Improve this_cpu_*() ops Performance for Non-x86 Architectures

LSF/MM/BPF 2026

Yang Shi yang@os.amperecomputing.com

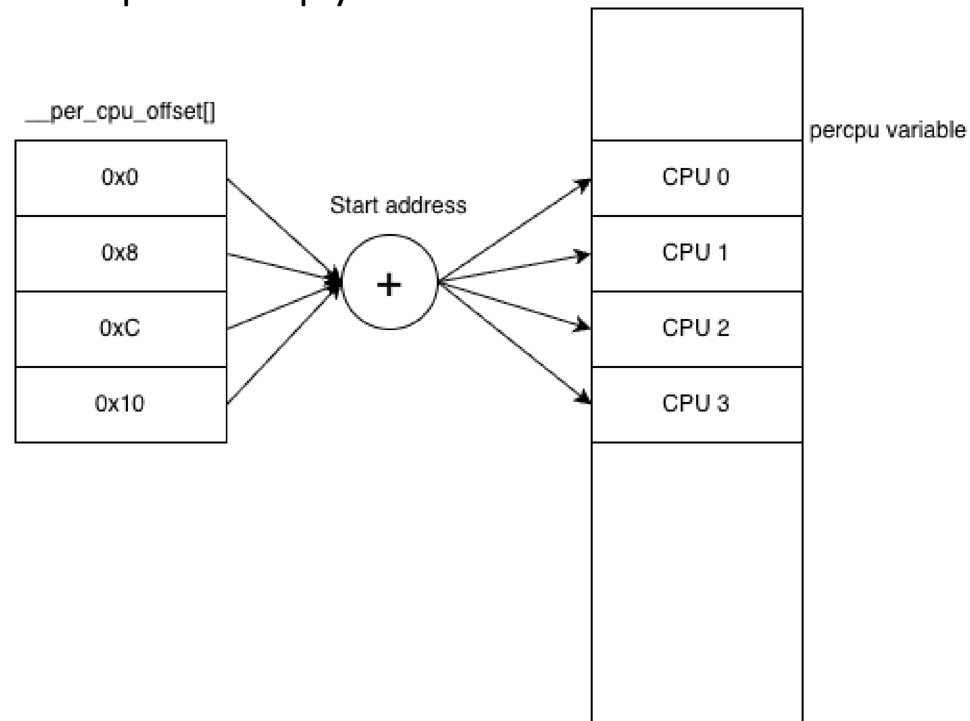
Christopher Lameter cl@gentwo.org

Agenda

- Problem
- Proposal
- Design
- Overhead and Limitations
- Performance Benchmark
- Future Usecases

Problem

- this_cpu_*() ops operate on a local copy of percpu variable for the current CPU
    - this_cpu_inc()/this_cpu_dec()/this_cpu_add()/this_cpu_sub(), etc
- A cpu specific offset (__per_cpu_offset[]) has to be added to the address in order to obtain the address of the current cpu's copy



Problem

- X86
  - the address can be calculated by prefixing an instruction with a segment register
  - x86 can manipulate a percpu counter with a single instruction, for example,

  inc %gs:[my_counter]

- non-x86
  - may not have segment registers (i.e. ARM64) or need expensive CPU mode switch (i.e. S390)
  - The address calculation must be atomic vs scheduler, which incurs extra overhead
  - The code flow looks like:
    - Disable preemption
    - Calculate the current CPU copy address by using the offset
    - Manipulate the counter
    - Enable Preemption

Proposal

- Need to guarantee this_cpu_*() APIs convert the offset returned by percpu allocator to a pointer which should be the same for all CPUs in order to remove preemption disable/enable, and the pointer maps to different physical memory depending on the CPU
  - offset: returned by percpu allocator APIs, for example, alloc_percpu()
  - pointer: converted by this_cpu_*() or per_cpu_ptr() from the offset
  - __per_cpu_offset[] is no longer needed for this_cpu_*()
- Don’t break per_cpu_ptr() APIs usecase either, which is mainly used to initialize percpu variables from one single CPU. It requires different addresses for each CPU’s copy
  - For example,

```
for_each_possible_cpu(cpu)
    *per_cpu_ptr(my_count, cpu) = 0;
```

Design

- Allocate **extra virtual memory** (1 CPU copy size) for “local mapping”
  - The “global mapping” is mainly used by per_cpu_ptr() APIs; the “local mapping” is mainly used by this_cpu_*() APIs
  - The virtual address of “local mapping” is same for all CPUs

Design

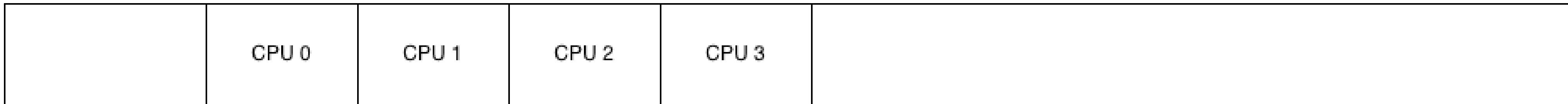
- Restriction for “local mapping” virtual memory allocation
  - alloc_percpu() returns an offset: addr – pcpu_base_addr (ignore __per_cpu_start to simplify it)
    - addr: “global mapping” address
    - pcpu_base_addr: “global mapping” base address initialized at boot stage
  - In order to use the same returned offset to access “local mapping”, kernel needs to set up “local mapping” base address and the “local mapping” allocation needs to guarantee:

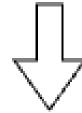
    laddr – local_pcpu_base == addr – pcpu_base_addr

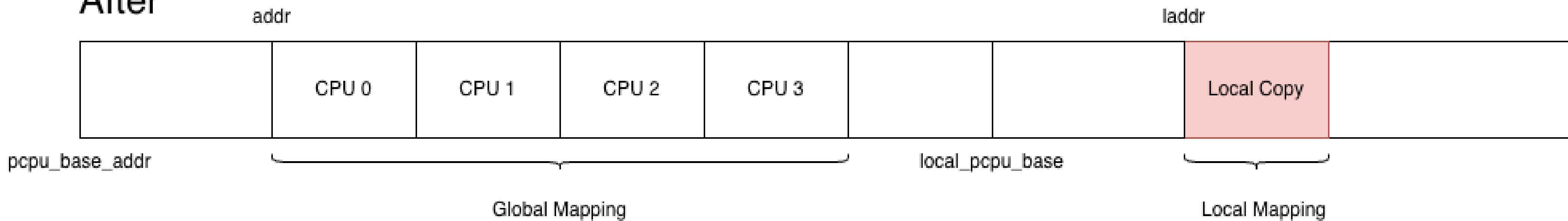
    - laddr: allocated “local mapping” address
    - local_pcpu_base: ”local mapping” base address
  - This requires “local mapping” allocation happens in a specific address range
  - This may require a dedicated percpu “local mapping” area in order to avoid conflicts with vmalloc()
  - No embed mode: may require percpu variables NOT be mapped in linear map address space

Design

- “local mapping” layout





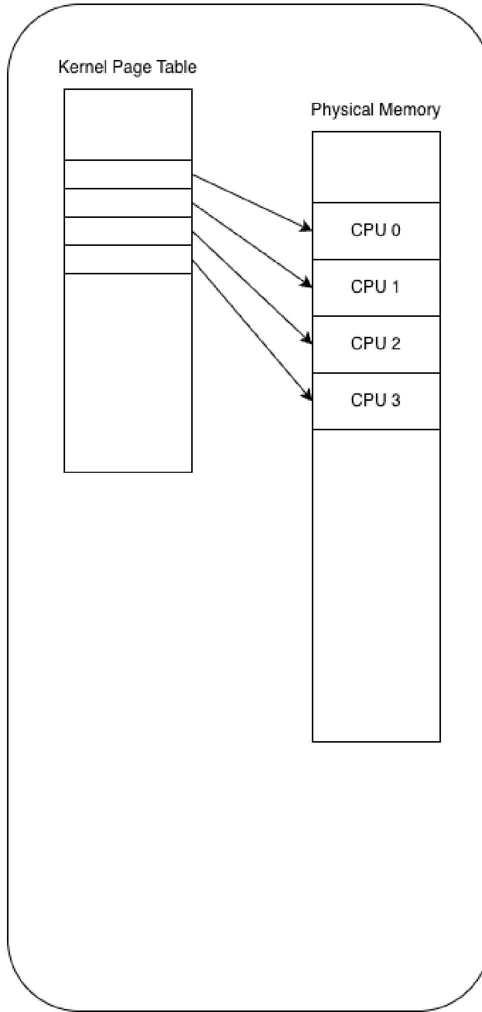


Design

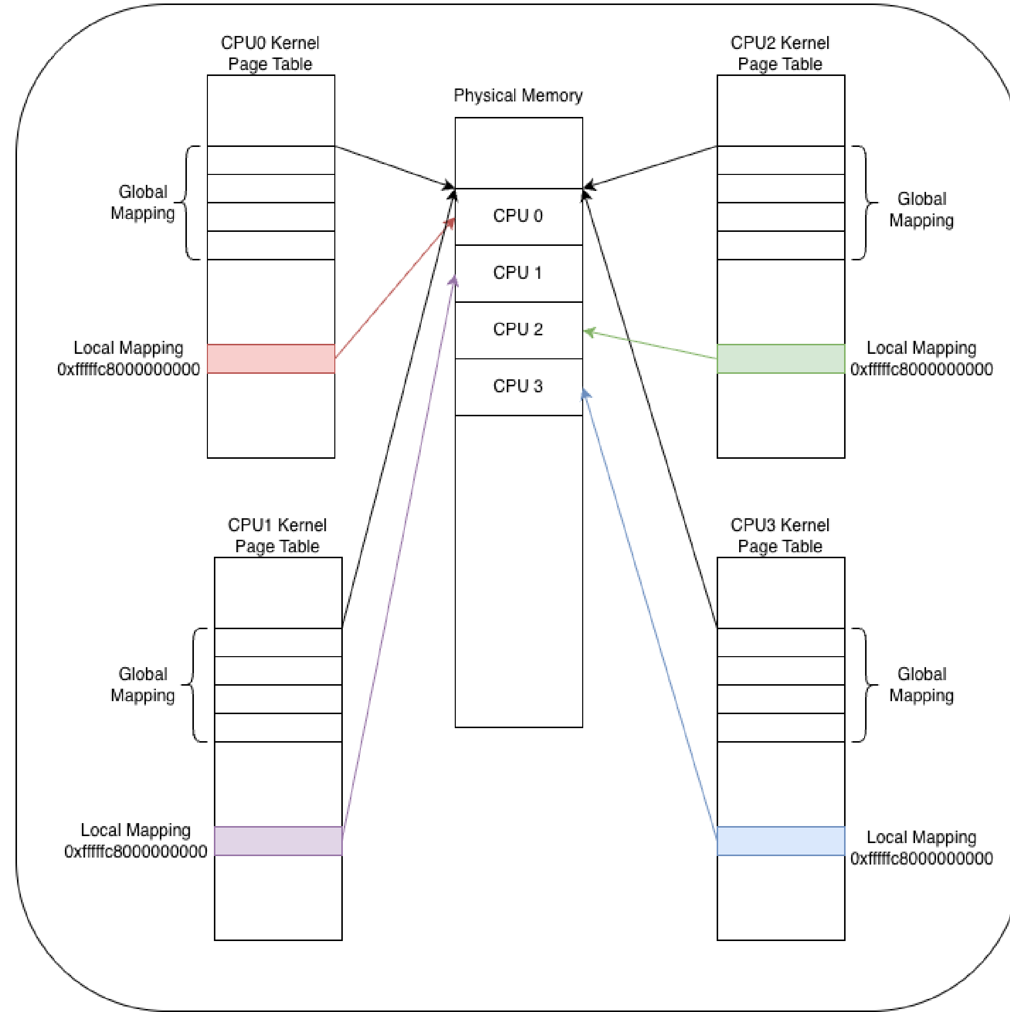
- The “local mapping” requires mapping to different physical memory on different CPUs in order to operate on the right data copy
  - This requires per cpu page tables
  - Each CPU sees its own kernel page table copy instead of sharing one single kernel page table
  - The most contents of the per cpu page tables can be shared except for the percpu “local mapping” area
  - CPU 0 uses swapper_pg_dir (init_mm.pgd) and all kernel page table operations (for example, vmalloc/vfree) happens on swapper_pg_dir except for percpu “local mapping” area
  - Need sync kernel page tables; can be optimized to just sync at PGD level

Design

Before



After



Design

- RFC patches
  - Implemented for ARM64
  - Minimum viable patches
  - Link: https://lore.kernel.org/linux-mm/20260429170758.3018959-1-yang@os.amperecomputing.com/

Overhead and Limitations

- Overhead
  - Some extra virtual memory space
    - Shouldn’t be too much. 960K was consumed with Fedora default kernel config. We have terabytes virtual address space on 64 bit machines
  - Some extra physical memory for per cpu page tables
    - 4K * (nr_cpus – 1) for PGD pages
    - Plus page tables used by percpu “local mapping” area
    - A couple of mega bytes with Fedora default config on 160 cores AmpereOne machine
    - More memory consumed with large page size (i.e. 16K and 64K)
  - Extra page table works for percpu allocation and free
    - Basically more CPUs more page table operations

Overhead and Limitations

- Limitations
  - May require dedicated virtual address space are for “local mapping” area, reduce the size of vmalloc area; so prefer 64 bit machines
  - May not work for shared TLB due to per cpu page tables
    - Some SMT machines may share TLB between hardware threads in the same core
    - Turning off CNP should solve it, need to measure the potential performance impact

Performance Benchmark

- Environment: 160 core AmpereOne 7.1-rc1 kernel 4K base page size
- Kernel build
  - Run kernel build (make –j160) with default Fedora kernel config in a memcg
  - 13% - 18% sys time improvement
  - 3% - 7% wall time improvement
- stress-ng vm ops
  - stress-ng --vm 160 --vm-bytes 128M --vm-ops 100000000
  - 6% - 8.5% improvement
- stress-ng vm ops + fork
  - stress-ng --mmapfork 160 --mmapfork-bytes 128M --mmapfork-ops 500
  - 15% improvement

Performance Benchmark

- Memcg regression test
  - Create 10K memcgs
    - Each memcg creation needs to allocate multiple percpu variables, for example, percpu refcnt, rstat and objcg percpu refcnt
  - Consumed 2112K more virtual memory for percpu “local mapping”
  - A few more mega bytes consumed by per cpu page tables
  - Elapsed time is basically same, no noticeable regression is found
- Fork regression test
  - rss stat counters and mm_cid_pcpu are percpu variables
  - stress-ng --fork 160 --fork-ops 10000000
  - 1% regression (the benchmark is not representative to real life workload due to small address space)
  - Stress-ng mmapfork benchmark sees significant improvement

Future Usecases

- Kernel text replication
    - Per cpu page tables make this could happen
    - Supposed to improve performance for large NUMA machines

Thank You

Backup

- Profiling for page_fault3_processes

  5.91% -1.82% [kernel.kallsyms] [k] mod_memcg_lruvec_state

  2.84% -1.30% [kernel.kallsyms] [k] percpu_counter_add_batch

- Profiling for memcg creation

  0.35% -0.33% [kernel.kallsyms] [k] percpu_ref_get_many

  0.61% -0.30% [kernel.kallsyms] [k] percpu_counter_add_batch

  0.34% +0.02% [kernel.kallsyms] [k] pcpu_alloc_noprof

  0.00% +0.05% [kernel.kallsyms] [k] free_percpu.part.0