Proposal: Supervisor Timer CSR and Virtual Supervisor Timer CSR

Version: 1.1
Authors: Siqi Zhao <zhaosiqi3@huawei.com> Yifei Jiang <jiangyifei@huawei.com>

1. Motivation

The current RISC-V specification defines only one clock event source, namely, the mtimecmp CSR. While this approach provides a simple solution, from the perspective of performance, there is ample room for improvement.

Whenever the HS-mode software needs to setup a timer, a trap is taken to M-mode. When VS-mode software needs to setup a timer, a trap is taken to HS-mode, and HS-mode software would emulate the M-mode CSR for VS-mode, which causes a second trap to M-mode.

While the trap from HS-mode might be acceptable, it is unlikely that any virtualization solution based on the current design is going to be performant. Timer maintenance is a frequent operation by any system software, multiple traps on its code path likely impact performance.

In this proposal, we propose to provide dedicated timer CSRs for HS-mode and VS-mode, to eliminate the traps incurred when setting timers.

2. New CSRs

2.1 CSR stimecmp for S-mode/HS-mode

The CSR stimecmp is a HSXLEN-bit read/write register. It is a dedicated timer CSR for S-mode/HS-mode.

Access to stimecmp does not trap when the hart is in S-mode/HS-mode.

The CSR address is to be defined as following.

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>[11:10]</th>
<th>[9:8]</th>
<th>[7:0]</th>
<th>Accessible Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>stimecmp</td>
<td>TBD</td>
<td>TBD</td>
<td>TBD</td>
<td>TBD</td>
<td>S-mode, M-mode</td>
</tr>
</tbody>
</table>

The stimecmp CSR is a read/write CSR accessible to S-mode and M-mode.
The stimecmp CSR is a per-hart CSR, i.e. each hart is equipped with its own such CSR.

2.2 CSR vstimecmp for VS-mode

The CSR vstimecmp is a HSXLEN-bit read/write register. It is a dedicated timer CSR for VS-mode.

When V==1, stimecmp is an alias of vstimecmp, i.e. instructions that access stimecmp when V==0 access vstimecmp instead.

Access to vstimecmp does not trap when V==1, nor when in HS-mode.

The CSR address is to be defined as following.

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>[11:10]</th>
<th>[9:8]</th>
<th>[7:0]</th>
<th>Accessible Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>vstimecmp</td>
<td>TBD</td>
<td>TBD</td>
<td>TBD</td>
<td>TBD</td>
<td>HS-mode, M-mode</td>
</tr>
</tbody>
</table>

The CSR is read/write accessible to HS-mode and M-mode.
The vstimecmp CSR is a per-hart CSR, i.e. each hart is equipped with its own such CSR.
3. Changes to Existing Hypervisor Extension CSRs

3.1 An Analysis of the Existing Design

The current RISC-V specification states that the vsie.STIE is an alias of hie.VSTIE. This design will not work when there is a separate timer CSR for VS-mode. Consider the following scenario:

A vCPU is running and its timer triggers an interrupt, the hip.VSTIP is set as a result of the timer interrupt. Since this interrupt generates a trap, the hypervisor starts to execute. The typical handling steps involve:

1. Enable global interrupt for the host
2. Take the interrupt.
3. Clear the hie.VSTIE, since the timer interrupt is level triggered
4. Return from interrupt handling.
5. Disable the global interrupt for the host
6. Finally, return to the vCPU.

However, step 3 causes an issue for the guest after returning, because now the guest's interrupt enabling state, i.e. vsie.STIE, is cleared by the hypervisor. Because vsie.STIE is an alias of hie.VSTIE. The hypervisor's clearing of hie.VSTIE clears vsie.STIE, too.

In an attempt to avoid this, the hypervisor can save the value of vsie.STIE on the trap from V=1, before it enables global interrupt. The hypervisor restores the value of vsie.STIE before returning to the vCPU, but after it disables global interrupt (to prevent timer interrupt caused by the value in the vstimecmp in the past). The above steps become (step 1 and step 7 are new):

1. Save vsie.STIE
2. Enable global interrupt for the host
3. Take the interrupt.
4. Clear the hie.VSTIE, since the timer interrupt is level triggered
5. Return from interrupt handling.
6. Disable the global interrupt for the host
7. Restore vsie.STIE
8. Finally, return to the vCPU.

Still, although the above software trick allows the hypervisor to avoid affecting the timer-enabling bit of the guest, there will be an issue. After the vCPU starts to run, there will immediately be a timer interrupt, and the guest's interrupt handler starts to run. The interrupt handler needs to clear vsie.STIE. However, if there is an exception before this instruction is executed, such as a guest page fault because the interrupt handler code is on a page that was swapped out, or an external interrupt arrives, a trap happens. On taking the trap, because the vsie.STIE, i.e. the hie.VSTIE, is set (otherwise the guest wouldn't start to execute the timer interrupt handler), and the value in the vstimecmp is in the past, as soon as the host enables global interrupt, there will be a timer interrupt again, resulting in double handling of the same interrupt.

To address the double handling, one can attempt to implement the hypervisor in a way that it always clears hie.VSTIE on taking a trap from V=1. However, this makes vCPU blocking complex. Because the hypervisor cannot take timer interrupt, it has to rely on the host's timer facility, e.g. the hrtimer, to manage the timer of the guest during the spinning of vCPU blocking. It also introduces more delay for the timer interrupt. If the hypervisor uses its own timer to track the guest timer, it needs to cancel its own timer and restore vstimecmp before returning to the vCPU. Between this operation and the time of returning to the vCPU, the hypervisor is not responding to any VS-mode timer interrupt, a fired timer during this will result in an immediate trap after the return to vCPU.
Still in an attempt to address the double handling, one can attempt to make the hypervisor record the fact that it has injected a timer interrupt by setting a flag, e.g. \texttt{vcpu->timer_injected}. It can set this flag after clearing \texttt{hie.VSTIE}. So if a trap happens after the guest starts to handle the timer interrupt but before the guest clears \texttt{vsie.STIE}, the hypervisor can still clear \texttt{hie.VSTIE} because \texttt{vcpu->timer_injected} is set. However, the problem is that this flag will not be cleared when the guest finishes the handling and sets \texttt{vsie.STIE}. On any trap after that guest sets \texttt{vsie.STIE}, the hypervisor always will clear \texttt{hie.VSTIE}, resulting in the same situation as that the hypervisor always clears \texttt{vsie.VSITE}. Note that the key here is that the hypervisor cannot be sure about the state of the timer handling code by reading the value of \texttt{vsie.STIE} even if it saves the value of \texttt{vsie.STIE} immediately after the trap. That is to say, during the handling of the timer interrupt in the guest, the \texttt{vsie.STIE} can be either cleared or set. The hypervisor cannot even be sure whether it should clear \texttt{vcpu->timer_injected}.

### 3.2 The Proposed Change

So the proposed change is to separate the \texttt{vsie} CSR and the \texttt{hie} CSR. When \texttt{V=0}, \texttt{hie.VSTIE} controls whether VS-mode timer interrupt is enabled. When \texttt{V=1}, this is done by \texttt{vsie.STIE}. Also, whenever the \texttt{vCPU} sets \texttt{vsie.STIE}, the hardware sets \texttt{hie.VSTIE}.

The VS-mode timer interrupt always go to \texttt{hip.VSTIP}. The hypervisor does not write to \texttt{hvip.VSTIP} to inject the interrupt. The \texttt{vCPU} indicates whether it can accept timer interrupt by setting / clearing \texttt{vsie.STIE}. In this design, the software flow goes as follows:

**Scenario 1** (VS-mode timer interrupt triggered during vCPU execution):

1. When \texttt{V=1}, the timer sets \texttt{hip.VSTIP}, since \texttt{hie.VSTIE = 1}, a trap is taken by the hart.
2. The hypervisor starts to execute.
3. As soon as the hypervisor enables global interrupt, it takes the VS-mode timer interrupt.
4. Hypervisor takes the the trap, starts to handle the interrupt
5. Hypervisor clear \texttt{hie.VSTIE} since this is level-triggered.
6. Hypervisor returns from interrupt handling.
7. Hypervisor returns to \texttt{vCPU}.
8. if \texttt{vsie.STIE = 1}, \texttt{vCPU} starts to handle the interrupt, it clears \texttt{vsie.STIE} first.
9. \texttt{vCPU} updates \texttt{stimecmp} (which is \texttt{vstimecmp} in fact), timer hardware clears \texttt{hip.VSTIP}
10. \texttt{vCPU} sets \texttt{vsie.STIE}, enabling VS-mode timer interrupt for \texttt{V=1}, hardware automatically sets \texttt{hie.VSTIE}, enabling the interrupt for \texttt{V=0}.

**Scenario 2** (VS-mode timer interrupt triggered when scheduling vCPU on a pCPU):

1. When \texttt{V=0}, during hypervisor's restoring vCPU context, e.g. \texttt{kvm_vcpu_load()}, after the \texttt{vstimecmp CSR} is restored and before the host disables interrupt right before returning to vCPU, a VS-mode timer interrupt is triggered, \texttt{hip.VSTIP} is set.
2. Hypervisor takes the trap, starts to handle the interrupt
3. Hypervisor clear \texttt{hie.VSTIE} since this is level-triggered.
4. Hypervisor returns from interrupt handling.
5. Hypervisor returns to \texttt{vCPU}.
6. if \texttt{vsie.STIE = 1}, \texttt{vCPU} starts to handle the interrupt, it clears \texttt{vsie.STIE} first.
7. \texttt{vCPU} updates \texttt{stimecmp} (which is \texttt{vstimecmp} in fact), timer hardware clears \texttt{hip.VSTIP}
8. \texttt{vCPU} sets \texttt{vsie.STIE}, enabling VS-mode timer interrupt for \texttt{V=1}. Hardware automatically sets \texttt{hie.VSTIE}, enabling the interrupt for \texttt{V=0}.

**Scenario 3** (VS-mode timer interrupt triggered when scheduling vCPU off a pCPU):
1. During hypervisor’s saving vCPU context, e.g. kvm_vcpu_put(), after the host enables interrupt and before it saves the value of vstimecmp, a VS-mode timer interrupt is triggered, hip.VSTIP is set.

2. Hypervisor takes the trap, starts to handle the interrupt

3. Hypervisor clear hie.VSTIE since this is level-triggered.

4. Hypervisor returns from interrupt handling.

5. Hypervisor saves hie, vstimecmp, hip, continues to schedule the vCPU off the pCPU.

6. Later the hypervisor schedules the vCPU on a pCPU, it runs code, e.g. kvm_vcpu_load(), which restores hie, vstimecmp, hip. Since hie.VSTIE is cleared, a value in vstimecmp in the past does not trigger a VS-mode timer interrupt again.

7. Since the value in vstimecmp is in the past, hip.VSTIP is set by timer hardware.

8. The hypervisor returns to the vCPU.

9. The vCPU starts to handle the timer interrupt, it clears vsie.STIE first.

10. vCPU updates stimecmp (which is vstimecmp in fact), timer hardware clears hip.VSTIP.

11. vCPU sets vsie.STIE, enabling VS-mode timer interrupt for V=1. Hardware automatically sets hie.VSTIE, enabling the interrupt for V=0.

4. Future: Improving Interrupt Handling

It is possible to further consider improving the delivery of the interrupts generated by the proposed timer CSR for VS-mode. When a timer interrupt is triggered, if the vCPU is currently running, the interrupt can be directly delivered to the vCPU, bypassing any processing by the host. The above changes to the CSRs are still needed. The reasoning is as following. Consider the following scenario. When V=0 and a timer interrupt is triggered by vstimercmp CSR. The hypervisor receives the interrupt, and immediately sets hie.VSTIE = 0 since this is still a level-triggered interrupt. Now if hie.VSITE and vsie.STIE are the same bit, the hypervisor will clear the vsie.STIE bit, same as analyzed above. The interrupt will never be delivered to the vCPU.