Improving CPU Isolation with per-cpu spinlocks: performance cost and analysis

Leonardo Brás Soares Passos
Linux Plumbers Conference 2023

whoami

- Leonardo Brás Soares Passos
- Work @ Red Hat (Virt-team)
  - Linux Kernel
    - Improving CPU Isolation & RT
    - Improving RISC-V arch code (as a side quest)
  - QEMU
    - Improving Live Migration
- Find me: leobras @ {redhat.com, GitLab, GitHub, IRC}

Introduction

- What we want?
  - To run time-sensitive tasks without interruption
- How can we achieve this?
  - Keeping best-effort tasks in other CPUs (aka CPU Isolation)
- What is (one of the things) preventing it?
  - Scheduling work on isolated cpus:
    schedule_work_on(isolated_cpu)

Use case: per-cpu caches

- This is a very efficient strategy for sharing global resources on SMP systems:
  - Each CPU using the resource gets a per-cpu cache
  - Allocation and freeing resources happen in the local cache
  - When local cache is full (or empty), it accesses the global cache for expanding (or shrinking) the local cache.
    - This reduces the occurrences of global locking
  - Used in memcg, slub, swap.
- Issue: Actively reclaiming resources from remote per-cpu caches requires schedule_work_on(all_online_cpus).
  - An IPI is issued, interrupting the work of all online CPUs.

The generic code

```
/* Hotpath: work locally */
local_lock(s->lock);
do_local_work_on(s);
local_unlock(s->lock);

/* Eventually do remote work */
for_each_online_cpu(cpu){
        schedule_work_on(cpu, s->work);
}
```

The generic code

```
/* Hotpath: work locally */
local_lock(s->lock);
do_local_work_on(s);
local_unlock(s->lock);

/* Eventually do remote work */
for_each_online_cpu(cpu){
        schedule_work_on(cpu, s->work);
}
```

Generates an IPI for a remote CPU

The generic code

```
/* Hotpath: work locally */
local_lock(s->lock);
do_local_work_on(s);
local_unlock(s->lock);

/* Eventually do remote work */
for_each_online_cpu(cpu){
        schedule_work_on(cpu, s->work);
}
```

Bad for CPU Isolation

Generates an IPI for a remote CPU

Getting rid of the schedule_work_on()

- Replace local_locks() with per-cpu spinlocks()
  - Get local CPU’s spinlock() for each local operation
  - Get remote CPU’s spinlock() for remote operation
    - Instead of schedule_work_on() that cpu
- Remote operations don’t happen very often
  - Contention on per-cpu spinlocks() should be very rare.
- Some work done on this, by Mel Gorman[1]:
  - 01b44456a7aa7 ("mm/page_alloc: replace local_lock with normal spinlock")

local_lock + IPI → spinlock

```
/* Hotpath: work locally */
local_lock(s->lock);
do_local_work_on(s);
local_unlock(s->lock);

/* Eventually do remote work */
for_each_online_cpu(cpu){
        schedule_work_on(cpu, s->work);
}
```

```
/* Hotpath: work locally */
spin_lock(s->lock);
do_local_work_on(s);
spin_unlock(s->lock);

/* Eventually do remote work */
for_each_online_cpu(cpu){
        p = per_cpu_ptr(mystruct, cpu);
        spin_lock(p->lock)
        p->work(p);
        spin_unlock(p->lock)
}
```

Wait, are not spinlock() expensive?

- Most of a spinlock()’s cost comes from:
  - Contention
  - Getting cacheline exclusiveness
  - Memory barriers

Wait, are not spinlock() expensive?

- Most of a spinlock()'s cost comes from:
  - Contention
    - Not expected, since remote operations don't happen often
  - Getting cacheline exclusiveness
  - Memory barriers

Wait, are not spinlock() expensive?

- Most of a spinlock()'s cost comes from:
  - Contention
    - Not expected, since remote operations don't happen often
  - Getting cacheline exclusiveness
    - Local CPU will mostly have that per-cpu spinlock()'s cacheline exclusiveness already, since remote operations don't happen often
    - Invalidation will only happen after a remote operation
  - Memory barriers

Wait, are not spinlock() expensive?

- Most of a spinlock()'s cost comes from:
  - Contention
    - Not expected, since remote operations don't happen often
  - Getting cacheline exclusiveness
    - Local CPU will mostly have that per-cpu spinlock()'s cacheline exclusiveness already, since remote operations don't happen often
    - Invalidation will only happen after a remote operation
  - Memory barriers
    - Are not supposed to be that expensive

Talk is cheap,
show me the numbers!

Two tests

- Lock – Write – Unlock
  - Simplest case, which locks a local struct for writing
  - Most seen cases will lock to modify the values in the struct
  - Repeated 1Mi times, total time taken
- kmalloc() test
  - Modify mm/memcontrol.c to use spinlock() on stock_pcp
  - Setup a cgroup with memcg
  - Do 1Mi kmalloc() in that cgroup, take the total time
  - Suggested by Roman Gushchin [2]

Two tests - Rules

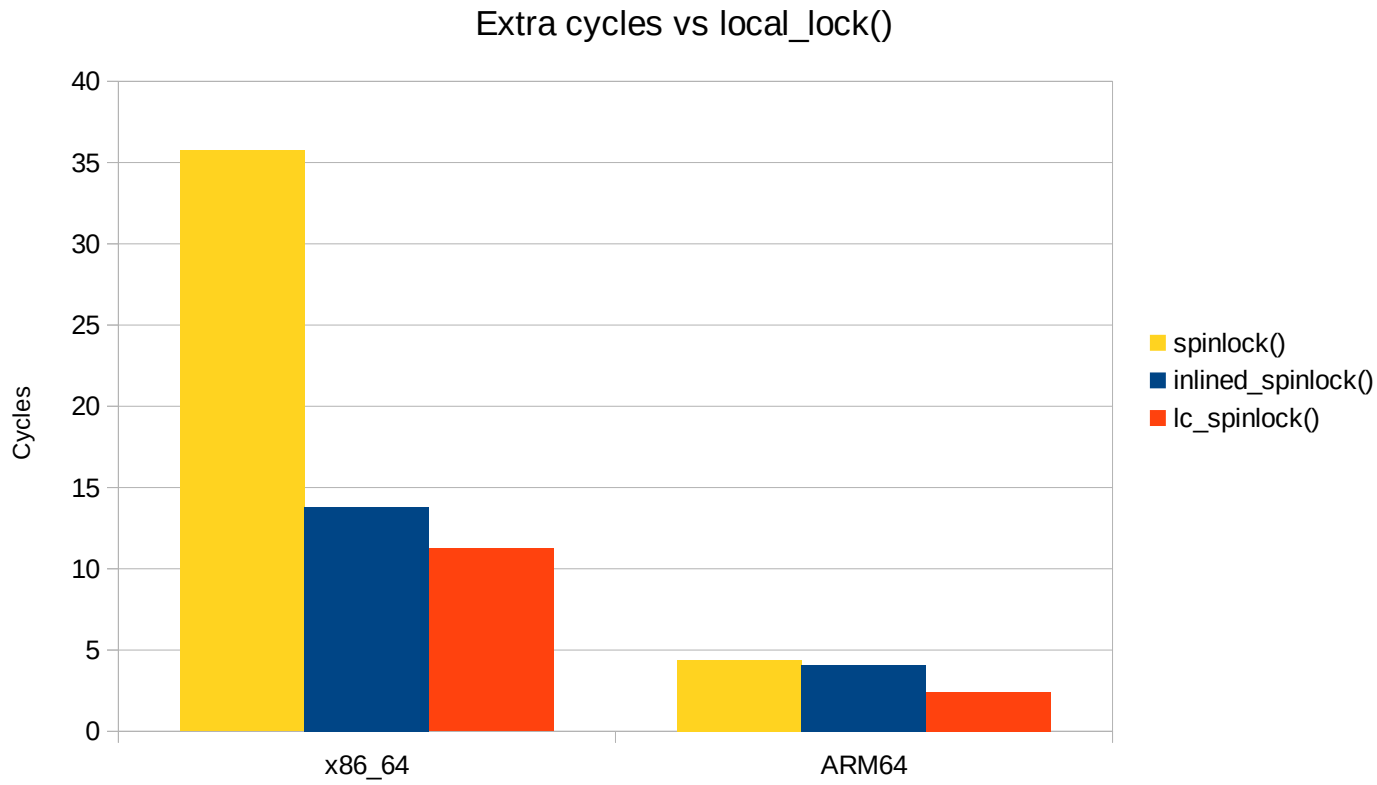
- Get 10x the lowest average of a 100 runs, PREEMPT_RT=n
  - Rules out interruptions
- Run on two CPU archs for reference
  - X86_64: AMD Epyc 7601 - 2017
  - ARM64: Marvell Octeon 10 (Neoverse N2) - 2023
- Collect function duration in cycles
  - x86_64: rdtsc_ordered()
  - ARM64: rmb() + arch_timer_read_cntpct_el0()
- Create a sysfs file to trigger the test
  - Add entry to memory_files[]

Test #1: Lock – Write - Unlock

```
s64 min_clk = LLONG_MAX;
for (int j = 0; j < 100; j++) {
    clk = get_clock();
    for (int i = 0; i < 1024 * 1024; i++) {
        test_lock(&t->lock);
        t->protected_data = i;
        test_unlock(&t->lock);
    }
    clk = get_clock() - clk;
    if (clk < min_clk)
        min_clk = clk;
}
```

- Tested Locks:
  - local_lock(): for reference
  - spinlock()
  - inlined_spinlock()
  - lc_spinlock()
    - Simple textbook spinlock() using xchg() as mechanism
    - Unfair spinlock,
    - For testing the difference of CAS (cmpxchg) vs Blind CAS (xchg)

Test #1: Results

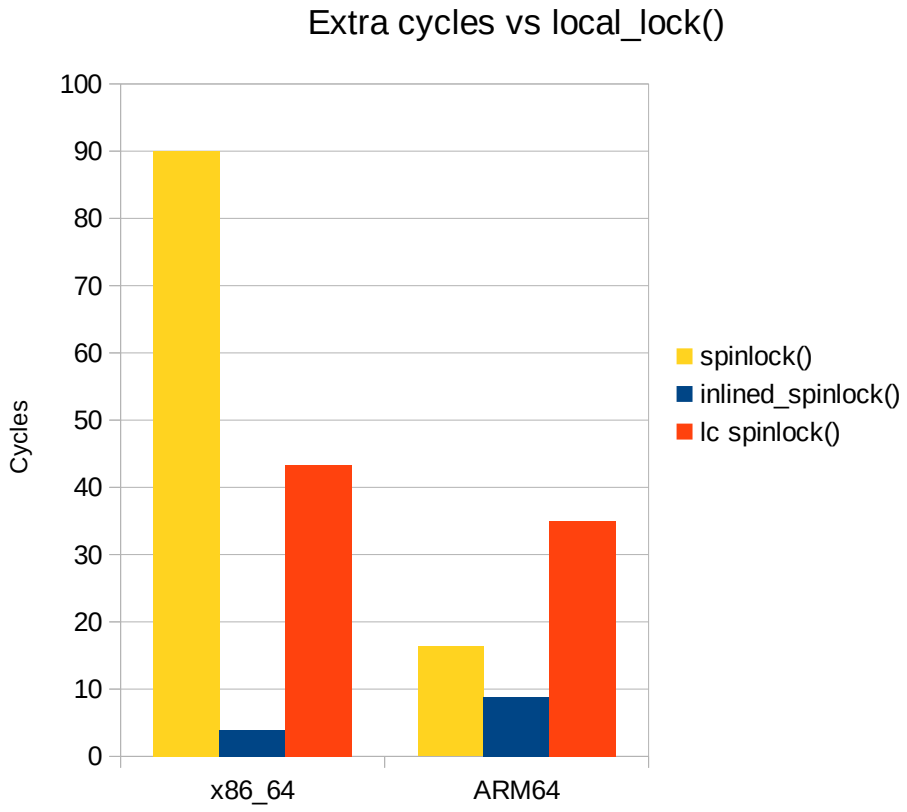


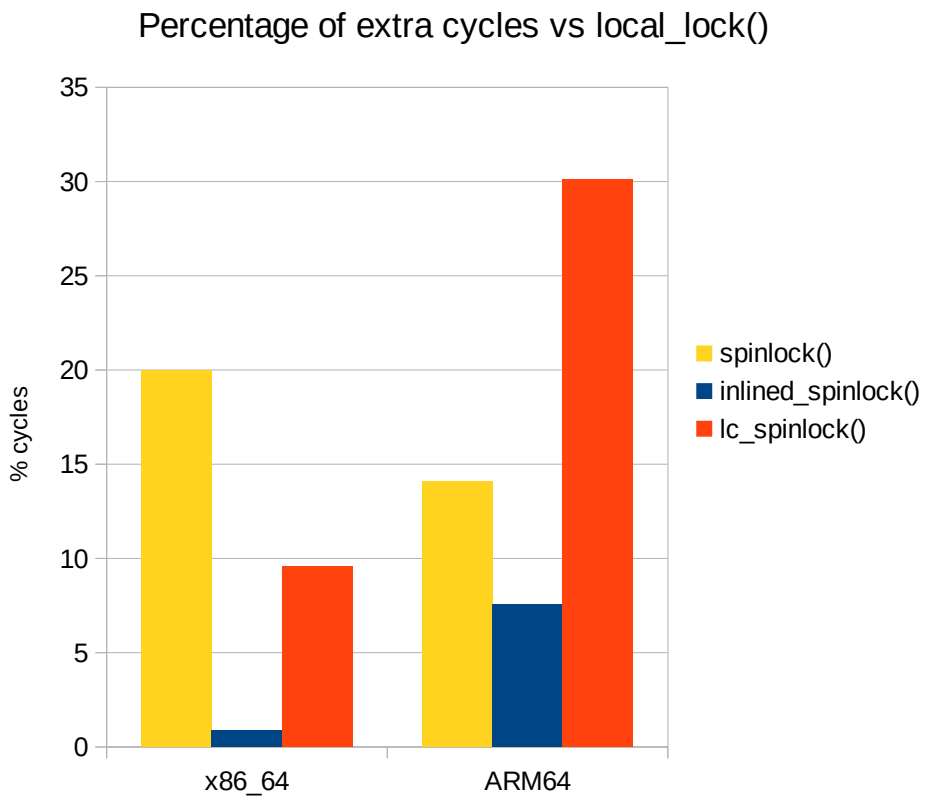
Test #2: kmalloc() test

```
s64 min_clk = LLONG_MAX;

ptrs = kvmalloc(sizeof(void *) * 1024 * 1024,
       GFP_KERNEL);

if (!ptrs)

    return -ENOMEM;

for (int j = 0; j < 100; j++) {

    clk = get_clock();

    for (int i = 0; i < 1024 * 1024; i++)

        ptrs[i] =
              kmalloc(8, GFP_KERNEL_ACCOUNT);

    clk = get_clock() - clk;

    if (clk < min_clk)

        min_clk = clk;

}
```

- Tested Locks on memcg:
  - local_lock(): for reference
  - spinlock()
  - inlined_spinlock()
  - lc_spinlock()
    - Simple textbook spinlock() using xchg() as mechanism
    - Unfair spinlock,
    - For testing the difference of CAS (cmpxchg) vs Blind CAS (xchg)

Test #2: Results





About test #1

- Just by inlining the spinlock() it saved:
  - 61% of the added cost for switching to spinlocks() on x86_64.
  - 7% of the added cost on ARM64.

- By using xchg (blind CAS) instead of cmpxchg (CAS) on lc_spinlock() it saved:
  - Extra 7% of the added cost of switching to spinlocks() on x86_64.
  - Extra 37% of the added cost on ARM64

Now on test #2

- Just by inlining the spinlock() it saved:
  - 95% of the added cost for switching to spinlocks() on x86_64.
  - 46% of the added cost on ARM64.
- The blind CAS on lc_spinlock() actually performed worse than inlined_spinlocks() in both archs.

Conclusion

- inlined_spinlock() would be the best replacement for local_lock trying to improve CPU Isolation
- Both tests showed that switching from local_lock() to inlined_spinlock() is not too expensive:
  - Ranges from 4 to 14 extra cycles per lock & unlock.
- This is true taking into account there are very few remote-CPU operations, meaning:
  - Most locks() are happening on a local, exclusive cacheline
  - There is not a relevant amount of contention happening

Wait!
PREEMPT_RT already turns local_lock() into spinlock()

Yeah, that's correct!

- Good! It means that there is already a success case in the local_lock() → spinlock() replacement :)
- But CPU Isolation is a feature that does not depend on PREEMPT_RT.
  - Would it be ok to have a CPU Isolation improvement that only gets enabled when PREEMPT_RT is enabled?
  - If so, there is a solution that:
    - Costs nothing
    - Improves CPU Isolation
    - Saves time during remote-CPU requests!

The other way

- If PREEMPT_RT already turns local_lock() into spinlock(), why it still requires a schedule_work_on(isolated_cpu) when accessing a remote per-CPU cache?
  - Couldn’t it just grab that remote per-CPU spinlock(), do the required work on the per-CPU struct, and then release it?
  - It already gets the cacheline exclusiveness when scheduling the work on that CPU, so that cost is already paid.
  - schedule_work_on(isolated_cpu) would only be required if there is any change that needs to be done in hardware, like changing a control register, or flushing hardware cache.

The other way : Proposal

- A previous patchset [3] proposes new helpers for local_lock() family (WIP, struggling with a better naming):
  - local_lock_n(s, cpu) / local_unlock_n(s, cpu):
    - PREEMPT_RT=n : Grab/release current CPU’s local_lock()
    - PREEMPT_RT=y : Grab/release percpu spinlock() for that cpu
  - local_schedule_work_on(work, cpu)
    - PREEMPT_RT=n: calls schedule_work_on(work, cpu) as expected.
    - PREEMPT_RT=y: grabs that cpu’s spinlock(), does the required work, then releases the spinlock()
  - local_flush_work(work) : Same idea, no-op in PREEMPT_RT=y

The other way : Example

```
void may_be_ran_remotely(int cpu) {
    local_lock_n(s, cpu);
    do_work();
    local_unlock_n(s, cpu);
}
void wont_be_ran_remotely() {
    local_lock(s);
    do_work_2();
    local_unlock(s);
}
```

```
void require_remote_work() {
    INIT_WORK(work, may_be_ran_remotely);

    for_each_online_cpu(cpu)
        local_queue_work_on(cpu, wq, work);

    /* Optional */
    for_each_online_cpu(cpu)
        local_flush_work(work);
}
```

Thanks!

Questions?
Suggestions?

References:

[1] https://lore.kernel.org/all/20220624125423.6126-8-mgorman@techsingularity.net/
[2] https://lore.kernel.org/all/Y+P2xp5BfmGh5Fin@P9FQF9L96D.corp.robot.car/
[3] https://lore.kernel.org/all/20230729083737.38699-2-leobras@redhat.com/