Proceedings of the Linux Symposium

Volume Two

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Ho Hum, Yet Another Memory Allocator…

Do We Need Another Dynamic Per-CPU Allocator?

Ravikiran G Thirumalai
kiran.th@gmail.com
Dipankar Sarma
Linux Technology Center, IBM India Software Lab
dipankar@in.ibm.com
Manfred Spraul
manfred@colorfullife.com

Abstract

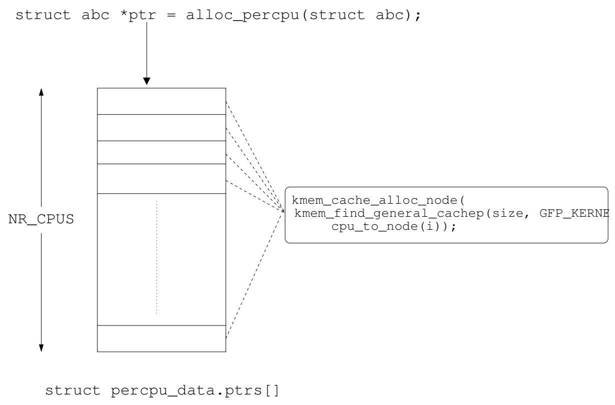
The Linux® kernel currently incorporates a minimalistic slab-based dynamic per-CPU memory allocator. While the current allocator exists with some applications in the form of block layer statistics and network layer statistics, the current implementation has issues. Apart from the fact that it is not even guaranteed to be correct on all architectures, the current implementation is slow, fragments, and does not do true node local allocation. A new per-CPU allocator has to be fast, work well with its static sibling, minimize fragmentation, co-exist with some arch-specific tricks for per-CPU variables and get initialized early enough during boot up for some users like the slab subsystem. In this paper, we describe a new per-CPU allocator that addresses all issues mentioned above, along with possible uses of this allocator in cache friendly reference counters (bigrefs), slab head arrays, and performance benefits due to these applications.

1 Introduction

The Linux kernel has a number of allocators, including the page allocator for allocating physical pages, the slab allocator for allocating objects with caching, and vmalloc allocator. Each of these allocators provide ways to manage kernel memory in different ways. With the introduction of symmetric multi-processing (SMP) support in the Linux kernel, managing data that are rarely shared among processors became important. While statically allocated per-CPU data has been around for a while, support for dynamic allocation of per-CPU data was added during the development of 2.6 kernel. Dynamic allocation allowed per-CPU data to be used within dynamically allocated data structures making it more flexible for users.

The dynamic per-CPU allocator in the 2.6 kernel was, however only the first step toward better management of per-CPU data. It was a compromise given that the use of dynamically allocated per-CPU data was limited. But with the need for per-CPU data increasing and support for NUMA becoming important, we decided to revisit the issue.

In this paper, we present a new dynamic per-CPU data allocator that saves memory by interleaving objects of the same processor, supports allocating objects from memory close to the CPUs (for NUMA platforms), works during early boot and is independent of the slab allocator. We also show it allows implementation of more complex synchronization primitives like distributed reference counters. We discuss some preliminary results and future course of action.



```
struct abc *ptr = alloc_percpu(struct abc);
```

```
kmem_cache_alloc_node(
 kmem_find_general_cachep(size, GFP_KERNE
     cpu_to_node(i));
```

Figure 1: Current allocator

2 Background

Over the years, CPU speed has been increasing at a much faster rate then speed of memory access. This is even more important in multi-processor systems where accessing memory shared between the processors could be significantly more costly if the corresponding cache line is not available in that processor's cache.

Operation	Cost (ns)
Instruction	0.7
Clock Cycle	1.4
L2 Cache Hit	12.9
Main Memory	162.4

Table 1: 700 MHz P-III Operation Costs

Table 1 shows the cost of memory operations on a 700 MHz Pentium$^{\text{TM}}$ III processor. When global data is shared between processors, cache lines bouncing between processors reduce memory bandwidth and thereby negatively impact scalability. As scalability improved during the development of the 2.6 kernel, the need for efficient management of infrequently shared data also increased. The first step towards this was interleaved static per-CPU areas proposed by Rusty Russell [3]. This allowed better management of cache lines by sharing them between CPU-local versions of different objects. As the need for per-CPU data increased beyond what could be declared statically, the first RFC for a dynamic per-CPU data allocator was proposed [6] along with a reference implementation [7].

Subsequent discussions led to a simplified implementation of a dynamic allocator in the 2.5 kernel as shown in Figure 1. This allocator provides an interface `alloc_percpu()` that returns a pointer cookie. The pointer cookie is the address of an array of pointers to CPU-local objects each corresponding to a CPU in the system. The array and the CPU-local objects are allocated from the slab. Simplicity was the most important factor with this allocator, but it clearly had a number of problems.

1. The slab allocations are no longer padded to cache line boundaries. This means that the current implementation would lead to false sharing.

2. An additional memory access (array of CPU-local object pointers) has a performance penalty, mostly due to associativity miss.

3. The array itself is not NUMA-friendly.

4. It wastes space.

We therefore implemented a new dynamic allocator that worked around the problems of the current one. This allocator is based on the reference implementation [7] published earlier. The key improvement has been the use of pointer arithmetics to determine the address of the CPU-local objects, which reduces dereferencing overhead.

3 Interleaved Dynamic Per-CPU Allocator

3.1 Design Goals

In order to address the inadequacies in the current per-CPU allocation schemes in the Linux kernel, a new allocator must do the following:

1. Fast pointer dereferencing to get to the per-CPU object
2. Allocate node local memory for all CPUs
3. Save on memory, minimize fragmentation, maximize cache line utilization
4. Work well with CPU hotplug and memory hotplug, sparse CPU numbers.
5. Get initialized early during boot
6. Independent of the slab allocator
7. Work well with its static sibling (static per-CPU areas)

A typical memory allocator returns a record (usually a pointer) that can be used to access the allocated object. A per-CPU allocator needs to return a record that can be used to access every copy of the object's private data to the corresponding CPU. In our allocator, this record is a cookie and the CPU-local versions of the allocated objects can be accessed using it. Dereferencing speeds are very important, since this is the fast path for all users of per-CPU data. The CPU-local versions of the object also need to be allocated from the memory nearest to the CPU on NUMA systems. Also, in order to avoid the overhead of an extra memory access in the current per-CPU data implementation, we needed to use pointer arithmetics to access the object corresponding to a given CPU. The pointer arithmetic should be simple and should use as few CPU cycles as possible.

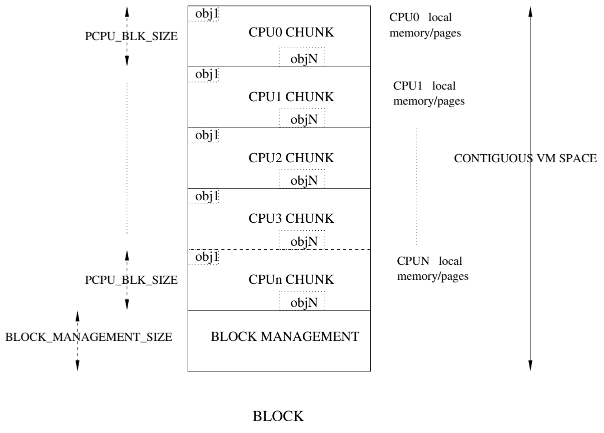


Figure 2: A block

3.2 Allocating a Block

The internal allocation unit of the interleaved per-CPU allocator is a `block`. Requests for per-CPU objects are served from a `block` of memory. The `block`s are allocated on demand for new per-CPU objects. A `block` is a contiguous virtual memory space (VA space) that is reserved to contain a chunk of objects corresponding to every CPU. It also contains additional space that is used to maintain internal structures for managing the blocks.

Figure 2 shows the layout of a `block`. The VA space within a `block` consists of two sections:

1. The top section consists of `NR_CPU` chunks of VA space each of size `PCPU_BLK_SIZE`. `PCPU_BLKSIZE` is a compile time constant. It represents the capacity of one `block`. Each CPU has one such per-CPU chunk within a `block`. Currently the size of each per-CPU chunk is two pages. `PCPU_BLKSIZE` is the size limit of a per-CPU object.

2. The bottom section of a `block` consists of memory used to maintain the per-CPU object buffer control information for this `block` and plus block descriptor size. This section is of size `BLOCK_MANAGEMENT_SIZE`.

While the VA for the entire `block` is allocated, the actual pages for each per-CPU chunk are allocated only if the corresponding CPU is present in the `cpu_possible_mask`. This has two benefits—it avoids unnecessary waste of memory and each chunk can be allocated so it is closest to the corresponding CPU. `alloc_page_node()` is used to get pages nearest to the CPU. The management pages at the bottom of a `block` are always allocated. Once the pages are allocated, VA space is then mapped with pages for the CPU-local chunks.

Also, as shown in Figure 3, there won't be mappings for any VA space corresponding to CPUs that are "not possible" on the system. The VA space is contiguous for `NR_CPUS` processors and this allows us to use pointer arithmetics to calculate the address of an object corresponding to a given CPU. We also save memory by not allocating for CPUs that are not in the `cpu_possible` mask.

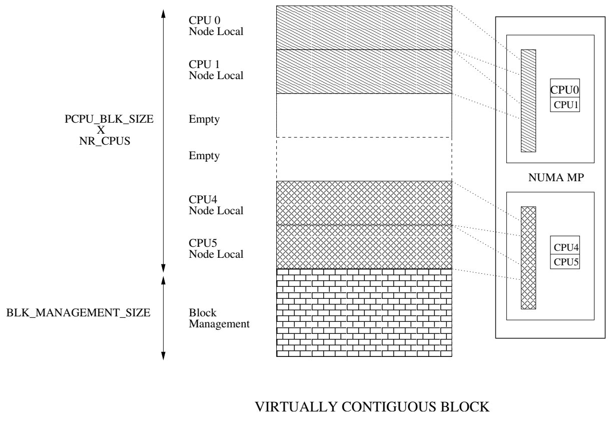


Figure 3: Page allocation for a block

3.3 Allocating Objects from a block

The per-CPU chunks inside a `block` are further divided into units of `currency`. A `currency` is the size of the smallest object that can be allocated in this scheme. The currency size is defined as `sizeof(void *)` in the current implementation. Any object in this allocator consists of one or more contiguous units of `currency`.

Each `block` in the system has a descriptor associated with it. The descriptor is defined as below:

```
struct pcpu_block {
    void *start_addr;
    struct page *pages[PCPUPAGES_PER_BLOCK * 2];
    struct list_head blklist;
    unsigned long bitmap[BITMAP_ARR_SIZE];
    int bufctl_fl[OBJS_PER_BLOCK];
    int bufctl_fl_head;
    unsigned int size_used;
};
```

This is embedded into the block management part of the block. In the current implementation, it is at the beginning of the block management section of a `block`. Each per-CPU object is allocated from one such `block` maintained by the interleaved allocator. The block descriptor records the base effective address of

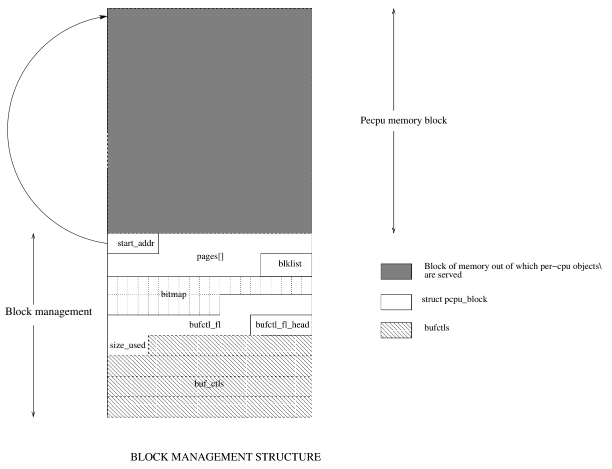


Figure 4: Managing blocks

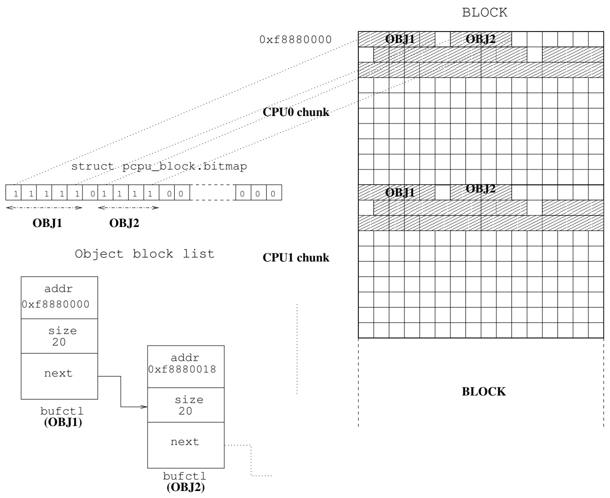


Figure 5: Object layout

the block (`start_addr`) as well as the allocation state of each `currency` within the `block`. The allocation state is recorded using a bitmap wherein each bit represents an isomorphic `currency` of every per-CPU chunk in that block. There are as many bits as the number of `currency` in one chunk of the block.

Figure 4 shows the organization of the block management area. Block descriptor has an array of pointers, each pointing to a CPU-local chunk of physical pages allocated for this block. Each object allocated from a `block` is represented by a `bufctl` data structure. These `bufctl` structures are embedded in the block management section of the `block` and they start right after the block descriptor. The block descriptor also has an array-based free list to allocate `bufctl` or object descriptors. `bufctl_fl` is the array-based free list and `bufctl_head` stores the head of this free list.

During allocation of a per-CPU object, the bitmap indicating `currency` allocation state is sorted and saved. This array is sorted in ascending order of available object sizes in that `block` due to contiguous `currency` regions. This array is traversed and the first element that fits the allocation requirement is used and the corresponding `currency` units are allocated.

Figure 5 shows the relation between a `block` in the allocator, per-CPU objects allocated from the `block`, the bitmap corresponding to these objects, `bufctl` structures and `bufctl` list for these objects. In this example, a per-CPU block starts at 0xf8880000. The currency size is assumed to be 4 (`sizeof (void *)` on x86). The squares in CPU chunks represent the allocator currency. The first five consecutive currencies make OBJ1 (shaded currencies in the block). Each currency is represented by a bit in the bitmap. Hence, bits 0–4 of the bitmap correspond to OBJ1. OBJ1 starts at address 0xf8880000. OBJ2 starts at 0xf8888018. The figure also depicts the bufctl structures and bufctl list for OBJ1 and OBJ2.

3.4 Managing blocks

The amount of per-CPU objects served by a single `block` is limited. So, our allocator allows allocation of new `block` on demand. Whenever a request for a per-CPU object cannot be met with any `block` currently in the system, a new `block` is added to the system that is isomorphic to existing ones.

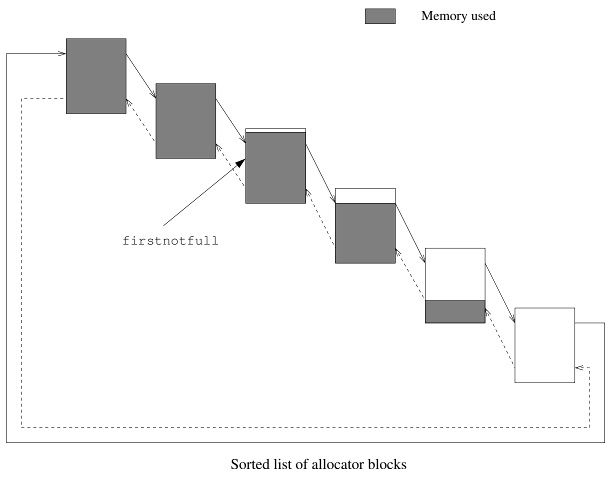


Figure 6: Managing block lists

These blocks are linked to one another in a circular doubly linked sorted list (Figure 6). `pcpu_block` counts the amount of memory used in the block (`size_used`). This list is sorted in descending order using `size_used`. The `firstnotfull` field contains the list position of the first `block` in the list that has available memory for allocation. On an allocation request, the list traversed from the `firstnotfull` position and the first available block with sufficient space is chosen for allocation. If no such block is found, a new block is created and added to this list. The blocks are repositioned in the list to preserve the sorted nature of the list upon every allocation and free request. During the course of freeing per-CPU objects, if the allocator notices that `blkp->size_used` goes to zero, the entire block—VA space, per-CPU pages, block management pages and the VA mapping are destroyed.

3.5 Accessing Per-CPU Data

A per-CPU allocation returns a pointer that is used as a cookie to access CPU-local version of the object for any given CPU. This pointer effectively points to address of the CPU-local object for CPU 0. To get to the CPU-local version of CPU N, the following arithmetics is used—`cpu_local_address = p + N * PCPU_BLKSIZE`, where `p` is the cookie returned by the interleaved allocator. Since `PCPU_BLKSIZE` is a carefully chosen compile time constant of a proper page order, the above arithmetics is optimized to a simple add and bit shift operations. The most expensive operation in accessing the CPU-local object is usually the determination of the current CPU number (`smp_processor_id()`). This is true for static per-CPU areas as well. To avoid the cost of `smp_processor_id()` during per-CPU data access, kernel developers like Rusty Russell have been contemplating using a dedicated processor register to get a handle to that processor's CPU-local data. The current static per-CPU area in the Linux kernel uses an array (`__per_cpu_offset[]`) to store a handle to each CPU's per-CPU data. With a dedicated processor register, `__per_cpu_offset[cpuN]` would be loaded into the register and users of per-CPU data would not need to make a call to `smp_processor_id()` to get to the CPU-local versions—simple arithmetic on the contents of the processor dedicated register will suffice. In fact, `smp_processor_id()` could be derived from the register based `__per_cpu_offset[]` table. This scheme can co-exist with our per-CPU allocator.

4 Using Dynamic Per-CPU Allocator

4.1 Per-CPU structures within the slab allocator

The Linux slab allocator uses arrays of object pointers to speed up object allocation and release. This avoids doing costly linked list or

```
struct kmem_cache_s {
    struct array_cache *array[NR_CPUS];
    /* ... additional members, only
     * touched from slow path ... */
};
struct array_cache {
    unsigned int avail;
    unsigned int limit;
    void * objects[];
};
```

Table 2: Main structures in the fast-path—before

spinlock operations in each operation. Each object cache contains one array for each CPU. If an array is not empty, then an allocation little more than looking up the per-CPU array and returning one entry from that array. Therefore the time required for the pointer lookup is the most significant part of the execution time for `kmem_cache_alloc` and `kmem_cache_free`.

At present, the lookup code mimics the implementation of the dynamic per-CPU variables: `kmem_cache_create` returns a pointer to the structure that contains the array of pointers to the per-CPU variables. Each allocation or release looks up the correct per-CPU structure and returns an object from the array. Table 2 shows the (slightly simplified) structures.

While this is a simple implementation, it has several disadvantages:

- It is a code duplication and it would be better if slab could reuse the primitives provided by the dynamic per-CPU variables. This is not possible, because it would create a cyclic dependency: the dynamic per-CPU variable implementation relies on the slab allocator for its own allocations.
- It is a simple per-CPU allocator, therefore each access required a table lookup. Depending on the value of `NR_CPUS`, there might be even frequent write operations, and thus cache line transfers on the cache line that contains the table.
- The implementation is fixed within `slab.c`, it's not possible to override it with arch specific code, even if an architecture supports a fast per-CPU variable lookup.

```
struct kmem_cache_s { /* per-CPU variable */
    struct kmem_globalcache *global;
    unsigned int avail;
    unsigned int limit;
    void * objects[];
};
struct kmem_globalcache { /* one instance */
    /* ... additional members, only
     * touched from slow path ... */
};
```

Table 3: Main structures for fast-path—after

Therefore the slab code was rearranged to use per-CPU variables natively for the object caches: `kmem_cache_create` returns the pointer to the per-CPU structure that contains the members that are needed in the fast-path of the allocator. The other members are stored in a new structure (`structkmem_globalcache`). The new structure layout is shown in Table 3.

The functions `kmem_cache_alloc()` and `kmem_cache_free()` only need to access `avail`, `limit`, and `objects`, thus there are no accesses to the global structure from the fast-path.

4.2 Statistics counters

As part of the scalable statistics counter work we carried out earlier, it has already been established that per-CPU data is useful for kernel statistics counters, and solves the problem of cache line bouncing on NUMA and multi-processor systems [5]. During the development of the 2.6 kernel, a number of kernel statistics were converted to use a dynamic per-CPU allocator. These include networking MIBs, disk statistics and the `percpu_counter` used in ext2 and ext3 filesystems. With our allocator, the per-CPU statistics counters become more efficient. In addition to faster dereferencing and node-local allocation, our allocator saves `NR_CPUS x sizeof(void *)` bytes of memory for each per-CPU counter by avoiding the array storing the CPU-local object pointers.

4.3 Distributed reference counters (bigrefs)

Rusty Russell has an experimental patch that makes use of the dynamic per-CPU memory allocator to avoid global atomic operations on reference counters[4].

A "bigref" reference counter would consist of two counters internally; one of type `atomic_t` which is the global counter, and another per-CPU counter of type `local_t`, the distributed counter. Per-CPU memory for the `local_t` is allocated when the bigref reference counter is initialized. The reference counter usually operates in the "fast" mode—it just increments or decrements the CPU-local `local_t` counter whenever the bigref reference counter needs to be incremented or decremented (`get()` and `put()` operations in Linux parlance). This operation on `local_t` per-CPU counters is much cheaper compared to operations on a global `atomic_t` type. In fact on x86, local increment is just an `incl` instruction.

The reference counter switches to a "slow" mode when the element being protected by the reference counter is no longer needed in the system and is being released or 'disowned'. This switch from fast mode to slow mode is done by using `synchronize_kernel()` to make sure all CPUs recognize slow mode operation before the 'disowning' completes. The reference counter is biased with a high value by setting the `atomic_t` counter with the high bias value before the switch to slow mode is initiated. In fact this biasing itself indicates beginning of the switch. This bias value is subtracted from the reference counter after the switch to slow mode.

Bigrefs save on space and dereference speeds when they use our per-CPU allocator. In addition to space saving, our allocator interlaces counters on cache lines too, which results in increased cache utilization.

5 Results

5.1 Slab enhancements

The new slab implementation discussed in Section 4.1 was tested with both micro-benchmarks and real-world test loads.

- Micro-benchmarks showed no change between the old and the new implementation; In a tight loop, `kmem_cache_alloc` needed around 35 CPU cycles on an 64-bit AMD Athlon^{TM}. The lack of improvement is not unexpected because a table lookup is only slow on a cache miss.
- Tests with tbench (version 3.03 with warm-up) on a 4-CPU HT Pentium 4 (2.8GHz Xeon) system showed an improvement of around 1%.

6 Future Work

The new allocator is not without its own limitations:

1. One major drawback of our allocator design is increased TLB footprint. Since our allocator uses the Linux vmalloc VM area to stitch all node local and block management pages into one contiguous block, hot per-CPU data may take too many TLB entries when there are too many allocator blocks within the system. Node-local page allocation and fast dereferencing are of utmost importance, so we have to use a virtually contiguous area for fast pointer arithmetic. But to limit the increased TLB usage, in the future, we may want to use large pages for blocks, and fit all per-CPU data in one block. This way, we can limit the number of TLB entries taken up by the dynamic per-CPU allocator.

2. The allocation operation is slow. It is not designed for allocation speed. It is designed for maximum utilization and minimum fragmentation. Given the current users of dynamic per-CPU allocator, it may not be valuable to improve allocation operation.

With the interleaved dynamic per-CPU allocator, it has also become possible to implement distributed locks [1] [2] and reference counters [4].

7 Conclusion

The interleaved per-CPU allocator we implemented is a step forward from where we are in the Linux kernel. The current allocator in the Linux kernel leads to false sharing and it is not optimized. Our allocator overcomes all of those problems. As the Linux kernel matures, this will allow use of more sophisticated primitives in the Linux kernel without adding any overhead. The design has evolved over a number of discussions and it is mature enough to handle all architectures.

8 Acknowledgments

Rusty Russell started the ball rolling by first implementing the static per-CPU areas and later advising us. We thank him for all of his contributions. We would also like to thank a number of Linux kernel hackers, including Dave Miller and Andrew Morton, all of whom advised us in many different situations. We are indebted to Gerrit Huizenga, Paul McKenney, Jim Wasko Jr., and Vijay Sukthankar for being so supportive of this effort.

9 Legal Statement

References

[1] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *Transactions of Computer Systems 9*, 1 (February 1991), 21–65.

[2] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the Third PPOPP* (Williamsburg, VA, April 1991), pp. 106–113.

[3] RUSSELL, R. Subject: [patch] 2.5.1-pre5: per-cpu areas. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=100759080903883&w=2`, December 2001.

[4] RUSSELL, R. Subject: Re: [patch] mm: Reimplementation of dynamic percpu memory allocator. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=110569951013489&w=2`, January 2005.

[5] SARMA, D. Scalable statistics counters. Available: `http://lse.sourceforge.net/counters/statctr.html`, May 2002.

[6] SARMA, D. Subject: [lse-tech] [rfc] dynamic percpu data allocator. Available: `http://marc.theaimsgroup.com/?l=lse-tech&m=102215919918354&w=2`, May 2002.

[7] SARMA, D. Subject: [rfc][patch] kmalloc_percpu. Available: `http://marc.theaimsgroup.com/?l=linux-kernel&m=102761936921486&w=2`, July 2002.