InterSloth: Global Hardware-Based Scheduling in a MultiCore-RTOS on RISC-V

Christian Dietrich

2019-09-24
Fixed-Priority Real-Time Scheduling

- Schedule Threads according to a Fixed Priority onto Processor(s)
  - Optimal priority assignments for unicore ({rate,deadline} monotonic)
  - For multicore: Global fixed-priority is most flexible schema

Real-time analysts would like a zero RTOS overhead, but...
- Global scheduling requires global synchronization (locking)
- Dispatch on different CPU requires inter-processor interrupt
Fixed-Priority Real-Time Scheduling

- Schedule Threads according to a Fixed Priority onto Processor(s)
  - Optimal priority assignments for unicore (\{rate,deadline\} monotonic)
  - For multicore: Global fixed-priority is most flexible schema

- Real-time analysts would like a zero RTOS overhead, but...
  - Global scheduling requires global synchronization (locking)
  - Dispatch on different CPU requires inter-processor interrupt
Fixed-Priority Real-Time Scheduling

- Schedule Threads according to a Fixed Priority onto Processor(s)
  - Optimal priority assignments for unicore (\{rate, deadline\} monotonic)
  - For multicore: Global fixed-priority is most flexible schema

Real-time analysts would like a zero RTOS overhead, but...
- Global scheduling requires global synchronization (locking)
- Dispatch on different CPU requires inter-processor interrupt
Fixed-Priority Real-Time Scheduling

- Schedule Threads according to a Fixed Priority onto Processor(s)
  - Optimal priority assignments for unicore \{{\text{rate},\text{deadline}}\} monotonic
  - For multicore: Global fixed-priority is most flexible schema

![Diagram of CPU scheduling with priorities]

- Real-time analysts would like a zero RTOS overhead, but...
  - Global scheduling requires global synchronization (locking)
  - Dispatch on different CPU requires inter-processor interrupt
Sloth: Threads as Interrupts

Sloth RTOS in a Nutshell: Threads \( \equiv \) ISR
- Interrupt controller already selects high-prio interrupt source.
- Interrupt service routine performs context switch between threads.
- Supports only unicore and partitioned multicore scheduling.
Sloth: Threads as Interrupts

Sloth RTOS in a Nutshell: Threads ≡ ISR
- Interrupt controller already selects high-prio interrupt source.
- Interrupt service routine performs context switch between threads.
- Supports only unicore and partitioned multicore scheduling.

C Code: 200–500 lines
ROM: 300–900 bytes
RAM: 0–20 bytes
Event latency: 12–60 cycles
Strict priority obedience
InterSloth: Extension to Global FP Scheduling

We require a strict priority-obedient IRQ controller, but existing... 
- ...use a threshold and choose at random (ARM)
- ...do not support re-delivery of IRQs (Intel)
- ...support only fixed CPU–IRQ mapping (Infineon AURIX)
We require a strict priority-obedient IRQ controller, but existing... 

- ...use a threshold and choose at random (ARM)
- ...do not support re-delivery of IRQs (Intel)
- ...support only fixed CPU–IRQ mapping (Infineon AURIX)

**MIRQ-V: A Strict Priority-Obedient Multi-Core IRQ Controller**
Motivation

*MIRQ-V: A Strict Priority-Obedient Multi-Core IRQ Controller*

InterSloth: An RTOS on Top of MIRQ-V

Evaluation

Conclusion
MIRQ-V: Features and Rocket Integration

- Feature set is designed for hard real-time systems
  - Freely configurable interrupt and CPU interfaces
  - Up to 255 interrupt/CPU priority levels
  - Highest-priority IRQ is always delivered to lowest-priority CPU
  - Software-triggered IRQ sources and IRQ migration

- Integration with a RISC-V processor
  - Rocket Chip Generator is written in Chisel HDL (Scala DSL)
  - MIRQ-V replaces the Platform-Level Interrupt Controller (PLIC)
  - Existing prototype and work on a more efficient implementation
Example Interrupt Delivery with Software IRQ

CPU #1

MIRQ-V
Priority(CPU #1) = 1
Priority(CPU #2) = 3

CPU #2

IRQ 1 / Priority = 2
trigger()
cpu = findMin(CPUs)
if cpu.prio < irq.prio
    trigger(cpu)
irq = claim()
/Lightning
isr_fns[irq]()
claim()
irq = 1
complete(irq)
Example Interrupt Delivery with Software IRQ

CPU #1

MIRQ-V
Priority(CPU #1) = 1
Priority(CPU #2) = 3

CPU #2

IRQ 1 / Priority = 2
trigger(

cpu = findMin(CPUs)
if cpu.prio < irq.prio
trigger(cpu)

irq = claim()
/Lightning
isr_fns[irq]()
claim()
irq = 1
complete(irq)
complete()
Example Interrupt Delivery with Software IRQ

CPU #1
MIRQ-V
Priority(CPU #1) = 1
Priority(CPU #2) = 3
CPU #2
IRQ 1 / Priority = 2

trigger()
cpu = findMin(CPUs)
if cpu.prio < irq.prio
trigger(cpu)
irq = claim()

/Lightning
isr_fns[irq]()
claim()
irq = 1

complete(irq)
Example Interrupt Delivery with Software IRQ

CPU #1

MIRQ-V
Priority(CPU #1) = 1
Priority(CPU #2) = 3

CPU #2

IRQ 1 / Priority = 2
trigger()
cpu = findMin(CPUs)
if cpu.prio < irq.prio
trigger(cpu)
irq = claim()

/Lightning
isr_fns[irq]()
claim()
irq = 1
complete(irq)

complete()
Example Interrupt Delivery with Software IRQ

CPU #1

MIRQ-V
Priority(CPU #1) = 1
Priority(CPU #2) = 3

CPU #2

IRQ 1 / Priority = 2
trigger()
cpu = findMin(CPUs)
if cpu.prio < irq.prio
trigger(cpu)
irq = claim()
/Lightning
isr_fns[irq]()
claim()
irq = 1
complete(irq)
Example Interrupt Delivery with Software IRQ

CPU #1
MIRQ-V
Priority(CPU #1) = 2
Priority(CPU #2) = 3
CPU #2
IRQ 1 / Priority = 2
trigger()
cpu = findMin(CPUs)
if cpu.prio < irq.prio
trigger(cpu)
irq = claim()
/Lightning
isr_fns[irq]()
claim()
irq = 1
complete(irq)
complete()
Example Interrupt Delivery with Software IRQ

CPU #1

MIRQ-V
Priority(CPU #1) = 2
Priority(CPU #2) = 3

CPU #2

IRQ 1 / Priority = 2
trigger()
cpu = findMin(CPUs)
if cpu.prio < irq.prio
trigger(cpu)
irq = claim()
/Lightning
isr_fns[irq]()
claim()
irq = 1
complete(irq)
Example Interrupt Delivery with Software IRQ

CPU #1

MIRQ-V
Priority(CPU #1) = 2
Priority(CPU #2) = 3

CPU #2

IRQ 1 / Priority = 2
trigger()
cpu = findMin(CPUs)
if cpu.prio < irq.prio
trigger(cpu)
irq = claim()
/Lightning
isr_fns[irq]()
claim()
irq = 1
complete(irq)
complete()
Example Interrupt Delivery with Software IRQ

CPU #1

MIRQ-V
Priority(CPU #1) = 2
Priority(CPU #2) = 3

CPU #2

IRQ 1 / Priority = 2
trigger()
cpu = findMin(CPUs)
if cpu.prio < irq.prio
trigger(cpu)
irq = claim()
/Lightning
isr_fns[irq]()
claim()
irq = 1
complete(irq)
complete()
Example Interrupt Delivery with Software IRQ

CPU #1

MIRQ-V
Priority(CPU #1) = 2
Priority(CPU #2) = 3

CPU #2

IRQ 1 / Priority = 2
trigger()
cpu = findMin(CPUs)
if cpu.prio < irq.prio
trigger(cpu)
irq = claim()
/Lightning
isr_fns[irq]()
claim()
irq = 1
complete(irq)
complete()
MIRQ-V: A few technical details

- Race conditions between delivery and CPU-priority changes
  - Normal PLIC IRQ Source has two states: *pending* and *in service*
  - Introduce *delivered* if CPU is informed but has not *claim()*ed.
  - Automatic re-delivery of delivered IRQs if a CPU priority changes.

```plaintext
CPU #0

MIRQ-V
Priority(CPU #1) = 1
Priority(CPU #2) = 3

CPU #1

irq = claim()

claim()

trigger()

IRQ 1 / Priority = 2

cpu = findMin(CPUs)
if cpu.prio < irq.prio
    trigger(cpu)
```
MIRQ-V: A few technical details

- Race conditions between delivery and CPU-priority changes
  - Normal PLIC IRQ Source has two states: *pending* and *in service*
  - Introduce *delivered* if CPU is informed but has not *claim()*ed.
  - Automatic re-delivery of delivered IRQs if a CPU priority changes.

- Backward Compatibility with the original PLIC
  - *migrate()* IRQ to other CPU, *trigger()* from software.
  - Encode new commands into *claim/completed-*register values

<table>
<thead>
<tr>
<th>Command</th>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>complete()</td>
<td>00</td>
<td>IRQ</td>
</tr>
<tr>
<td>migrate()</td>
<td>01</td>
<td>IRQ</td>
</tr>
<tr>
<td>trigger()</td>
<td>10</td>
<td>IRQ</td>
</tr>
</tbody>
</table>
Motivation

MIRQ-V: A Strict Priority-Obedient Multi-Core IRQ Controller

InterSloth: An RTOS on Top of MIRQ-V

Evaluation

Conclusion
InterSloth: An RTOS for MIRQ-V

- MIRQ-V already performs most of the heavy lifting
  - No global synchronization need, as MIRQ-V is single source of truth.
  - Scheduling and re-scheduling decisions are calculated in parallel.
  - The CPU with the lowest priority is informed about high-priority IRQ.

- InterSloth must handle preemption of already running ISRs

**Remember**: Sloth $\Rightarrow$ Thread $\equiv$ ISR

- Preemption can happen at any time and must be performed transparent.
- ISR prologue always safes old thread context.
- Start new thread or resume to old thread context.
Example of Thread Activation and Preemption

CPU #0
Task 1
Activate-
Task(Task2)
CPU #1
Task 2
/Lightning
Activate-
Task(Task3)
/Lightning
T
save_ctx(Task1);
migrate(IRQ1);
dispatch(Task3);
Task 3
TerminateTask()
/Lightning
T
restore_ctx(Task1);
dispatch(Task1);
Task 1

save_ctx(Task1);
migrate(IRQ1);
dispatch(Task3);
Task 3
TerminateTask()
restore_ctx(Task1);
dispatch(Task1);
Motivation

MIRQ-V: A Strict Priority-Obedient Multi-Core IRQ Controller

InterSloth: An RTOS on Top of MIRQ-V

Evaluation

Conclusion
Evaluation: MIRQ-V

- IRQ arbitration in 1 cycle, delivery in 4 cycles

- MIRQ-V Design with parametrizable number of IRQs/CPUs
  - LUT demand is linearly larger than PLIC LUT demand
  - Full Rocket Chip uses 23,000 LUTs

![Graph showing LUT requirements for PLIC and MIRQ-V with different interrupt sources](image-url)
Evaluation: InterSloth

- Cycle-Accurate Simulator on Verilog level (Verilator)
  - Timing Measurements from Software with RISC-V’s `mcycle` register
  - Measure time that is spent in the InterSloth kernel

- Measure three characteristic InterSloth system calls
  - Context switch must save and restore 32 general-purpose registers.

<table>
<thead>
<tr>
<th>System Call</th>
<th>Cycles</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActivateTask()</td>
<td>318</td>
<td>130</td>
</tr>
<tr>
<td>TerminateTask()</td>
<td>93</td>
<td>60</td>
</tr>
<tr>
<td>ChainTask()</td>
<td>261</td>
<td>148</td>
</tr>
</tbody>
</table>

*Trigger thread and dispatch on different CPU*

*Start the idle thread*

*Destroy current thread and dispatch in this CPU*
Conclusion

Problems for Multi-Core Real-Time Scheduling
- Overheads for Inter-Core Communication and Synchronization
- Multi-Core IRQ Controllers are not strictly enforcing priorities.

MIRQ-V: A Strict Priority-Obedient Multi-Core IRQ Controller
- Always route highest-priority IRQ to lowest-priority CPU.
- Parametrizable Hardware Design with Rocket (RISC-V) Integration
- Software-Triggered IRQs and IRQ Migration

InterSloth: Minimal-Effort Global Fixed-Priority Scheduling
- ISRs save the old and restore/start the new thread context.
- Requires no global synchronization between CPUs

Future Work
- MIRQ-V: Decrease LUT Demand by Optimized Delivery Invalidation
- InterSloth: Support more RTOS Primitives (Mutexes, Alarms, Events)


