LPC - PerCpu Atomics

Paul Turner <pjt@google.com>
Andrew Hunter <ahh@google.com>

RSEQ: Restartable Sequences

Problem:

Synchronizing per-cpu operations from user-space is hard.

- *Liable to be pre-empted or change cpus at any time*

Writing code that is safe from this is difficult.

- Thread-safe locks are expensive (atomic operations).
- Synchronizing with the kernel (syscall/etc) is expensive.
- Asking the kernel not to pre-empt not viable
  - How do you handle a contended per-cpu spinlock?
  - How do you handle "bad" code that loops in such a section?
  - How do you handle "good" code that's still spending too much time in such sections?
  - How do we interrupt a "low" priority thread in such an operation to allow a more important thread to proceed? This does not solve being able to suspend your thread's execution in the general case.

RSEQ: Restartable Sequences

Key Observations:

- Many operations only require data atomicity around a single **commit** operation. e.g. for a linked list insert we only need to be able to guarantee that we set "*head->next* = *new*" atomically. The other steps, such as setting "*new->next* = *head*" are **preparatory** and may be discarded/recomputed in the event *head* changes.
- If we restrict our domain to a single CPU then we can catch all such changes above by treating context-switches as an "edge".
- With the above restriction we can use a non-atomic store to accomplish the **commit**.

Motivating example: consider malloc

Consider a caching malloc's new/free operations using per-cpu caches.

Prepare:
Figure out what what the cache for the current cpu is and where our object belongs in it. Prepare the object for insertion.

Write:
Commit the object to the local cache by writing at the insertion point determined in our preparation above.

Restart:
Re-prepare the object for insertion.

How do we generalize this?

RSEQ: Prepare-Write-Restart block

Introduce a new model "Prepare, Write, Restart" (PWR) model for user-space to use in these transactions.

A PWR block consists of 3 parts:

- **Prepare block**
- **Write instruction** (immediately following the prepare block)
- **Restart block**

Invariant:

Any processing of a PWR block will guarantee that the "PW" phases are **serialized** on a per-cpu basis. A natural corollary to this is that local data-store ordering guarantees a commit will always be visible to any restarted execution on the same CPU.

RSEQ: Prepare Block

This is the '**restartable**' state, it consists of any data/algorithmic operations required to set up, or a gate, a commit.

Any context-switch within the prepare block will result in the resumption of execution occurring at the start of this block, not where it left of.

Note in particular "algorithmic" operations above. An example of this is checking of state required to issue the commit, e.g. is the spinlock we are attempting to acquire uncontended?

RSEQ: Write Block

Our write block consists of a single instruction store which 'commits' the work performed in our preparatory block.

Our invariant guarantees that any PWR block which completes the write instruction (commit) ran without interruption.

RSEQ: Restart Block

What is a restart? When do we need to do it?

We need to restart when something may have updated the realities used held our pre-commit preparation.

Examples:

- We've been pre-empted and another PWR block may have run.
- We may also have been migrated to a new cpu in this time.

Typically restarting consists only of unwinding and resuming execution at the beginning of our preparation block.

RSEQ: Linked list push

```
/* push element into a per-cpu linked list.
   Returns: cpu the element was stored on */
int rseq_push_linked_list(per_cpulist_t *target, list_t *element)
```

- **Prepare**
  1. Determine the current CPU
  2. element->next = target->head[cpu]
- **Write**
  1. target->head[cpu]->next = element
- **Restart**
  1. Resume execution at Prepare above

RSEQ: Actual example - Spinlock

```
/*----------------------------------------------------------*/
/* int rseq_spin_lock_raw (void *data, int slot); */
  .globl rseq_spin_lock_raw
  .type  rseq_spin_lock_raw, @function
ras_spin_lock_raw:
  .cfi_startproc
ras_spin_lock_raw_region0:
    FETCH_LOCAL_DATA (%rdi, %rsi, %r10)
    ras_spin_lock_raw_region1:
    cmpq $0, (%r10)
    jne ras_spin_lock_raw_region1
    movq $1, (%r10)
ras_spin_lock_raw_region2:
    ret
ras_spin_lock_raw_region3:
   .cfi_endproc
/*----------------------------------------------------------*/

/* Restart handler */
HANDLE_REGION_PREFIX (ras_spin_lock_raw_region, 0, 2, 0)
HANDLE_REGION_PREFIX (ras_spin_lock_raw_region, 2, 3, 2)
```

Prepare:
Compute address for current cpu's lock-word

(FETCH_LOCAL_DATA is a macro which retrieves the currently active cpu into %rax, as well as the target for the lock-word into % r10)

Commit
Write 1 into the lock-word

RSEQ: Limitations

- The commit* **must** be a single store, this is what allows us to exploit memory ordering to avoid atomic operations.
- Once within a *PWR* block you **can not** safely call external routines as there is no way to properly handle a restart in this instance.
- Restarts look like an unexpected *long-jmp,* the sequences themselves must therefore be hand-coded.
- Modifying another CPUs per-cpu data requires care.

*** Note:**
For commits requiring multiple updates, e.g. a doubly-linked list, percpu spinlocks or DCAS very effective.

RSEQ: Currently supported semantics

The below semantics are all on a *per-cpu* basis.

- spinlocks ("free" to acquire in fast-path, guaranteed to be running cpu at time of acquisition)
- fast event counters (no atomic add/sub!)
- "free" cmpxchg
- “free” double-compare-and-swap

Easy to add more--entirely from userspace.

Fence, RSEQ becomes RCU

Normally can't modify another CPU's RSEQ data--even to privatize.

- breaks any ability to reclaim resources
- problem is other CPUs operating on out of date data...hmm…

Gate all accesses through a pointer read **inside critical section**

Introduce new operation, **Fence**:

- "Restarts" any other CPUs executing a critical section.
- Guarantees a memory barrier on any interrupted cpus.

Update becomes:
<update pointer>; Fence();
[No references to old pointer can now exist.]