Improve Linux User-Space Core Libraries with Restartable Sequences
Speaker

• Mathieu Desnoyers
• CEO at EfficiOS Inc.
• Maintainer of: LTTng kernel and user-space tracers, Userspace RCU library, Linux kernel membarrier and rseq system calls,
• Author of the Restartable Sequence patchset merged into Linux 4.18.
• What are restartable sequences (rseq) ?
• Restartable sequences:
  – Use-cases,
  – Algorithm,
  – Upstreaming status,
• Librseq,
• Glibc rseq thread registration,
Content

- Restartable Sequences Shortcomings,
- cpu_opv system call,
- Rseq adoption: user-space projects,
- Benchmarks.
What are Restartable Sequences (rseq)?

- Sequences of user-space instructions with a preparation stage, finalized by a single commit instruction,
- Either executed atomically with respect to preemption, migration, signal delivery, or aborted before the final commit instruction,
- Kernel guarantees “atomic” execution by moving IP to abort handler if needed,
- Use-cases: super-fast update operations on per-cpu data in user-space.
Restartable Sequences Use-Cases

- LTTng-UST ([http://lttng.org](http://lttng.org))
  - User-space tracing in memory buffers shared across processes
- Userspace RCU ([http://liburcu.org](http://liburcu.org))
  - Single-process per-cpu grace period tracking,
  - Multi-process per-cpu grace-period tracking,
- jemalloc and glibc per-cpu memory allocator,
- Application-level per-cpu statistics counters,
- ARM64 PMC read from user-space on big.LITTLE without fault on migration.
Restartable Sequences Algorithm

**Restartable Sequence Critical Section**

Thread-Local Storage __rseq_abi:

```c
struct rseq {
    int32_t cpu_id;
    struct rseq_cs *rseq_cs;
};
```

```c
struct rseq_cs {
    void *start_ip;
    void *post_commit_ip;
    void *abort_ip;
};
```
Restartable Sequences Algorithm

- Restartable sequence critical section:
  - Preemption or signal delivery interrupting critical section move instruction pointer to abort handler before returning to user-space,
  - Needs to be implemented in assembly,
  - Ends with a single store instruction.
Restartable Sequences Upstreaming Status

- **Linux 4.18:**
  - rseq system call merged,
  - rseq wired up for x86 32/64, powerpc 32/64, arm 32, mips 32/64,

- **Linux 4.19:**
  - rseq wired up for arm 64, s390 32/64,

- **Ongoing work:**
  - librseq,
  - glibc rseq registration/unregistration at thread start/exit,
  - new cpu_opv system call.
Librseq

- User-space library,
- Handle restartable sequence thread registration with explicit library API call by each thread,
- Provides headers implementing rseq inline assembly code for common use-cases, e.g. per-cpu compare-and-store and per-cpu add.
Glibc Rseq Thread Registration (Ongoing Work)

- Automatically register rseq at thread start and nptl init, unregister rseq at thread exit (ongoing work),
- Introduce a reference counter field in rseq Thread-Local Storage to allow glibc as well as early-adopter applications and libraries to manage rseq registration ownership.
Restartable Sequences Shortcomings

• Interaction with debugger single-stepping:
  – Restartable sequences will loop forever (no progress) if single-stepped by a debugger.

• Unable to migrate data between per-cpu data structures without changing the CPU affinity mask, e.g.:
  – Migration of free memory between per-cpu pools,
  – Migration of tasks by per-cpu user-space task schedulers.

• Handling critical sections in signal handlers nested early/late over thread creation/destruction when rseq is not registered is not straightforward.
cpu_opv() System Call (Ongoing Work)

- Vector of operations (similar to iovec) to be executed with preemption disabled, on a given CPU,
- Can be used as fallback when rseq fails,
- Kernel temporarily pins all pages touched by operations,
- Limited to 16 operations. Overall sequence of operations limited to 4216 bytes (cache-cold: 4.7µs preemption off latency on x86-64).
- Implements “compare” eq/ne operations that allow checking whether input data provided by user-space has not been modified concurrently.
- Implements memcpy, add, bitwise, shift, and memory barrier operations.
Rseq Adoption: User-Space Projects

- Library early adopters (likely for: lttng-ust, liburcu,jemalloc)
  - Provide their own weak __rseq_abi TLS symbol (with refcount field),
  - Lazy registration, pthread_setspecific for unregistration,
- Application early adopters
  - Provide their own weak __rseq_abi TLS symbol (with refcount field),
    or implement their own library for rseq,
  - Explicit registration/unregistration at thread start and before it exits,
- Integration into glibc
  - Provide strong __rseq_abi TLS symbol (with refcount field),
  - Registration at pthread start and nptl init, unregistration at thread exit,
  - Use by glibc memory allocator.
Benchmarks

• Test hardware
  – arm32: ARMv7 Processor rev 4 (v7l) "Cubietruck", 2-core,
  – x86-64: Intel E5-2630 v3@2.40GHz, 16-core, hyperthreading enabled.
### Benchmarks

* Per-CPU statistic counter increment

<table>
<thead>
<tr>
<th></th>
<th>getcpu+atomic (ns/op)</th>
<th>rseq (ns/op)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>arm32:</td>
<td>344.0</td>
<td>31.4</td>
<td>11.0</td>
</tr>
<tr>
<td>x86-64:</td>
<td>15.3</td>
<td>2.0</td>
<td>7.7</td>
</tr>
</tbody>
</table>

* LTTng-UST: write event 32-bit header, 32-bit payload into tracer per-cpu buffer

<table>
<thead>
<tr>
<th></th>
<th>getcpu+atomic (ns/op)</th>
<th>rseq (ns/op)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>arm32:</td>
<td>2502.0</td>
<td>2250.0</td>
<td>1.1</td>
</tr>
<tr>
<td>x86-64:</td>
<td>117.4</td>
<td>98.0</td>
<td>1.2</td>
</tr>
</tbody>
</table>

* liburcu percpu: lock-unlock pair, dereference, read/compare word

<table>
<thead>
<tr>
<th></th>
<th>getcpu+atomic (ns/op)</th>
<th>rseq (ns/op)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>arm32:</td>
<td>751.0</td>
<td>128.5</td>
<td>5.8</td>
</tr>
<tr>
<td>x86-64:</td>
<td>53.4</td>
<td>28.6</td>
<td>1.9</td>
</tr>
</tbody>
</table>
Benchmark: Prototype Rseq Integration in jemalloc

- Using rseq with per-cpu memory pools in jemalloc at Facebook (based on rseq 2016 implementation).
- The production workload response-time has 1-2% gain avg. latency, and the P99 overall latency drops by 2-3%.
Benchmark: Reading the Current CPU Number

ARMv7 Processor rev 4 (v7l)
Machine model: Cubietruck

- Baseline (empty loop): 8.4 ns
- Read CPU from rseq cpu_id: 16.7 ns
- Read CPU from rseq cpu_id (lazy registration): 19.8 ns
- glibc 2.19-0ubuntu6.6 getcpu: 301.8 ns
- getcpu system call: 234.9 ns

x86-64 Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz:

- Baseline (empty loop): 0.8 ns
- Read CPU from rseq cpu_id: 0.8 ns
- Read CPU from rseq cpu_id (lazy registration): 0.8 ns
- Read using gs segment selector: 0.8 ns
- "lsl" inline assembly: 13.0 ns
- glibc 2.19-0ubuntu6 getcpu: 16.6 ns
- getcpu system call: 53.9 ns
• **linux-rseq development (volatile):**

• **librseq development:**
  – https://github.com/compudj/librseq/

• **glibc rseq integration development (volatile):**
  – https://github.com/compudj/glibc-dev/

• **Additional tests/benchmarks branch for rseq (volatile):**
  – https://github.com/compudj/rseq-test
Related Presentations

• “PerCpu Atomics”, Paul Turner, Andrew Hunter, Linux Plumbers Conference 2013


  - https://lwn.net/Articles/KernelSummit2017/
Related Articles

- Restartable sequences
  - https://lwn.net/Articles/650333/
- Restartable sequences restarted
  - https://lwn.net/Articles/697979/
- Restartable sequences and ops vectors
  - https://lwn.net/Articles/737662/
Questions ?