# this_cpu operations

**Author:** Christoph Lameter, August 4th, 2014
**Author:** Pranith Kumar, Aug 2nd, 2014

this_cpu operations are a way of optimizing access to per cpu variables associated with the *currently* executing processor. This is done through the use of segment registers (or a dedicated register where the cpu permanently stored the beginning of the per cpu area for a specific processor).

this_cpu operations add a per cpu variable offset to the processor specific per cpu base and encode that operation in the instruction operating on the per cpu variable.

This means that there are no atomicity issues between the calculation of the offset and the operation on the data. Therefore it is not necessary to disable preemption or interrupts to ensure that the processor is not changed between the calculation of the address and the operation on the data.

Read-modify-write operations are of particular interest. Frequently processors have special lower latency instructions that can operate without the typical synchronization overhead, but still provide some sort of relaxed atomicity guarantees. The x86, for example, can execute RMW (Read Modify Write) instructions like inc/dec/cmpxchg without the lock prefix and the associated latency penalty.

Access to the variable without the lock prefix is not synchronized but synchronization is not necessary since we are dealing with per cpu data specific to the currently executing processor. Only the current processor should be accessing that variable and therefore there are no concurrency issues with other processors in the system.

Please note that accesses by remote processors to a per cpu area are exceptional situations and may impact performance and/or correctness (remote write operations) of local RMW operations via this_cpu_*.

The main use of the this_cpu operations has been to optimize counter operations.

The following this_cpu() operations with implied preemption protection are defined. These operations can be used without worrying about preemption and interrupts:

```
this_cpu_read(pcp)
this_cpu_write(pcp, val)
this_cpu_add(pcp, val)
this_cpu_and(pcp, val)
this_cpu_or(pcp, val)
this_cpu_add_return(pcp, val)
this_cpu_xchg(pcp, nval)
this_cpu_cmpxchg(pcp, oval, nval)
this_cpu_cmpxchg_double(pcp1, pcp2, oval1, oval2, nval1, nval2)
this_cpu_sub(pcp, val)
this_cpu_inc(pcp)
this_cpu_dec(pcp)
this_cpu_sub_return(pcp, val)
this_cpu_inc_return(pcp)
this_cpu_dec_return(pcp)
```

## Inner working of this_cpu operations

On x86 the fs: or the gs: segment registers contain the base of the per cpu area. It is then possible to simply use the segment override to relocate a per cpu relative address to the proper per cpu area for the processor. So the relocation to the per cpu base is encoded in the instruction via a segment register prefix.

For example:

```
DEFINE_PER_CPU(int, x);
int z;

z = this_cpu_read(x);
```

results in a single instruction:

```
mov ax, gs:[x]
```

instead of a sequence of calculation of the address and then a fetch from that address which occurs with the per cpu operations. Before this_cpu_ops such sequence also required preempt disable/enable to prevent the kernel from moving the thread to a different processor while the calculation is performed.

Consider the following this_cpu operation:

```
this_cpu_inc(x)
```

The above results in the following single instruction (no lock prefix!):

```
inc gs:[x]
```

instead of the following operations required if there is no segment register:

```
int *y;
int cpu;

cpu = get_cpu();
y = per_cpu_ptr(&x, cpu);
(*y)++;
put_cpu();
```

Note that these operations can only be used on per cpu data that is reserved for a specific processor. Without disabling preemption in the surrounding code this_cpu_inc() will only guarantee that one of the per cpu counters is correctly incremented. However, there is no guarantee that the OS will not move the process directly before or after the this_cpu instruction is executed. In general this means that the value of the individual counters for each processor are meaningless. The sum of all the per cpu counters is the only value that is of interest.

Per cpu variables are used for performance reasons. Bouncing cache lines can be avoided if multiple processors concurrently go through the same code paths. Since each processor has its own per cpu variables no concurrent cache line updates take place. The price that has to be paid for this optimization is the need to add up the per cpu counters when the value of a counter is needed.

## Special operations

```
y = this_cpu_ptr(&x)
```

Takes the offset of a per cpu variable (&x !) and returns the address of the per cpu variable that belongs to the currently executing processor. this_cpu_ptr avoids multiple steps that the common get_cpu/put_cpu sequence requires. No processor number is available. Instead, the offset of the local per cpu area is simply added to the per cpu offset.

Note that this operation is usually used in a code segment when preemption has been disabled. The pointer is then used to access local per cpu data in a critical section. When preemption is re-enabled this pointer is usually no longer useful since it may no longer point to per cpu data of the current processor.

## Per cpu variables and offsets

Per cpu variables have *offsets* to the beginning of the per cpu area. They do not have addresses although they look like that in the code. Offsets cannot be directly dereferenced. The offset must be added to a base pointer of a per cpu area of a processor in order to form a valid address.

Therefore the use of x or &x outside of the context of per cpu operations is invalid and will generally be treated like a NULL pointer dereference.

```
DEFINE_PER_CPU(int, x);
```

In the context of per cpu operations the above implies that x is a per cpu variable. Most this_cpu operations take a cpu variable.

```
int __percpu *p = &x;
```

&x and hence p is the *offset* of a per cpu variable. this_cpu_ptr() takes the offset of a per cpu variable which makes this look a bit strange.

## Operations on a field of a per cpu structure

Let's say we have a percpu structure:

```
struct s {
        int n,m;
};

DEFINE_PER_CPU(struct s, p);
```

Operations on these fields are straightforward:

```
this_cpu_inc(p.m)

z = this_cpu_cmpxchg(p.m, 0, 1);
```

If we have an offset to struct s:

```
struct s __percpu *ps = &p;

this_cpu_dec(ps->m);

z = this_cpu_inc_return(ps->n);
```

The calculation of the pointer may require the use of this_cpu_ptr() if we do not make use of this_cpu ops later to manipulate fields:

```
struct s *pp;

pp = this_cpu_ptr(&p);

pp->m--;

z = pp->n++;
```

## Variants of this_cpu ops

this_cpu ops are interrupt safe. Some architectures do not support these per cpu local operations. In that case the operation must be replaced by code that disables interrupts, then does the operations that are guaranteed to be atomic and then re-enable interrupts. Doing so is expensive. If there are other reasons why the scheduler cannot change the processor we are executing on then there is no reason to disable interrupts. For that purpose the following __this_cpu operations are provided.

These operations have no guarantee against concurrent interrupts or preemption. If a per cpu variable is not used in an interrupt context and the scheduler cannot preempt, then they are safe. If any interrupts still occur while an operation is in progress and if the interrupt too modifies the variable, then RMW actions can not be guaranteed to be safe:

```
__this_cpu_read(pcp)
__this_cpu_write(pcp, val)
__this_cpu_add(pcp, val)
__this_cpu_and(pcp, val)
__this_cpu_or(pcp, val)
__this_cpu_add_return(pcp, val)
__this_cpu_xchg(pcp, nval)
__this_cpu_cmpxchg(pcp, oval, nval)
__this_cpu_cmpxchg_double(pcp1, pcp2, oval1, oval2, nval1, nval2)
__this_cpu_sub(pcp, val)
__this_cpu_inc(pcp)
__this_cpu_dec(pcp)
__this_cpu_sub_return(pcp, val)
__this_cpu_inc_return(pcp)
__this_cpu_dec_return(pcp)
```

Will increment x and will not fall-back to code that disables interrupts on platforms that cannot accomplish atomicity through address relocation and a Read-Modify-Write operation in the same instruction.

## &this_cpu_ptr(pp)->n vs this_cpu_ptr(&pp->n)

The first operation takes the offset and forms an address and then adds the offset of the n field. This may result in two add instructions emitted by the compiler.

The second one first adds the two offsets and then does the relocation. IMHO the second form looks cleaner and has an easier time with (). The second form also is consistent with the way this_cpu_read() and friends are used.

## Remote access to per cpu data

Per cpu data structures are designed to be used by one cpu exclusively. If you use the variables as intended, this_cpu_ops() are guaranteed to be "atomic" as no other CPU has access to these data structures.

There are special cases where you might need to access per cpu data structures remotely. It is usually safe to do a remote read access and that is frequently done to summarize counters. Remote write access something which could be problematic because this_cpu ops do not have lock semantics. A remote write may interfere with a this_cpu RMW operation.

Remote write accesses to percpu data structures are highly discouraged unless absolutely necessary. Please consider using an IPI to wake up the remote CPU and perform the update to its per cpu area.

To access per-cpu data structure remotely, typically the per_cpu_ptr() function is used:

```
DEFINE_PER_CPU(struct data, datap);

struct data *p = per_cpu_ptr(&datap, cpu);
```

This makes it explicit that we are getting ready to access a percpu area remotely.

You can also do the following to convert the datap offset to an address:

```
struct data *p = this_cpu_ptr(&datap);
```

but, passing of pointers calculated via this_cpu_ptr to other cpus is unusual and should be avoided.

Remote access are typically only for reading the status of another cpus per cpu data. Write accesses can cause unique problems due to the relaxed synchronization requirements for this_cpu operations.

One example that illustrates some concerns with write operations is the following scenario that occurs because two per cpu variables share a cache-line but the relaxed synchronization is applied to only one process updating the cache-line.

Consider the following example:

```
struct test {
        atomic_t a;
        int b;
};

DEFINE_PER_CPU(struct test, onecacheline);
```

There is some concern about what would happen if the field 'a' is updated remotely from one processor and the local processor would use this_cpu ops to update field b. Care should be taken that such simultaneous accesses to data within the same cache line are avoided. Also costly synchronization may be necessary. IPIs are generally recommended in such scenarios instead of a remote write to the per cpu area of another processor.

Even in cases where the remote writes are rare, please bear in mind that a remote write will evict the cache line from the processor that most likely will access it. If the processor wakes up and finds a missing local cache line of a per cpu area, its performance and hence the wake up times will be affected.