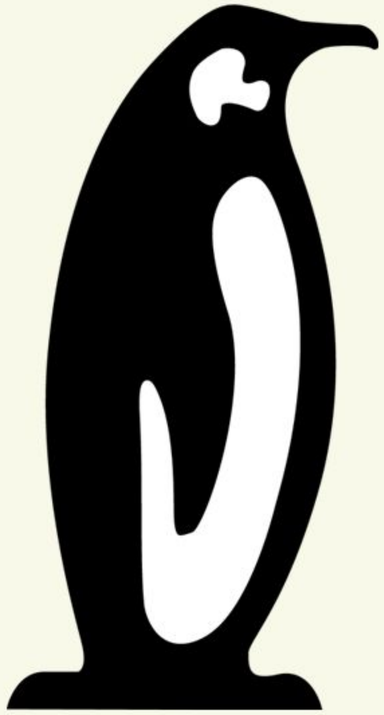


Linux Plumbers Conference

Richmond, Virginia | November 13-15, 2023

Linux
Plumbers
Conference | Richmond, VA | Nov. 13-15, 2023

RISC-V Vector: Current and Next?

RISC-V Vector - Overview

Current status

- Supports running Vector in user-mode:
  - Context save/restore happen in `switch_to()`
- Supports ptrace, signal interfaces.
- Vector unit is disabled by default. Processes take a trap at its first-Use, then the kernel decides whether does it allocate, track Vector context according to a configurable policy.

In Development

- Kernel-Mode Vector:
  - Supports running kernel-mode Vector between kernel_vector_{begin, end} APIs with preemption on/off.
  - Restores user's Vector context only when returning to user
  - Includes vectorized subroutines (**planned for v4**)
    - memcpy/memset/memmove
    - copy_{to, from}_user with scalar fixup

RISC-V Kernel Mode Vector - Challenges

Blindly calling vectorized functions would not lead to optimal performance:

- Vector (SIMD) is not always better than scalar. There are things needed to be taken into accounts, when comparing scalar with vector:
  - Hardware performance catachrestic
  - Initial, Tear-down cost of using Vector (detour)
  - Context switches (preemptible Vector)

(mem*/copy_*_user)	hackbench (sec)	Netperf (Mb/s)
Scalar	1.279	6047.93
Use Vector, unconditionally	1.478 (-15.6%)	9361.85 (+54.79%)

Details:

- ran on a downstream kernel (v6.2) with the v3 patch series + vectorized mem*/copy*user + some context switch instrument code
- FPGA simulates a SiFive core @ 2 GHz with 85 ns LLC miss panalties

RISC-V Kernel Mode Vector - Try Optimizing kernel_vector_begin

Implementation Detail of the Kernel-mode Vector:

- Check if current context is able to use Vector:
  - `!in_hardirq()`
- Save once and defer restore for user's V context
  - `riscv_v_vstate_save`
  - `riscv_v_vstate_set_restore`
- Disable preemption and flip on Vector Unit:
  - `__this_cpu_xchg(vector_context_busy, true);`
  - `riscv_v_enable();`

RISC-V Kernel Mode Vector - Try Optimizing kernel_vector_begin

Implementation Detail of the Kernel-mode Vector:

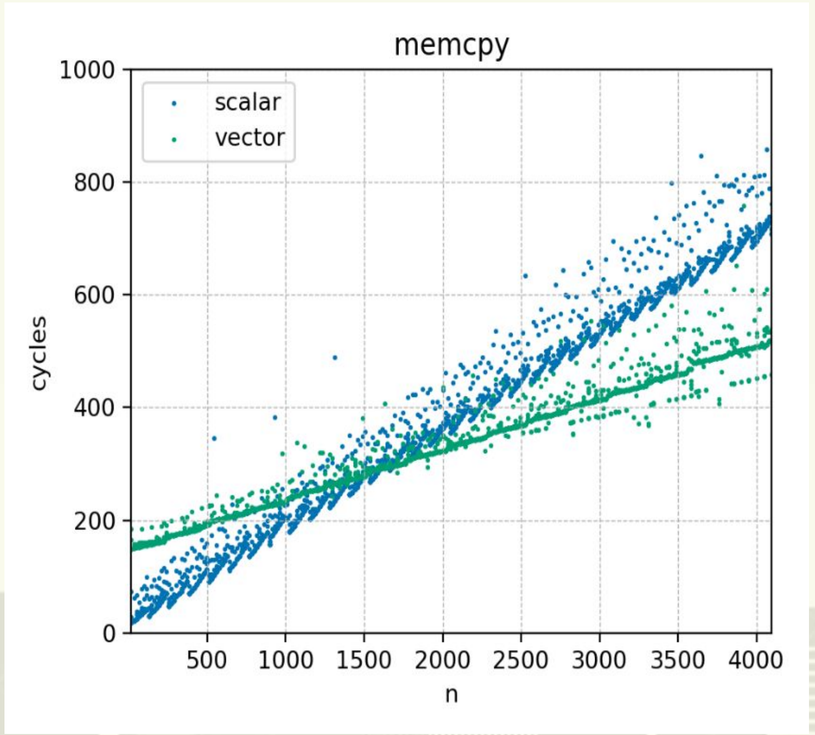
- Check if current context is able to use Vector:
  - `!in_hardirq()`
- Save once and defer restore for user's V context
  - `riscv_v_vstate_save`
  - `riscv_v_vstate_set_restore`
- Disable preemption and flip on Vector Unit:
  - `__this_cpu_xchg(vector_context_busy, true);`
  - `riscv_v_enable();`

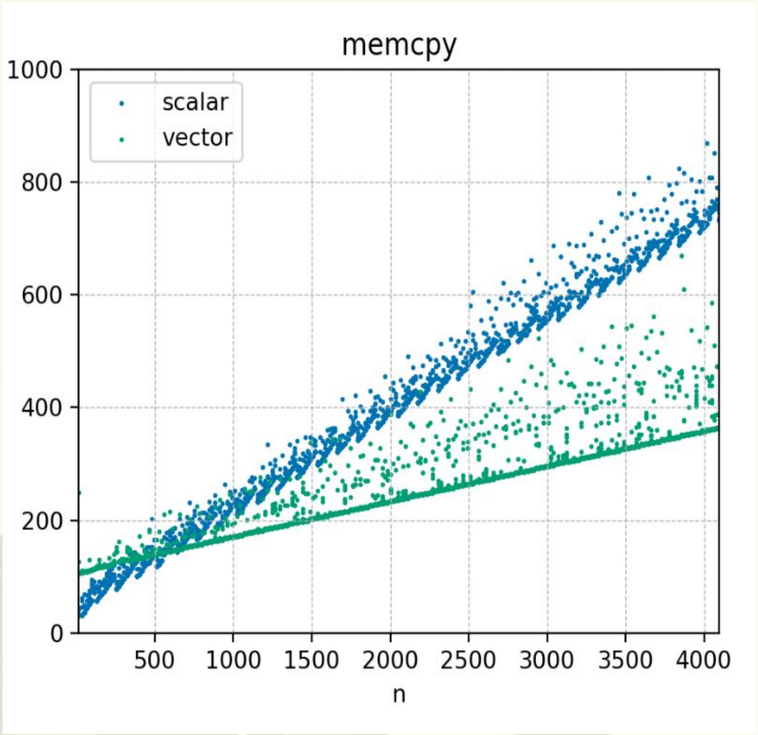
Approaches

- Check if current context is able to use Vector:
  - `test_thread_flag()`
- Save once and defer restore for user's V context
  - `riscv_v_vstate_save`
  - `riscv_v_vstate_set_restore`
- Disable preemption and flip on Thread flag:
  - `test_and_set_thread_flag()`
  - Enable Vector for kernel at boot and trap entry

Performance of memset/memcpy/memmove on SiFive Vector core (FPGA):

v3 versus inline + optimized kernel_vector_{begin, end}: remove pcpu, redundant checks, and enable V





Details:

Simulate frequency of 2GHz,

and 85ns LLC miss latency.

VLEN = 128b

$$T_{vec} = M_{vec} \cdot n + C_{vec}$$

Performance of memset/memcpy/memmove on SiFive core (FPGA):

Benchmarking results are not sensitive to these changes:

- When sizes are large, then the constant cost is neglectable
- When sizes are small, the cost become more significant.
- However, minimizing the constant cost just makes things "less worse" but not "better"

v3 series	hackbench (sec)	Netperf (Mb/s)
Scalar	1.279	6047.93
Use Vector, unconditionally	1.478 (-15.6%)	9361.85 (+54.79%)
Use Vector for optimal size	1.288 (+0.7%)	9494 (+56.98%)

v3, optimized kernel_vector_*	hackbench (sec)	Netperf (Mb/s)
Scalar	1.302	6118.63
Use Vector, unconditionally	1.466(-12.6%)	9398.87 (+53.61%)
Use Vector for optimal size	1.307 (-0.38%)	9487.52 (+55.06%)

RISC-V Kernel-Mode Vector - Proposal

Proposal:

- Enable Vector for larger operation sizes only (e.g. > 1K)
  - A DT interface for venders to give optimal values

Any of these?

- Check if current context is able to use Vector:
  - `test_thread_flag()`
- Save once and defer restore for user's V context
  - `riscv_v_vstate_save`
  - `riscv_v_vstate_set_restore`
- Disable preemption and flip on Thread flag:
  - `test_and_set_thread_flag()`
    - Enable Vector for kernel at boot and trap entry

RISC-V Kernel-Mode Vector - Softirq

Some Vector-intensive user runs in softirq:

```
commit 13150149aa6ded1e6bbe0025beac6e12604dd87c
Author: Ard Biesheuvel <ardb@kernel.org>
Date:   Tue Mar 2 10:01:12 2021 +0100

    arm64: fpsimd: run kernel mode NEON with softirqs disabled

    Kernel mode NEON can be used in task or softirq context, but only in
    a non-nesting manner, i.e., softirq context is only permitted if the
    interrupt was not taken at a point where the kernel was using the NEON
    in task context.

    This means all users of kernel mode NEON have to be aware of this
    limitation, and either need to provide scalar fallbacks that may be much
    slower (up to 20x for AES instructions) and potentially less safe, or
    use an asynchronous interface that defers processing to a later time
    when the NEON is guaranteed to be available.
```

Potential Solutions:

- Context nesting, when in softirq():
  - kernel_vector_begin()
    - save active kernel V context to pcpu or kernel_vstate,
  - kernel_vector_end()
    - mark the context for trap return
  - Trap return:
    - Restore V context
- Or, just disable bottom halves

Preemptible Vector - Functional Considerations

Reasons:

- Long running SIMD, with preemption disabled, could hit system responsiveness.
  - Large copies
  - A kernel thread runs with auto-vectorization?
  - copy_*_user without a scalar fallback on faults
- Yielding Vector unit might not intuitive to programmers

How-to:

- Add a kernel Vector context
- Do V-context tracking at trap handler
- Deal with the right context in contex switches

API-Design:

- **Specialized or generic API?**
  - `kernel_vector_begin_preemptible()`
- **Should we allow calling `schedule()` in preemptible-V?**
  - By tracking sstatus.VS
- **Do we pre-allocate kernel mode Vector context?**
  - This is to prevent check fails in preemptible rcu

Preemptible Vector - Performance Considerations

Cost of a Vector context switch

- Time slice after each context switch: 0.75 ms
  - 1500000 cycles on 2GHz

- Cost of a context switch:

VLEN=128	Cycle Counts	Number of Samples
save	70.47	224619
restore	92.3	3820579

Expected Value:

$$E_{vec}(n) = \frac{T_{vec(n)}}{T_{slice}} \cdot (T_{vec}(n) + T_{vec\_save} + T_{vec\_restoe}) + (1 - \frac{T_{vec}(n)}{T_{slice}}) \cdot T_{vec}(n)$$

$$E_{scalar}(n) = T_{scalar}(n)$$

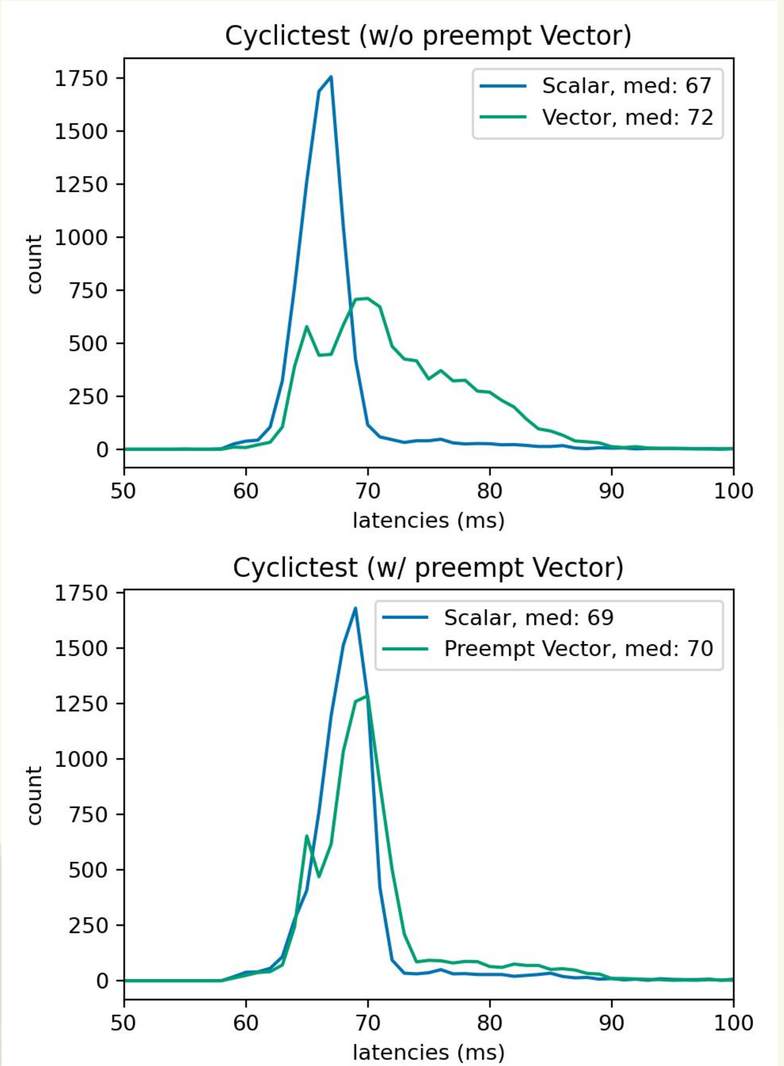
- The cost of starting/ending Vector context should be included in the vector execution time
- If a context switch must happen during Vector execution, then
  - T(n) + Tsave + Trestore < Tscalar is a gain. (n >= 2K for memcpy)

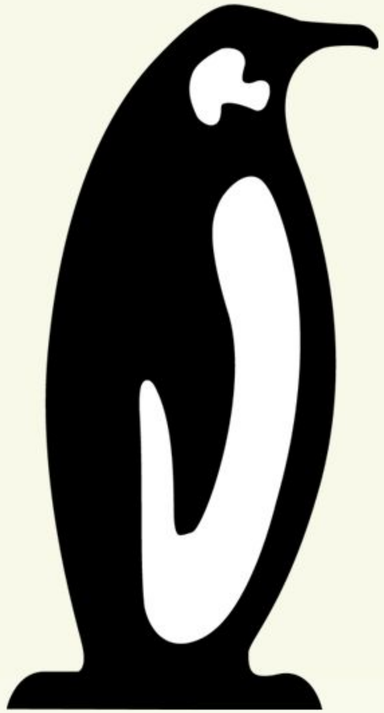
Preemptible Vector - Performance Considerations

Experiment Details:

- Each cyclictest runs with 12 netperf processes in the background
- All kernel-mode Vector context are preallocated at processes' start time for the preempt Vector part

Netperf total bw (Mb/s)	without preempt Vector	with preempt Vector
Scalar	5458.93	5707.57
Vector	8568.05	8510.47





Linux Plumbers Conference

Richmond, Virginia | November 13-15, 2023