KVM vs. Message Passing Throughput

Reducing Context Switching Overhead

Rik van Riel, Red Hat

KVM Forum 2013

KVM vs. Message Passing Workloads

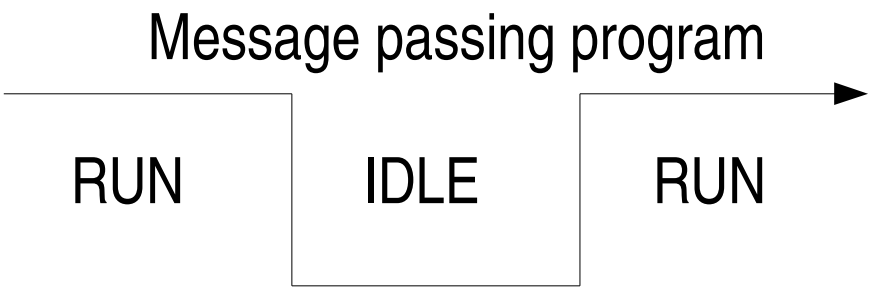
- Message passing workloads
    - What is going on?
    - Where is the overhead?
- Underlying cause: optimizations
- Potential solutions
    - Paravirtualized C-state driver
    - Lazy FPU switching
    - Lazy context switching (idle poll)
- Conclusions

Message Passing Workload

- Any workload that
  - Sends many (short) messages
  - Spends little time processing each message
  - Spends time waiting for a reply (or the next message)
  - Sometimes short intervals between messages
- Network
- IPC
- Process internal “messages”, eg. Java thread locking
- Example workloads:
  - Transaction processing
  - Web server with database

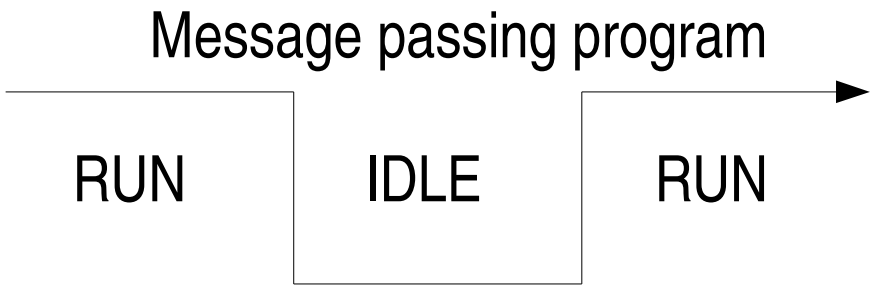
Message Passing Workload

- Process an event
- Wait for the next event
- Process that event
- Frequent transitions between running and idle



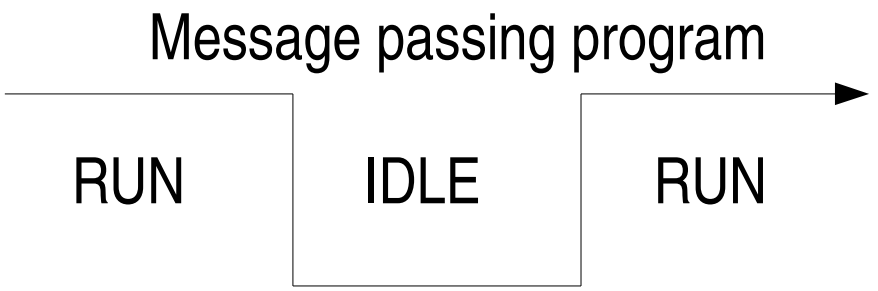
Message Passing Workload (+ kernel)

- On running -> idle transition, kernel:
  - Stores registers
  - Runs scheduler
  - Stores FPU/extended context
- On idle -> running transition, kernel:
  - Restores registers
  - Reloads MMU context (sometimes)
  - Restores FPU/extended context (HW optimized)



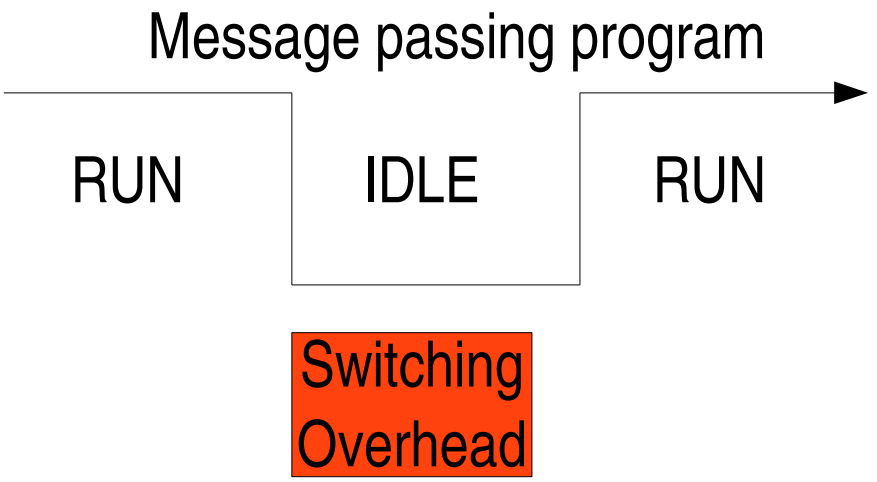
Message Passing Workload (+ virt)

- On running -> idle transition:
  - Trap to host, VMEXIT
  - Store FPU/extended state (unconditionally)
  - Update VCPU state, run scheduler, etc.
- On idle -> running transition:
  - Restore FPU/extended state (HW optimized)
  - Restore guest state, VMENTER



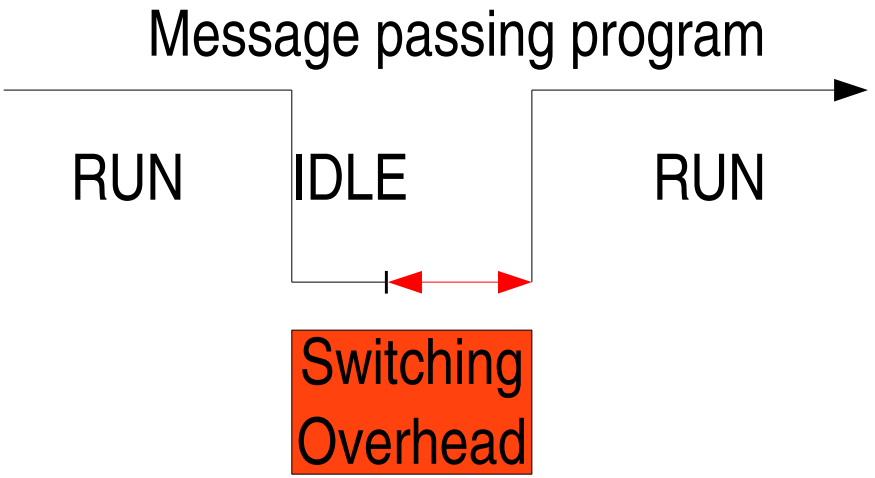
Visualizing the problem

- If the overhead is less than the idle time, things are fine



Visualizing the problem

- If the overhead is more than the idle time, big trouble
- Idle time can be arbitrarily short
  - Just send more messages
- What if the idle time is half the switching overhead?
  - Program stays “idle” for twice as long as desired!
  - If run time per message is short, throughput can be reduced severely

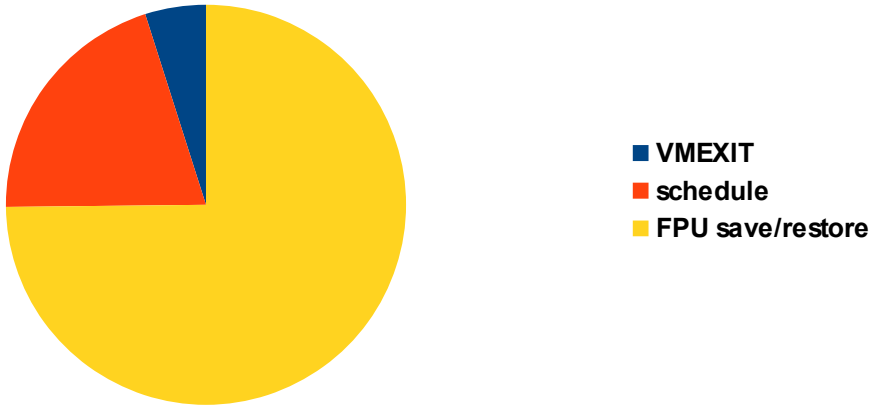


Where is the overhead?

- Idle -> running transition
  - Restoring FPU/extended state
    - Optimized by hardware, if FPU has not been used since the guest stopped running (FXSTOROPT)
  - Guest IRQ code
    - Optimized with x2apic & PV-EIO
- Running -> idle transition
  - VMEXIT, storing VMCS
    - VMEXIT is fast on modern hardware
    - KVM does lazy store, largely optimized away
  - Storing FPU/extended state (FXSAVEOPT)
    - Always slow, due to hardware optimizations
  - Schedule takes some CPU time

Measuring the overhead

- Java test program with reader/writer threads, and a lock on the queue
- Various things tried:
  - default (vcpu_put/schedule/vcpu_load): 135531 / s
  - vcpu_put/safe_halt/vcpu_load: 150476 / s
  - vcpu_put/safe_halt/vcpu_load, skip FPU save: 205711
  - safe_halt (no vcpu_put / vcpu_load): 214624 / s
  - never trap to host (yield_on_hlt = n): 218260 / s



Underlying cause: optimizations

- The system is optimized for busy, not for idle
  - Guest can context switch without trapping to the host
  - Includes guest reloading FPU “extended context”
  - Guest context switches done without any additional overhead
    - Including updating VMCS, etc...
- When guest goes idle:
  - Call HLT instruction for power saving
  - HLT traps to host, checks if something else needs CPU
- Host does not know if the guest did a context switch
- Host does not know if guest used FPU
  - Host needs to be safe, and save all guest context info
  -

Potential solutions

- Avoid trapping to the host on short pauses
  - Paravirtualized C-state driver
- Lazy FPU switching
  - Leave contents in the FPU registers
  - Only save them when somebody else needs the FPU
- Lazy context switching
  - Keep the current context in the CPU when going idle

Paravirtualized C-state Driver

- Bare Metal C-state (cpuidle) Driver
- Paravirtualized C-state Driver
- Paravirtualized C-state Driver Issues

Bare Metal C-state (cpuidle) Driver

- Puts the CPU in power saving modes
- At sleep time:
  - Guesses how long the CPU will be idle for
  - Puts the CPU in a power saving mode appropriate for that sleep time
    - Look up the correct mode in a table of recommended sleep times (and wakeup latencies)
    - Longer sleep? Deeper power saving mode
  - All it has to do is predict the future
- At wakeup time:
  - Subtract wakeup latency actual sleep time, compare with estimate (to correct future predictions)
- Paravirtualized will be a little harder...

Example cpuidle table

Nehalem states	Exit Latency (uS)	Target Residency (uS)
Poll	0	0
C1-NHM	3	6
C1E-NHM	10	20
C3-NHM	20	80
C6-NHM	200	800

Paravirtualized C-state Driver

- A cpuidle (c-state) driver for virtual machines
- Disable automatic trap-on-HLT on the host side
- Host exports table with wakeup latencies and target residency times to the guest
- When a guest goes idle:
  - Estimate idle time
  - Select idle state
- Only two states available
  - For short idle times, stay in the guest
  - For long idle times, trap to the host
    - Host can use power saving, or
    - Host can run something else

Paravirtualized C-state Driver Issues

- The PV C-state driver idea has some fundamental issues
- Unpredictable wakeup latencies
  - Who knows what the host will be doing?
    - Not even the host knows in advance
  - Difficult to fill in the wakeup latency table
- Larger variability will make predicting harder
- Not trapping to the host for short idle times
  - Guest uses CPU time while idle
  - Host may want to run something else when the guest needs the CPU again
- Staying on the CPU may keep the program that would answer our messages from running
  - Delaying the thing we are waiting for is counter-productive, and could lead to strange feedback effects

Example cpuidle table

Nehalem states	Exit Latency (uS)	Target Residency (uS)
Poll	0	0
C1-NHM	3	6
C1E-NHM	10	20
C3-NHM	20	80
C6-NHM	200	800

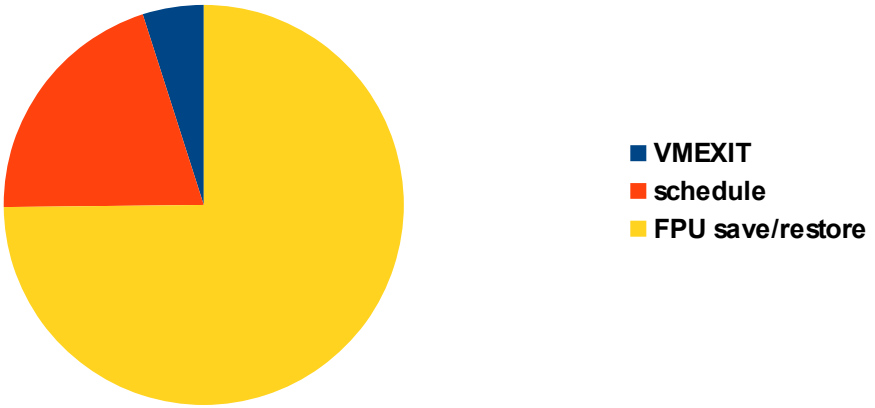
KVM states	Exit Latency (uS)	Target Residency (uS)
Stay in guest	0	0
Trap to host	???	???

Lazy FPU switching

- Delay FPU context save after context switch
  - Until something else needs the FPU, or
  - Until the thread runs somewhere else, or
  - Until some debugger or tracer needs the FPU context
- Send IPI if FPU context needs to be saved on remote CPU
  - Could make FPU saving even slower than it already is...
  - Could hit IPI + power saving latencies!
- Complicates the FPU code
- Unclear if this benefits anything besides KVM
- First implemented by Avi Kivity in 2010
- Shot down by Ingo, for reasons above
- Do we need something more generic?

Lazy Context Switching

- Goal: reduce both FPU and schedule overhead



- Introduction
- Details
- Default Poll Function
- Workflow
- Tradeoffs

Lazy Context Switching Introduction

- Context switch overhead is not just hurting KVM
  - Very fast IO devices (millions of IOPS/s) are slowed down by tasks sleeping to wait for IO completion
  - A second user is a reason for infrastructure...
- Basic idea:
  - Allow any task to become an idle task temporarily
    - Instead of context switching to the idle task
  - Avoid context switch overhead for short sleeps
  - Treat CPU as idle CPU
    - Power saving during longer sleeps
    - Run other tasks if they need CPU time

Lazy Context Switching Details

- New kernel function:
  - idle_poll
  - Gets two pointers in idle_poll_struct
- Poll function:
  - Checks whether the task is done waiting, and should continue running
  - Argument to the poll function (if necessary)

```
int idle_poll(struct idle_poll_info *ipi)

struct idle_poll_info {
        int (*poll)(void *);
        void *data;
};
```

Lazy Context Switching Details

- Task preemption
  - Task state needs to be saved on preemption
  - Other things may need to be done
  - preempt_notifiers already exist, no need for new code
- Wait list setup
  - For KVM, kvm_vcpu_block adds the vcpu thread to a waitqueue
  - Allows vcpu_kick to wake the thread
  - This can continue like before

Lazy Context Switching Default Poll Function

- KVM has not much status to check
  - Only “did the task get woken by vcpu_kick?”
  - Did task->state change to TASK_RUNNABLE?
- This is generic functionality, which could be used by others
- Implemented in idle_poll_default()
  - Switch to other task, if there is a runnable one
  - Place CPU in lazy TLB mode
  - Identify CPU as idle to idle_cpu()
  - Run idle balancer
  - Call CPU power saving code, in case of long sleep
  - Make sure preempt notifiers are set

Lazy Context Switching Workflow

- If CPU stays idle, until original thread returns:
  - Mask task as running (non-idle) again
  - Context switch is avoided
    - Expensive FPU/extended state save avoided
- At task wakeup time
  - Wake it up on the CPU where it still lives
  - Possibly slightly better locality than the current wakeup code?
- If something else needs to run on the CPU
  - Use existing scheduler code to fire preempt_notifier
  - Save FPU/extended state
  - Switch to new task

Lazy Context Switching Tradeoffs

- Advantages:
  - Avoids expensive FPU/extended state store if CPU stays idle
  - Simple infrastructure
  - Multiple use cases
  - If the host has something else to run, it can run now
- Disadvantages:
  - Breaks Unix paradigm that the idle task is always PID 0
  - Adds a little bit of code to scheduler
  - Expensive FPU store when something else needs the CPU, instead of doing the store while nothing wants to run
- Needs some heuristic to avoid the downside? Time will tell

Conclusions

- Dealing with message passing workloads is hard
  - Hardware is optimized for being busy, or for being idle
  - Not for continuously switching between the two
- Paravirt C-state Driver
  - Isolated source code changes
  - Not clear how to avoid fundamental issues (solvable?)
- Lazy FPU switching
  - Special-purpose modifications to FPU code
  - Potentially a bad worst case, with IPIs (solvable?)
- Lazy Context Switching
  - Useful for multiple things, fewest potential downsides
  - Requires some changes to scheduler code
- Stay tuned for “exciting” patches...

Questions?

Suggestions?

Opinions?