Enabling Fast Per-CPU User-Space Algorithms with Restartable Sequences
What are Restartable Sequences (rseq) ?

- Idea originating from Paul Turner and Andrew Hunter (Google),
- Synchronization mechanism for per-CPU data,
- Collaboration between kernel and user-space,
  - Shared Thread-Local Storage (TLS) between kernel and user-space,
  - Registered through a new system call,
  - Kernel and user-space restart critical section if preempted or interrupted by a signal.
- Accelerate algorithms which make use of per-CPU data.
Problems Addressed by Restartable Sequences

- Modifying *per-CPU* data from user-space is slow compared to *Thread-Local Storage (TLS)* data updates,
  - Due to atomicity requirements caused by preemption and migration,
  - A thread can be preempted at any point, requiring cpu-local atomic operations,
  - A thread can be migrated at any point between getting the current CPU number and writing to per-CPU data,
  - Requires lock-prefixied atomic operations on x86, load-linked/store-conditional on ARM, PowerPC, ...
Problems Addressed by Restartable Sequences

- Modifying data shared between **threads** and **signal handlers** requires cpu-local atomic operations,
  - Due to atomicity requirements caused by signal delivery,
  - A thread can be interrupted by a signal handler at any point (unless signals are explicitly ignored or blocked), requiring cpu-local atomic operations,
  - Requires cpu-local atomic operations on x86, load-linked/store-conditional on ARM, PowerPC, ...
  - Those are slower than normal load/store operations,
  - Affects both TLS and per-CPU data structures.
Problems Addressed by Restartable Sequences

- User-space cannot efficiently disable preemption, migration, nor signal delivery, for short critical sections,
- On x86, LOCK-prefixed and cpu-local atomic operations are costly compared to non-atomic operations,
- On Power8, Load-Linked/Store-Conditional atomic operations are costly compared to non-atomic operations,
  - May also be the case on large ARM 32/64 SoCs?
Use-Cases Benefiting from per-CPU data over TLS

- Memory allocators speed and memory usage,
  - Workloads with more threads than CPUs,
  - Workloads with blocking threads,
  - E.g. webserver performing blocking I/O, databases, web browsers,

- Ring buffers speed (tracing),
  - [http://lttng.org](http://lttng.org) user-space tracer,

- RCU grace-period tracking in user-space,
  - Speed and facilitates implementation of multi-process RCU,
  - [http://liburcu.org](http://liburcu.org)
Restartable Sequences Algorithm

Nestable C.S.

Sequence counter protected critical section. Can be implemented in C. Sequence counter comparison done within IP-fixup critical section.

Commit

Critical section protected by kernel moving instruction pointer to abort handler. Needs to be implemented in assembly. Ends with a single store instruction.

Preemption or signal delivery restarts the critical section constructed from the two overlapping regions.
ABI: Restartable Sequences TLS Structure

```
struct rseq {
    int32_t cpu_id;
    uint32_t event_counter;
    struct rseq_cs *rseq_cs;
};
```

(simplified: pointers are actually 64-bit integers)
ABI: RSeq Critical Section Descriptor

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

(simplified: pointers are actually 64-bit integers)
Using Restartable Sequences

- Intended to be used though library
  - librseq.so / rseq.h
- Register/unregister threads:
  - rseq_register_current_thread()
  - rseq_unregister_current_thread()
  - Can be done lazily with pthread_key
- Mark beginning of Nestable C.S.
  - rseq_start()}
Using Restartable Sequences

- Commit sequence:
  - rseq_finish()
    - Single-word store (final commit)
  - rseq_finish2()
    - Speculative single word store followed by final commit single-word store
    - Can be used for ring buffer pointer push:
      - Speculative store to next slot (pointer) followed by final store to head offset,
  - rseq_finish_memcpy()
    - Speculative copy of an array followed by final commit single-word store,
    - Can be used for ring buffer inline data push:
      - Speculative memcpy into ring buffer, followed by final store to head offset.
Interaction with Debugger Single-Stepping

• Restartable sequences will loop forever (no progress) if single-stepped by a debugger,

• Handling of debugger single-stepping can be performed entirely user-space,

• Three approaches:
  - Split counters fallback (fastest, only for split-counters),
  - Flag test with locking fallback,
  - Reference counter test and atomic operation fallback.
Per-CPU Counter Example: Split Counters Fallback

```
struct rseq_state rseq_state;
intptr_t *targetptr, newval;
int cpu;
bool result;

rseq_state = rseq_start();
cpu = rseq_cpu_at_start(rseq_state);
newval = data->c[cpu].rseq_count + 1;
targetptr = &data->c[cpu].rseq_count;
if (unlikely(!rseq_finish(targetptr, newval, rseq_state)))
    uatomic_inc(&data->c[cpu].count);
```

```
struct test_data_entry {
    uintptr_t count;
    uintptr_t rseq_count;
};
```

Read by summing count + rseq_count for each CPU.
Per-CPU Counter: Locking Fallback

```
struct test_data_entry {
    uintptr_t rseq_count;
};
```

```
struct rseq_state rseq_state;
intptr_t *targetptr, newval;
int cpu;
bool result;

do_rseq(&rseq_lock, rseq_state, cpu, result,
    targetptr, newval,
    {
        newval = data->c[cpu].rseq_count + 1;
        targetptr = &data->c[cpu].rseq_count;
    });
```

The do_rseq() macro does two attempts with rseq, then fallback to locking.
Per-CPU Counter: Reference Count Fallback

```c
rseq_state = rseq_start();
if (!uatomic_read(&rseq_refcount)) {
    /* Load refcount before loading rseq_count. */
    cmm_smp_rmb();
    cpu = rseq_cpu_at_start(rseq_state);
    newval = data->c[cpu].rseq_count + 1;
    targetptr = &data->c[cpu].rseq_count;
    if (likely(rseq_finish(targetptr, newval, rseq_state)))
        return;       /* Success. */
}
put_ref = refcount_get_saturate(&rseq_refcount);
cpu = rseq_current_cpu_raw();
uatomic_inc(&data->c[cpu].rseq_count);
if (put_ref) {
    /* inc rseq_count before dec refcount, match rmb. */
    cmm_smp_wmb();
uatomic_dec(&rseq_refcount);
}
```
## Restartable Sequences: ARMv7 Benchmarks

**ARMv7 Processor rev 4 (v7l)**
**Machine model: Cubietruck**

<table>
<thead>
<tr>
<th>Counter increment speed (ns/increment)</th>
<th>1 thread</th>
<th>2 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>global volatile inc (baseline)</td>
<td>5.6</td>
<td>N/A</td>
</tr>
<tr>
<td>percpu rseq inc</td>
<td>40.8</td>
<td>41.2</td>
</tr>
<tr>
<td>percpu rseq rlock cmpxchg</td>
<td>49.6</td>
<td>50.1</td>
</tr>
<tr>
<td>percpu rseq spinlock</td>
<td>95.9</td>
<td>96.7</td>
</tr>
<tr>
<td>percpu atomic inc</td>
<td>56.3</td>
<td>56.4 (__sync_add_and_fetch_4)</td>
</tr>
<tr>
<td>percpu atomic cmpxchg</td>
<td>66.1</td>
<td>67.1 (__sync_val_compare_and_swap_4)</td>
</tr>
<tr>
<td>global atomic inc</td>
<td>49.6</td>
<td>82.4 (__sync_add_and_fetch_4)</td>
</tr>
<tr>
<td>global atomic cmpxchg</td>
<td>52.9</td>
<td>181.0 (__sync_val_compare_and_swap_4)</td>
</tr>
<tr>
<td>global pthread mutex</td>
<td>155.3</td>
<td>932.5</td>
</tr>
</tbody>
</table>
### Restartable Sequences: x86-64 Benchmarks

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>1 Thread</th>
<th>8 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>global volatile inc (baseline)</td>
<td>2.3</td>
<td>N/A</td>
</tr>
<tr>
<td>percpu rseq inc</td>
<td>2.1</td>
<td>2.6</td>
</tr>
<tr>
<td>percpu rseq rlock cmpxchg</td>
<td>3.0</td>
<td>3.4</td>
</tr>
<tr>
<td>percpu rseq spinlock</td>
<td>4.9</td>
<td>5.2</td>
</tr>
<tr>
<td>percpu LOCK; inc</td>
<td>6.2</td>
<td>6.9</td>
</tr>
<tr>
<td>percpu LOCK; cmpxchg</td>
<td>10.0</td>
<td>11.6</td>
</tr>
<tr>
<td>global LOCK; inc</td>
<td>6.2</td>
<td>134.0</td>
</tr>
<tr>
<td>global LOCK; cmpxchg</td>
<td>10.0</td>
<td>356.0</td>
</tr>
<tr>
<td>global pthread mutex</td>
<td>19.2</td>
<td>993.3</td>
</tr>
</tbody>
</table>
## Restartable Sequences: Power8 Benchmarks

**Power8 Guest with 64 vcpus (8 vcores) (atomics implemented with relaxed ll/sc):**

<table>
<thead>
<tr>
<th>Counter increment speed (ns/increment)</th>
<th>1 thread</th>
<th>16 threads</th>
<th>32 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>global volatile inc (baseline)</strong></td>
<td>6.5</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>percpu rseq inc</strong></td>
<td>6.8</td>
<td>7.1</td>
<td>8.4</td>
</tr>
<tr>
<td><strong>percpu rseq rlock cmpxchg</strong></td>
<td>7.0</td>
<td>10.0</td>
<td>15.3</td>
</tr>
<tr>
<td><strong>percpu rseq spinlock</strong></td>
<td>19.3</td>
<td>24.4</td>
<td>51.4</td>
</tr>
<tr>
<td><strong>percpu atomic inc</strong></td>
<td>16.3</td>
<td>17.5</td>
<td>21.1</td>
</tr>
<tr>
<td><strong>percpu atomic cmpxchg</strong></td>
<td>30.9</td>
<td>32.5</td>
<td>49.9</td>
</tr>
<tr>
<td><strong>global atomic inc</strong></td>
<td>18.5</td>
<td>1937.6</td>
<td>4701.8</td>
</tr>
<tr>
<td><strong>global atomic cmpxchg</strong></td>
<td>26.9</td>
<td>4106.2</td>
<td>12642.6</td>
</tr>
<tr>
<td><strong>global pthread mutex</strong></td>
<td>400.0</td>
<td>4167.3</td>
<td>8462.9</td>
</tr>
</tbody>
</table>
CPU Number Getter Speedup

- ARM32 currently reads current CPU number through system call,
- ARM32 cannot implement vDSO approaches similarly to x86, no segment selector,
- Solution: add a current CPU number field to the rseq TLS ABI,
  - Kernel updates the current CPU number value before each return to user-space,
  - User-space gets the current CPU number from a simple thread-local storage variable load.
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 register): 19.8 ns
- glibc 2.19-0ubuntu6.6 getcpu: 301.8 ns
- getcpu system call: 234.9 ns

Speedup rseq over glibc getcpu: 35:1
CPU Number Getter Speedup on x86-64

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 register): 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

- It turns out this approach can also be used to improve sched_getcpu() on x86-64.
- Speedup rseq over glibc getcpu: approximately 20:1
Current Restartable Sequence Status

- Currently gathering real-life application benchmarks to support upstream Linux inclusion,
- Currently shows improvement for:
  - User-space tracing (LTTng-UST)
    - Intel i7-5600U@2.60GHz 109ns/event -> 90ns/event
  - Per-thread memory allocation
    - Allocator fragmentation multi-threaded stress-test memory consumption,
    - Facebook production workload response-time: 1-2% gain avg latency, P99 overall latency drop by 2-3%
  - Userspace RCU
    - Allow implementing multi-process grace periods with fast read-side.
Disclaimers/Links

- Benchmarks in this presentations were taken on v8 of the patchset as posted on LKML. Some speed improvements have been pushed into the development branches since then.
- Current development branch for rseq (volatile):
  - https://github.com/compudj/linux-percpu-dev/tree/rseq-fallback
- Current benchmark branch for rseq (volatile):
  - https://github.com/compudj/rseq-test
- Restartable sequences restarted
  - https://lwn.net/Articles/697979/
Discussion/Questions