Threading on Eventdev

ERICSSON

Mattias Rönnblom Principal Software Developer, Ericsson AB 2026-05-13

Agenda

ERICSSON

- Eventdev?
- Eventdev Multithreading Rationale
- Implementation
- Benchmarks
- Limitations
- Questions

Eventdev

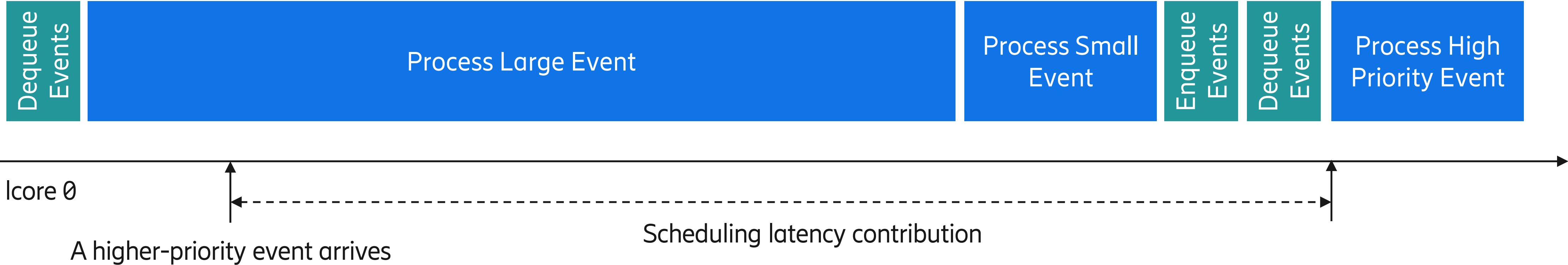
ERICSSON

- Data plane work scheduler
- DPDK API presenting a parallel event machine <rte_eventdev.h>
- Event = job
  - Run-to-completion – no preemption
  - Run by EAL threads (lcores)
  - Result is side effects, zero or more new jobs and/or packet TX
- All lcore work is the result of an event
- Job examples
  - Protocol processing for one or more packets
  - A timeout
  - A control plane request

Why Jobs in Eventdev Should Be Short

- The event device can‘t preempt a job running on an lcore
  - Switch from low to high priority work may have to wait
- Event devices tend to assume that jobs are small and about the same size

Scheduling Latency Example



When Short Jobs Are Hard

ERICSSON

- Continuation-dependent workflow
  - No natural way to break processing into bit-sized chunks
  - Complex to save and restore application state
- Operation requires external input to complete
  - Wants to block, can’t block
- Legacy threaded code

The Case for a Thread-like Construct

ERICSSON

- Run “threaded” (stackful) code on the same lcores as event-driven processing (e.g., packet handling)
- Allow for many “threads“
- “Threads” cheap to create and destroy, and to switch between

Eventdev Multithreading

libcoro

ERICSSON

- Coroutines for the data plane
- Stackful
- Cooperative multitasking
  - Explicit, voluntary yield
- Mempool for execution stack and input and output parameter memory
- Context switching assembly from Boost.Context (x86_64, ARM64, PPC, RISC-V)
- Prototype only

libcoro API

ERICSSON

Functions

__rte_experimental int	**rte_coro_init** (size_t stack_size, size_t max_in_param_size, size_t max_out_param_size, size_t max_coroutines, uint64_t default_time_slice, unsigned int mempool_cache_size, int socket_id)
__rte_experimental void	**rte_coro_cleanup** (void)
__rte_experimental struct rte_coro *	**rte_coro_create** (rte_coro_run_t run)
__rte_experimental void	**rte_coro_destroy** (struct rte_coro *coro)
__rte_experimental uint32_t	**rte_coro_id** (const struct rte_coro *coro)
__rte_experimental void	**rte_coro_resume** (struct rte_coro *coro)
__rte_experimental bool	**rte_coro_has_exited** (const struct rte_coro *coro)
__rte_experimental void *	**rte_coro_in_param** (struct rte_coro *coro, size_t size)
__rte_experimental void *	**rte_coro_out_param** (struct rte_coro *coro, size_t size)
__rte_experimental void	**rte_coro_set_time_slice** (struct rte_coro *coro, uint64_t time_slice)
__rte_experimental void	**rte_coro_yield** (void)
__rte_experimental void	**rte_coro_maybe_yield** (void)
__rte_experimental **__rte_noreturn** void	**rte_coro_exit** (void)
__rte_experimental uint32_t	**rte_coro_self_id** (void)
__rte_experimental void *	**rte_coro_self_in_param** (size_t size)
__rte_experimental void *	**rte_coro_self_out_param** (size_t size)
__rte_experimental struct rte_coro *	**rte_coro_self** (void)
__rte_experimental uint64_t	**rte_coro_resume_count** (struct rte_coro *coro)
__rte_experimental uint64_t	**rte_coro_maybe_yield_count** (struct rte_coro *coro)

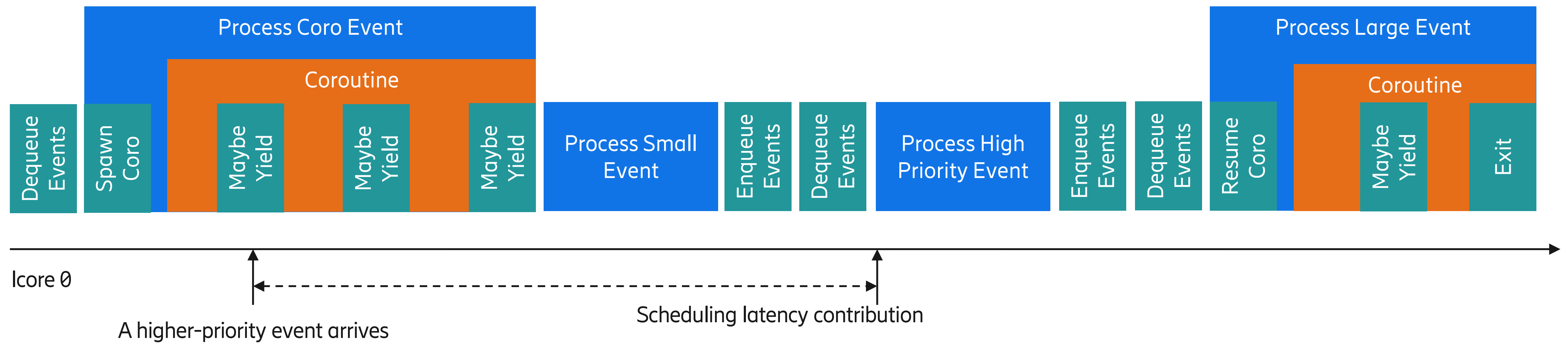
Eventdev Coroutine Integration

ERICSSON

- Upon dequeuing an event calling for a large operation, spawn a coroutine for it
- Run the first time slice of the coroutine
  - Occasionally calls rte_coro_maybe_yield()
- When the first time slice is up, the old (main) stack is switched back
- If coroutine is not completed, enqueue an event with a reference to the coroutine
  - Control passed back to the event device
- On a future dequeue of that event, resume processing of the coroutine
  - If more work is required, enqueue another event for a new resumption etc.

Eventdev + Coroutine Example Execution

ERICSSON



A coroutine may also be migrated across different lcores between invocations

Benchmarks

pi

ERICSSON

- Calculate pi many times, using two methods
  1. Loop with regular function calls
  2. Loop creating a coroutine each for every pi calculation
- Single lcore

```
matro@isengard$~/mc/pi> taskset -c 16 ./pi --no-huge -m 500M
EAL: Detected CPU lcores: 32
EAL: Detected NUMA nodes: 1
EAL: Detected static linkage of DPDK
EAL: Multi-process socket /run/user/1000/dpdk/rte/mp_socket
EAL: Selected IOVA mode 'VA'
Calls: 10000
Without coroutines:
  Total latency: 10.023 s
  Latency per call: 1002.306 us
Coroutines:
  Total latency: 10.083 s
  Latency per call (coroutine): 1008.345 us
  Coroutine overhead per call: 6.039 us
  Switches per call: 93.8
  Overhead per switch: 0.064 us
  Switch considerations per call: 300.0
  Configured time slice: 10.000 us
  Actual time slice: 10.751 us
  Coroutine overhead: 0.603%
matro@isengard$~/mc/pi>
```

Coroutine creation or switch overhead? Not an issue!

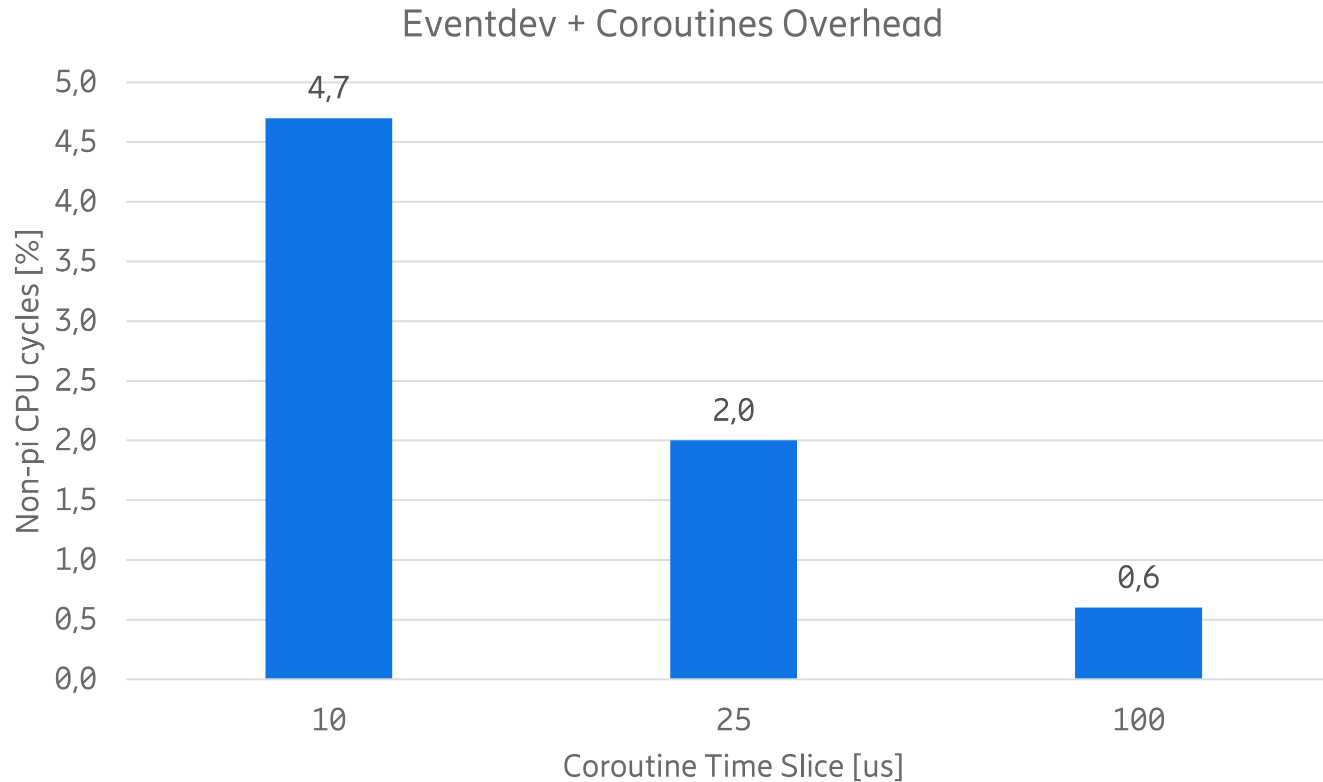
pitp

ERICSSON

- Calculates pi many times on many lcores
- Test is run in two ways
  - Without coroutines and with a low-overhead, static load balancing scheme
  - With coroutines on top of Eventdev

pitp: Time Slices and Overhead

ERICSSON



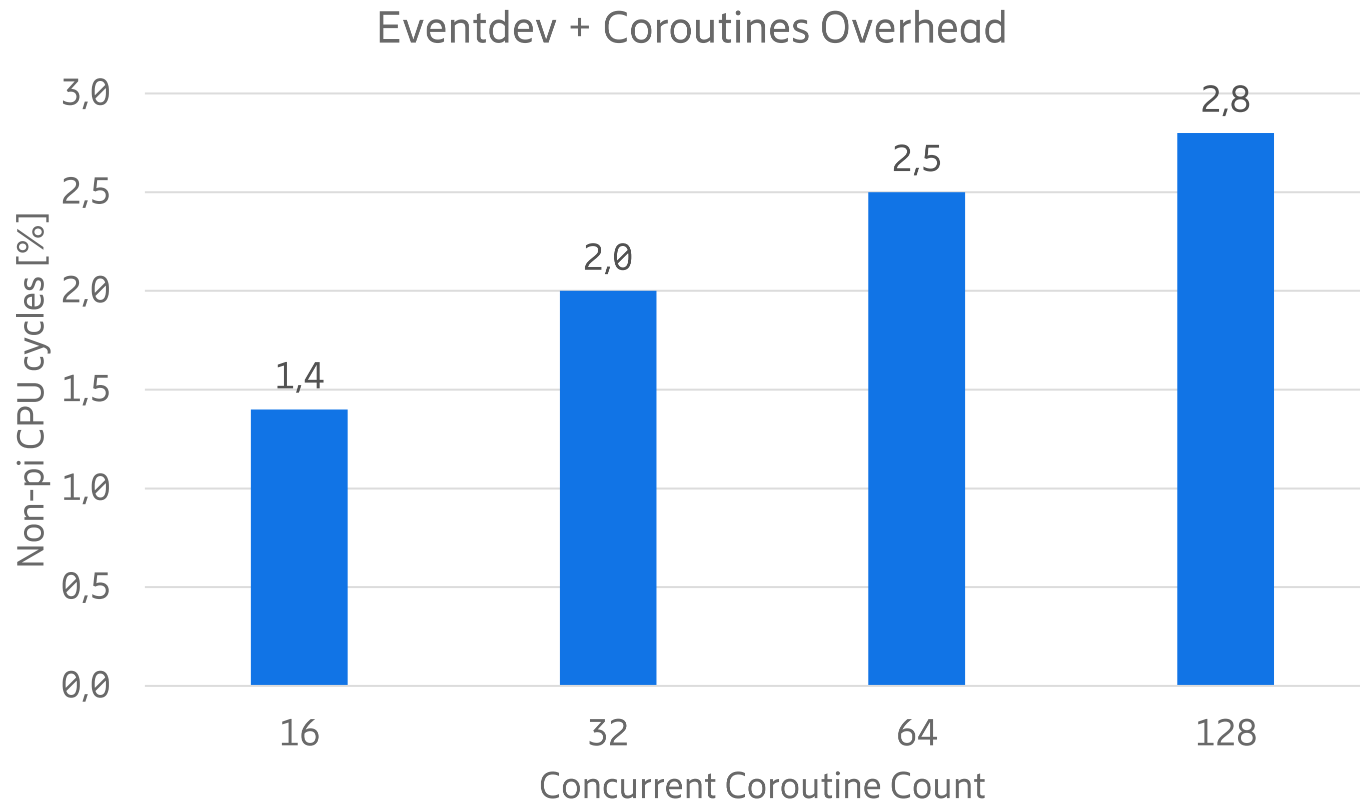
Overhead

- Event device enqueue/dequeue
- Idle cycles (non-optimal LB)
- Coroutine
- Test app

SUT Configuration
Rhodes version: v25
Rhodes scheduling policy: RT
DPDK: v24.11
CPU: Raptor Lake E-cores @ 2,8 GHz
Pi computation lcores: 16
Concurrent Coroutines: 32
Event dequeue/enqueue burst size: 1

pitp: Concurrent Coroutine Count and Overhead

ERICSSON



SUT Configuration
Time slice: 25 us

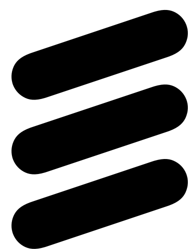
Limitations

ERICSSON

- No blocking syscalls (like always on an lcore thread)
- No yielding while holding a lock
- Use coroutine param memory instead of thread-local storage
- Execution stacks take up a lot of memory
- Application-level plumbing required to create/resume coroutines
  - Should be a platform service, and use libdispatcher

ERICSSON

Questions?



ERICSSON