only causing a slight average latency increase.

We furthermore observed the following effects during the tests:

- Using QCOW2 images (backing or temporary) imposes high latencies on the guest, irrespective of PREEMPT-RT use and careful priority settings.
- Using local SDL graphic output or applying high load on the emulated serial port also causes priority inversions for the guest.

The latter effects are apparently related to the locking scheme in QEMU user space part. As a global "I/O lock" is held while accessing potentially blocking Linux services, every concurrent access to QEMU's I/O device model, for example by a VCPU, will suffer from the same delay. These bottlenecks are expected to cause problems also on large SMP guest systems as the number of parallel I/O access naturally increases here. On the long-term, QEMU's device model will therefore have to be made thread-safe by applying a more fine-grained and scalable locking scheme.

Yet hidden behind the above effects and/or not sufficiently stressed by our tests are delays caused by significant CPU load inside QEMU's I/O layer. For example, the user space networking stack may face large packet queues or decide to fork-off helper programs. Such paths require further analysis, even when a more scalable locking scheme has already been establish. They could still impact their direct caller, namely VCPUs issuing synchronous I/O operations or the main I/O thread dispatching incoming data from the host.

As the priority inversion effects in user space dominated our tests, we had no closer look at the locking situation in KVM's kernel model. However, at the time of writing, there were ongoing activities to establish a more fine-grained locking scheme also in kernel space [16].

7 Conclusion

In this paper, we analyzed QEMU/KVM regarding its currently achievable soft real-time capabilities. We demonstrated that, when carefully tuning the setup and using PREEMPT-RT on the host, sub-millisecond scheduling latencies inside guests can be achieved.

We furthermore introduced a paravirtual scheduling interface. It overcomes the priority inversion problems on embedded systems that are related to black-box scheduling of guests. Measurements demonstrated that our first implementation does not introduce obvious priority inversions on host or guest side. It currently comes with an overhead that results in 15% to 25% higher average latencies and is expected to have a measurable impact on the overall system performance as well. This impact should be reducible by optimizing our approach, specifically by avoiding unneeded guest exists due to paravirtual scheduling updates. We expect improvements by skipping updates on context switches that do not change any parameters and by using lazy update algorithms.

Along with the publication of this paper, we will also release the source code that was developed for this research work. Our goal is to evolve the code base towards mainline acceptability.

References

[1] *Top 500, Operating System share for 06/2009*, TOP500.Org, 2009. http://www.top500.org/stats/list/33/os

[2] *Internals of the RT Patch*, Steven Rostedt, Darren Hart, Proceedings of the Linux Symposium, 2007.

[3] *Real-Time Linux Wiki*, 2009. http://rt.wiki.kernel.org

[4] *KVM: The Linux Virtual Machine Monitor*, Avi Kivity et al., Proceedings of the Linux Symposium, 2007.

[5] *Kernel Based Virtual Machine*, 2009. http://www.linux-kvm.org

[6] *IBM Systems Virtualization, Version 2 Release 1*, IBM Corporation, 2005. http://publib.boulder.ibm.com/infocenter/eserver/v1r2/topic

[7] *QEMU - open source processor emulator*, 2009. http://www.qemu.org

[8] *qemu-kvm git repository*, 2009. http://git.kernel.org/?p=virt/kvm/qemu-kvm.git

[9] *Real-Time group scheduling*, Linux kernel 2.6.31, Documentation/scheduler/sched-rt-group.txt, 2009.

[10] *kvm kernel git repository*, 2009. http://git.kernel.org/?p=linux/kernel/git/avi/kvm.git

[11] *RT test utils*, 2009. http://git.kernel.org/?p=linux/kernel/git/tglx/rt-tests.git

[12] *The Calibrator (v0.9e)*, Stefan Manegold, 2004. http://homepages.cwi.nl/ manegold/Calibrator

[13] *Preventing Guests from Spinning Around*, Thomas Friebel, Xen Summit 2008. http://www.xen.org/files/xensummitboston08/LHP.pdf

[14] *[PATCH RFC 0/4] Paravirtual spinlocks*, Jeremy Fitzhardinge, 2008. http://permalink.gmane.org/gmane.comp.emulators.xen.devel/53239

[15] *The Open Group Base Specifications Issue 6, IEEE Std 1003.1*, IEEE and Open Group, 2004.

[16] *KVM Mailing List: [RFC] more fine grained locking for IRQ injection*, Gleb Natapov, 2009. http://permalink.gmane.org/gmane.comp.emulators.kvm.devel/37114