Restartable Sequences: Scheduler-Aware Scaling of Memory Use on Many-Core Systems

Mathieu Desnoyers
EfficiOS Inc.

Outline

- RSEQ adoption status
- RSEQ next steps
- Per-memory-space virtual CPU ID RSEQ extension
- Scheduler context switch
- Benchmarks and schedstat profiling
- NUMA
- Discussion

RSEQ Adoption Status

- Architectures
  - ARM, MIPS, Power, Risc-V, s390, x86,
  - Also csky and loongarch
    - but merged upstream without user-space tests :-(
- GNU C library: rseq used since glibc-2.35
  - Used to implement sched_getcpu(3)
  - Other use-cases being discussed, e.g. memory allocator
- tcmalloc, CRIU, DynamoRIO

RSEQ Next Steps

- Use-cases: memory allocators, ring buffers, counters,
- Generally remove the need to configure user-space data structure partitioning based on the number of threads vs cores [1]:
  - Global (few threads),
  - Per-thread (nr_threads <= nr_cores),
  - Per-core (nr_threads >= nr_cores).
- Per-CPU data memory use on single-threaded processes.
- Per-CPU data memory use when using cpusets on many-cores systems.

EfficiOS

Per-Memory-Space Virtual CPU ID RSEQ Extension

- Idea originally from Paul Turner (Google), discussed with him at LPC2019.
- Allocate "virtual" CPU IDs within a process, which can be limited by the number threads running concurrently.
- The Google implementation was not publicly available, so I implemented it myself to see what I could come up with. [2]

Scheduler Context Switch

- Extend the scheduler to continuously track the number of threads concurrently running on behalf of each mm.
- When the scheduler switches to a thread, that thread is assigned a vcpu_id which is guaranteed to be unused by any other thread from the same memory space until the thread is scheduled out.
- This can be done with a per-mm bitmap (mm_vcpumask) bounded by the number of possible cpus on the system. Updates are atomic bit test-and-set and atomic bit clear.
- Additional atomic operations on scheduler context switch fast-path is frowned upon for good reasons.

Benchmarks (hackbench)

Scheduler overhead is significant for threaded workload without further optimization.

10 groups using 40 fd, each sender passes messages of 100 bytes, x86-64 E5-2630

- Per-process (10000 messages)
  - Baseline: 10.5±0.3 s
  - With mm vcpu_id: 10.6±0.4 s
- Per-thread (10000 messages)
  - Baseline: 15.2±0.2 s
  - With mm vcpu_id: 15.9±0.4 s (**+4.6 %**)
- 10 processes, each per-thread (1000 messages)
  - Baseline: 8.1±0.4 s
  - With mm vcpu_id: 8.3±0.4 s

Benchmarks (perf bench)

No significant scheduler overhead noticed. However other workloads may be more sensitive.

- perf bench message (process)
  - baseline: 134±9 ms
  - vcpu-id no-optimization: 139±7 ms
- perf bench message (threaded)
  - baseline: 114±7 ms
  - vcpu-id no-optimization: 111±7 ms
- perf bench message 2 instances (threaded)
  - baseline: 161±16 ms
  - vcpu-id no-optimization: 154±14 ms
- perf bench pipe
  - baseline: 8.8±2.0 s
  - vcpu-id no-optimization: 8.2±1.7 s

Virtual CPU-ID Allocator: Opt-in vs Always-on

- Considering the impact on scheduler performance, Google's approach [3] is to make the vcpu-id allocation opt-in per-process.
- If our aim is to have glibc use this for its memory allocator, the opt-in approach simply won't help in the long run. We need to consider the performance impact more carefully.

Performance improvements

- Single-threaded mm
  - Statically use vcpu-id 0
    - except on NUMA, where a different constant can be returned for each NUMA node.
- Scheduling between threads from the same mm
  - Hand over the vcpu-id from previous to next thread.
- Scheduling between threads from different mm
  - Per-runqueue cache of (vcpu-id, mm) pairs.

Schedstats Counter Profiling

* perf bench sched messaging (single instance, multi-process):

On sched-switch:

single-threaded vcpu-id:	99.98	%
transfer between threads:	0	%
runqueue cache hit:	0.02	%
runqueue cache eviction (bit-clear):	0	%
runqueue cache discard (bit-clear):	0	%
vcpu-id allocation (bit-set):	0	%

On release mm:

vcpu-id remove (bit-clear):	0	%

On migration:

vcpu-id remove (bit-clear):	0	%

EfficiOS

Schedstats Counter Profiling

* perf bench sched messaging -t (single instance, multi-thread):

On sched-switch:

single-threaded vcpu-id:	0.1	%
transfer between threads:	98.2	%
runqueue cache hit:	1.1	%
runqueue cache eviction (bit-clear):	0.0	%
runqueue cache discard (bit-clear):	0.0	%
vcpu-id allocation (bit-set):	0.3	%

On release mm:

vcpu-id remove (bit-clear):	0.2	%

On migration:

vcpu-id remove (bit-clear):	0.1	%

EfficiOS

Schedstats Counter Profiling

* perf bench sched messaging -t (two instances, multi-thread):

On sched-switch:

single-threaded vcpu-id:	0.1	%
transfer between threads:	89.5	%
runqueue cache hit:	9.7	%
runqueue cache eviction (bit-clear):	0.0	%
runqueue cache discard (bit-clear):	0	%
vcpu-id allocation (bit-set):	0.4	%

On release mm:

vcpu-id remove (bit-clear):	0.2	%

On migration:

vcpu-id remove (bit-clear):	0.1	%

Schedstats Counter Profiling

* perf bench sched pipe (one instance, multi-process):

On sched-switch:

single-threaded vcpu-id:	100.00 %
transfer between threads:	0.00 %
runqueue cache hit:	0.00 %
runqueue cache eviction (bit-clear):	0 %
runqueue cache discard (bit-clear):	0 %
vcpu-id allocation (bit-set):	0.00 %

On release mm:

vcpu-id remove (bit-clear):	0 %

On migration:

vcpu-id remove (bit-clear):	0.00 %

Benchmarks (hackbench)

Scheduler overhead is non-significant.

10 groups using 40 fd, each sender passes messages of 100 bytes, x86-64 E5-2630

- Per-process (10000 messages)
  - Baseline: 10.5±0.3 s
  - With mm vcpu_id: 10.5±0.5 s
- Per-thread (10000 messages)
  - Baseline: **15.2±0.2 s**
  - With mm vcpu_id: **15.0±0.1 s**
- 10 processes, each per-thread (1000 messages)
  - Baseline: 8.1±0.4 s
  - With mm vcpu_id: 8.4±0.3 s

Benchmarks (perf bench)

No significant scheduler overhead noticed. However other workloads may be more sensitive.

- perf bench message (process)
  - baseline: 134±9 ms
  - vcpu-id with-optimization: 138±7 ms
- perf bench message (threaded)
  - baseline: 114±7 ms
  - vcpu-id with-optimization: 112±7 ms
- perf bench message 2 instances (threaded)
  - baseline: 161±16 ms
  - vcpu-id with-optimization: 157±14 ms
- perf bench pipe
  - baseline: 8.8±2.0 s
  - vcpu-id with-optimization: 8.4±1.6 s

Benchmarks

- I would kindly ask Google to share benchmarks covering execution of their workload with and without virtual CPU ID when they find time to test my patches.
- Performance benefit for tcmalloc ?
- What is the overhead with/without scheduler fast-path optimizations ?
  - Is the complexity of those optimizations needed ?

NUMA

- My design assumption here is that NUMA should really be only an optimization which works "as is" (although less efficiently) without code changes when user-space is not NUMA-aware.
- Guarantee needed is similar to a ”real" cpu id with respect to its NUMA topology:
  - the mapping between cpu id and NUMA node ID stays invariant if there is no NUMA topology change.
- Guarantee for mm vcpu_id:
  - for the lifetime of a process, the mapping between vcpu_id and NUMA node id stay invariant unless there is a NUMA topology change in the kernel.

NUMA (2)

- This allow allocating NUMA-local memory on first use of a vcpu-id, and then all following accesses to from this vcpu-id will be NUMA-local (except NUMA topology reconfiguration).
- Expose an additional node_id field in struct rseq, to be loaded along with mm_vcpu_id within a rseq c.s. when memory needs to be allocated on behalf of the current NUMA node.

NUMA (3)

- Internally, this is implemented by adding the following bitmaps to each mm:
  - vcpu-id allocation bitmap (one bitmap per NUMA node),
  - overall NUMA node vcpu-id allocation bitmap.
- Implement "find first" operations over pairs of cpumasks:
  - cpumask_first_zero_and_zero(),
  - cpumask_first_one_and_zero().
- Updates to those NUMA-specific bitmaps only need to be done the first time a vcpu-id is allocated for a memory space. Fast-paths are only lookups.

Open Questions

- Should the scheduler use the per-NUMA-node vcpu ID allocation bitmap into account when taking migration decisions ?
  - This could ensure that the scheduler favors re-using already allocated vcpu-ids rather than migrating threads to numa nodes with few vcpu-ids allocated.
- Extend struct mm (memory space) or add a pointer to struct mm ?
- Perhaps my runqueue { mm, vcpu_id } cache idea could be re-used to cache mm user references as well.
- I would like to make this available for shared memory as well (per-container). See Containers MC. [4]

References

[1] “Supporting per-processor local-allocation buffers using lightweight user-level preemption notification”, Alex Garthwaite, David Dice, Derek White, Proceedings of the 1st International Conference on Virtual Execution Environments, VEE 2005, Chicago, IL, USA, June 11-12, 2005

[2] [PATCH v3 00/23] RSEQ node id and virtual cpu id extensions

- https://lore.kernel.org/lkml/20220729190225.12726-1-mathieu.desnoyers@efficios.com/

[3] tcmalloc struct kernel_rseq

- https://github.com/google/tcmalloc/blob/master/tcmalloc/internal/linux_syscall_support.h#L26

[4] “Restartable Sequences: Scaling Per-Core Shared Memory Use in Containers”, Linux Plumbers Conference Container MC

- https://lpc.events/event/16/contributions/1238/

Discussion