Reworking of KVA allocator in Linux kernel

Linux Kernel Engineer
Sony Mobile(Lund, Sweden)
Vlad Rezki
urezki@gmail.com

Lisbon-2019

Reworking of KVA allocator in Linux kernel

- Motivation
- Special requirements for the KVA allocator
- Current allocation scheme
- Current allocation scheme drawbacks
- New allocation scheme
- Performance analysis
- Performance test results
- Contribution
- Todo-list

Motivation

1. High demand in big data
2. Work-loads which are critical to time and latency

- audio/video/8K high resolution/5G areas(mobile segment)
- KVA is getting more and more used nowadays in the kernel
    - filesystems, kernel stacks, BPF, percpu, fork path, drivers, etc
    - new kvmalloc()/kvfree() interface introduced in 2017
        - If the slab fails(due to big size request)
        - fallback to vmalloc(bypassing the OOM killer)

Motivation(cont.)

Initiative of improving KVA allocator comes from getting many **issues** with **allocation time**, simply saying, sometimes it is terribly slow. As a result many workloads are affected by that slowness:

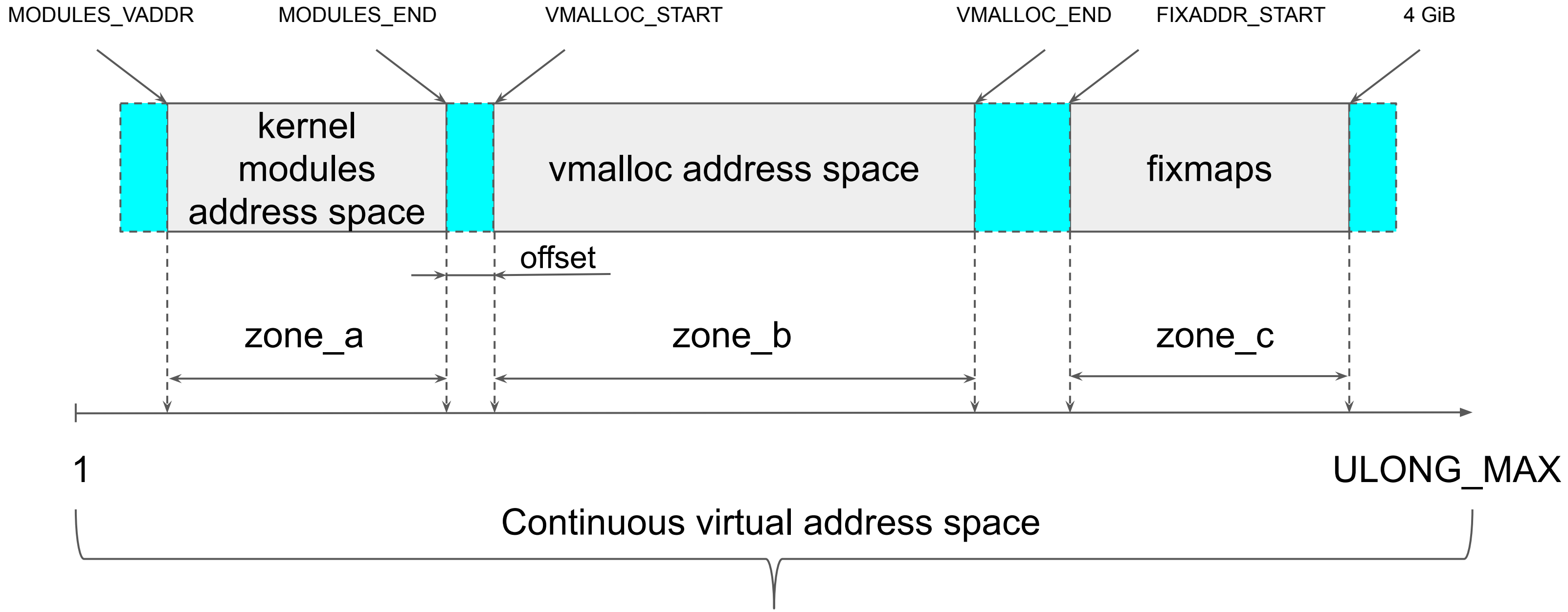
- Bluetooth audio skips
- Framedrops in UI and video playback
- Application launch times and etc.

Special requirements for the KVA allocator

- Support zone allocations in KVA space
- Sequential allocation to maximize locality
- Minimize external fragmentation

Support zone allocations in KVA space

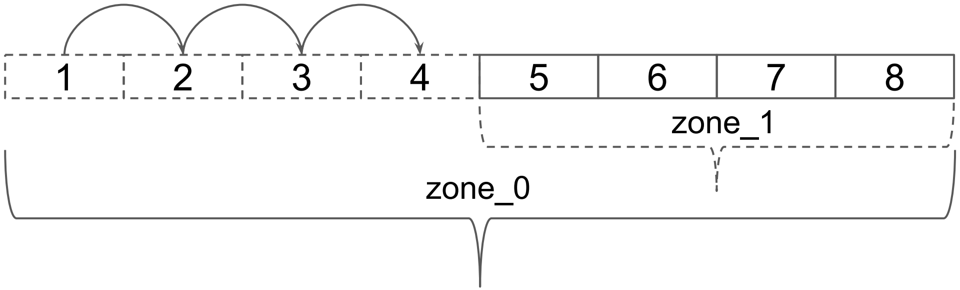
See the picture and explanation below:



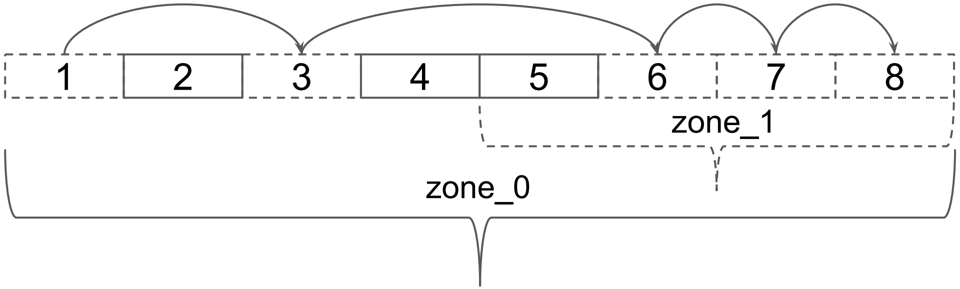
Sequential allocation to maximize locality

- There is at least one important issue if an allocation is not sequential
  - Waste of free space in a specific zone(if included into another one)

a) Sequential allocation in zone_0



b) Random allocation in zone_0



Minimize external fragmentation

- *Reduce implementation overhead.* It is wasted of memory for the internal data structures of the allocator implementation and bookkeeping.
- *Satisfy an allocation request*. External fragmentation occurs when free blocks of memory are available for allocation but they are too small.
- *Improve allocation time.* Due to high number of internal objects an allocation time usually gets increased.

Current allocation scheme (high level)

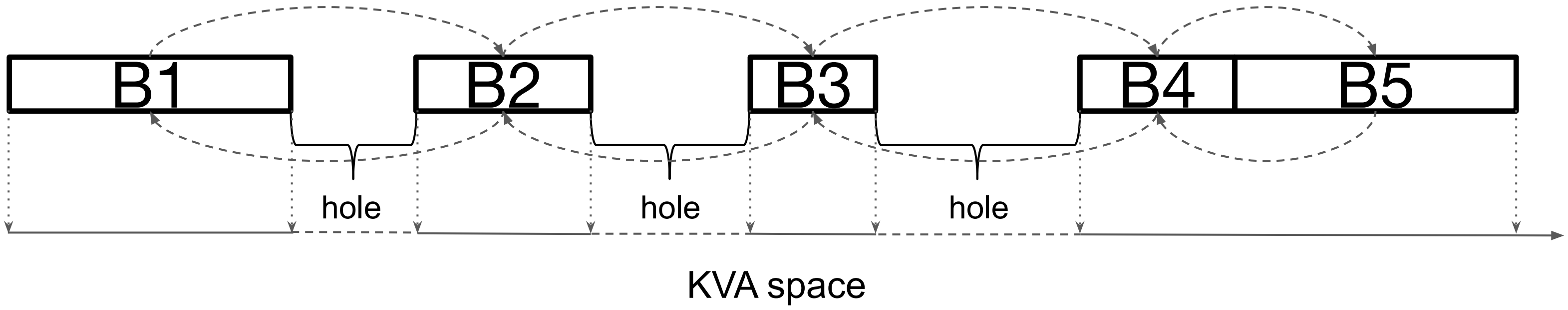
This allocator uses a double linked list containing busy blocks. Also, those blocks are sorted by the red-black tree. The tree allows to find a start address of required zone where an allocation has to be done.

To allocate a new memory block the search is done over **busy list iteration** until a suitable hole is found between two busy areas.

Therefore, each time a new allocation **occurs** internal data structures of the allocator get increased.

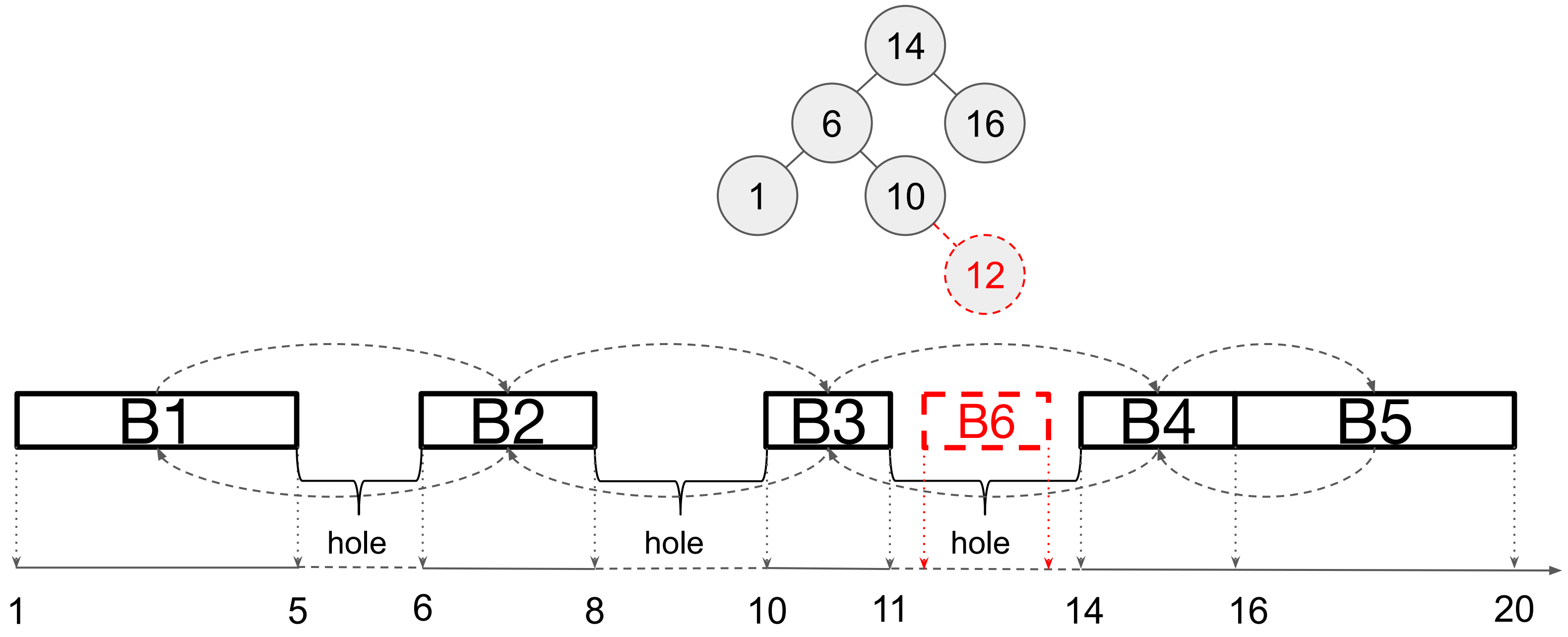
Current allocation scheme(high level cont.)

As an example let's consider 5 allocated memory blocks: B1, B2, B3, B4, B5 and three holes: F1, F2, F3. In order to allocate a new block we have to iterate over the list(B1-B5) checking a hole size between, until a fitting base is found:



Current allocation scheme(high level cont.)

The red-black tree is maintained to have a fast access to allocated earlier object when it is deallocated(not limited to it).



Current allocation scheme drawbacks

There are two main issues with current method:

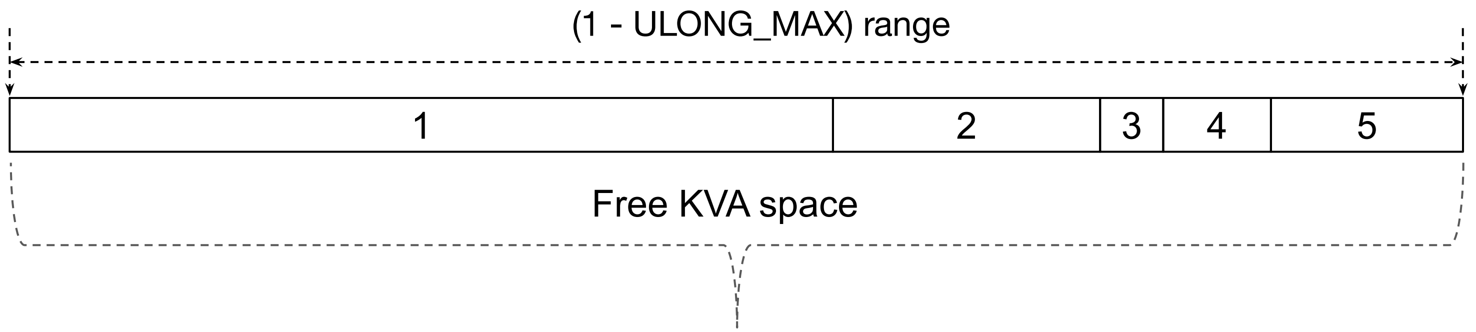
- It has O(N) complexity
- Due to external fragmentation and different permissive parameters an allocation can take a long time**(milliseconds)**.

New allocation scheme

- Allocate from free blocks(is built during early boot)
- The new allocation method uses an augment **red-black** tree
- All free blocks are sorted in ascending order by the tree
- Linked list is used for O(1) access to prev/next
  - When deallocate
    - Find a spot(tree traversal)
    - Fast merge with prev/next nodes
- Nodes are **augmented** with the size of maximum available block in its left or right subtree
- Complexity: **~O(log(N))**

New allocation scheme(cont.)

During initializing phase the KVA memory layout is organized into one free area that has 1 - ULONG_MAX range(can be more and depends on ARCH).



Here we have 5 free blocks with different sizes which are sorted in order of increasing addresses. That is just example.

New allocation scheme(cont.)

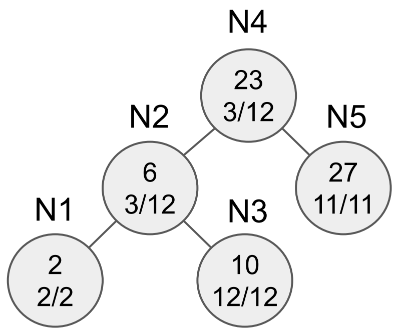
N1 - starts from 2, size is 2, max subtree size is 2

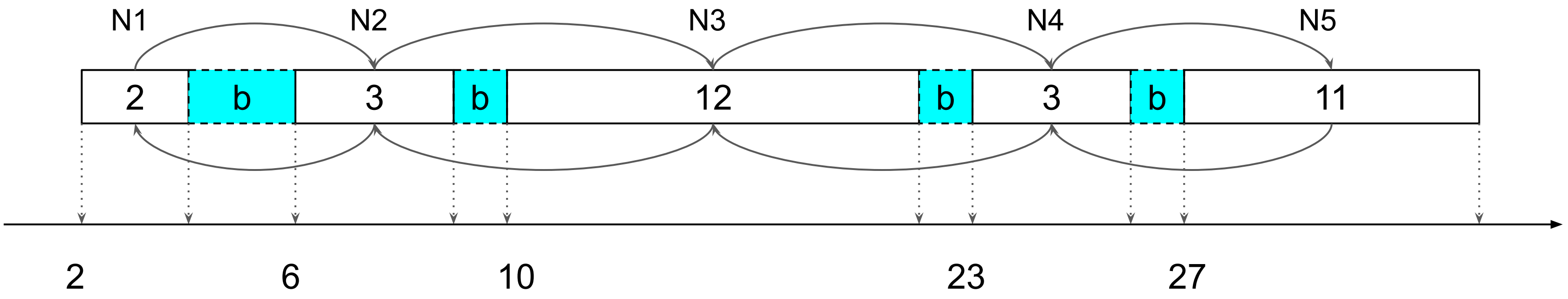
N2 - starts from 6, size is 3, max subtree size is 12

N3 - starts from 10, size is 12, max subtree size is 12

N4 - starts from 23, size is 3, max subtree size is 12

N5 - starts from 27, size is 11, max subtree size is 11



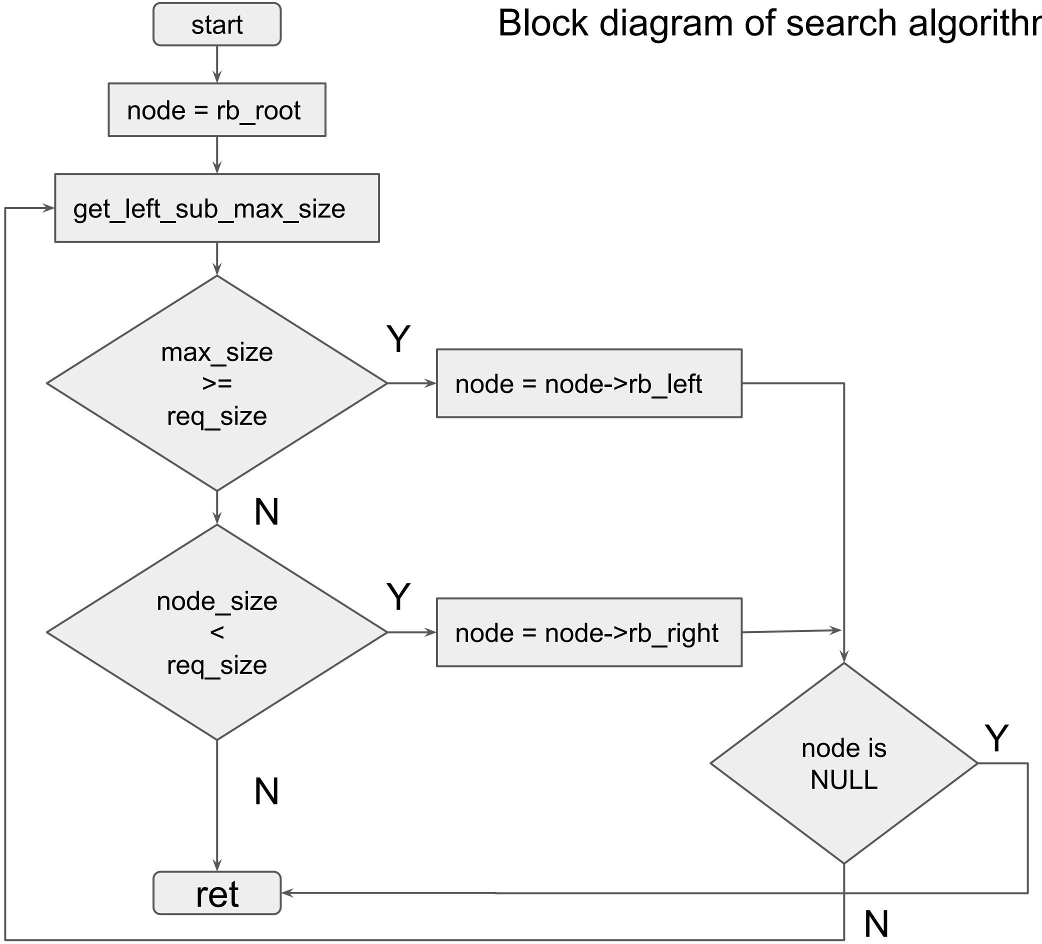


New allocation scheme(cont.)

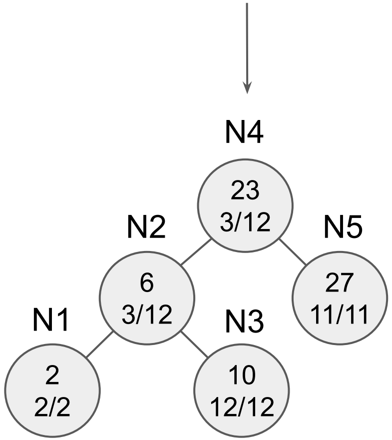
Allocation

- Start tree traversal from the root node
- Check left subtree max size
- Follow the left subtree **if** request is <= available size
- Go toward the block that fits
- When the block is found - it is split(3 cases)
    - LE_FIT/RE_FIT
    - FL_FIT
    - NE_FIT

Block diagram of search algorithm

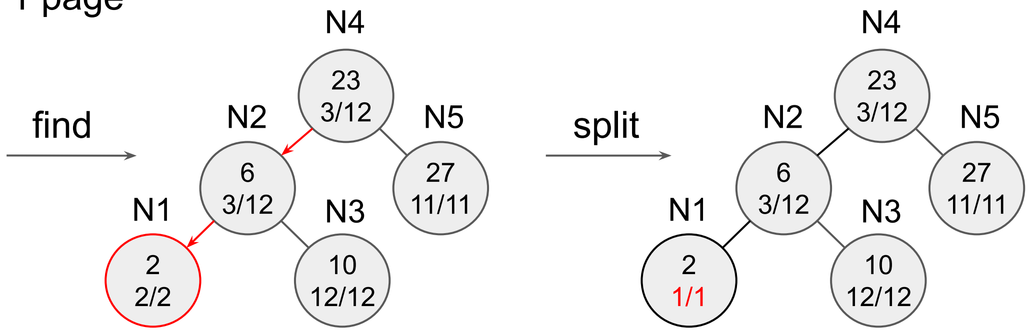


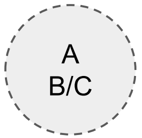
search parameters



New allocation scheme(cont.)

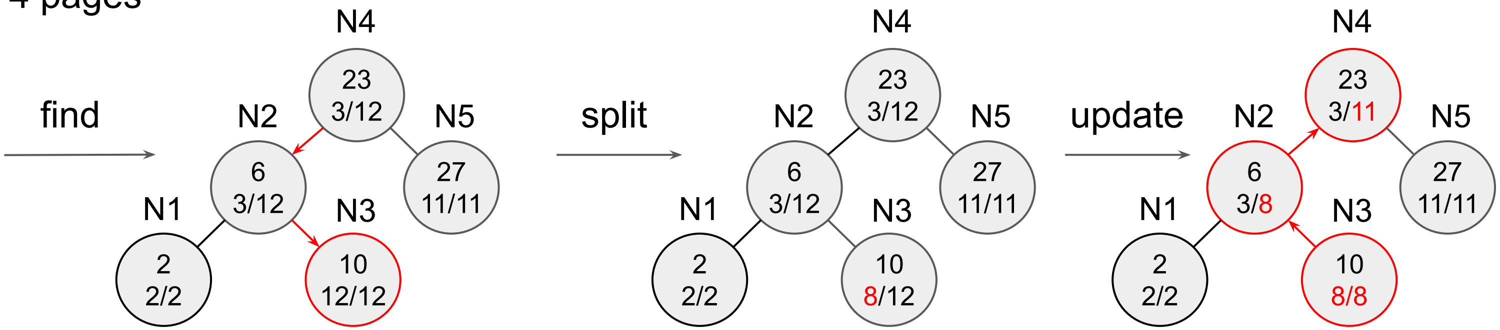
a) allocate 1 page





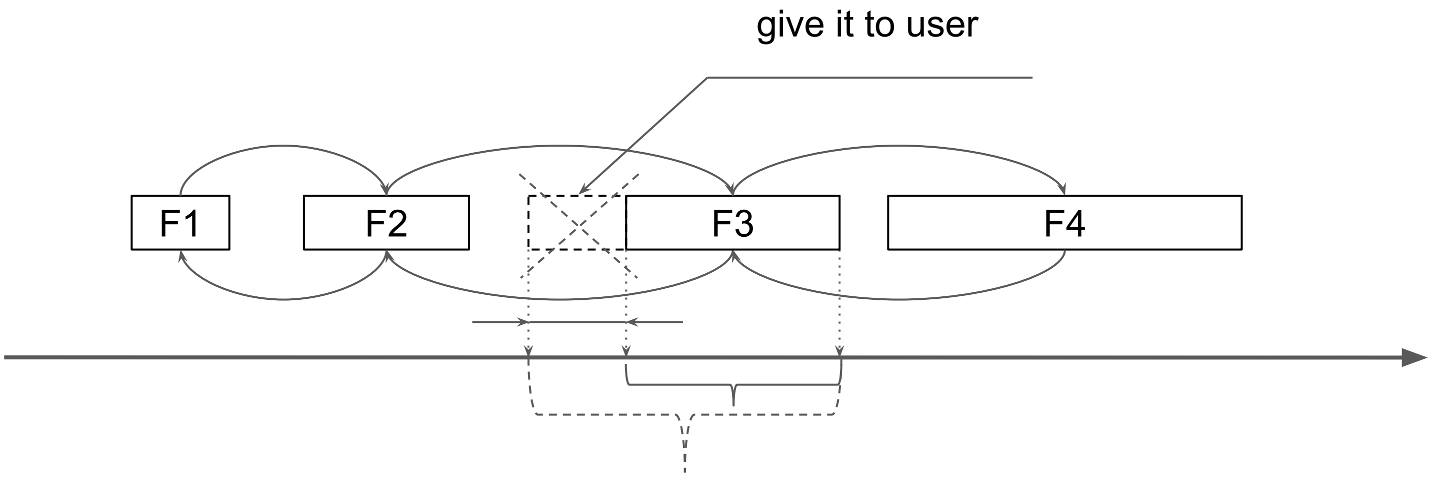
A - block start address
B - block size
C - subtree max size

b) allocate 4 pages



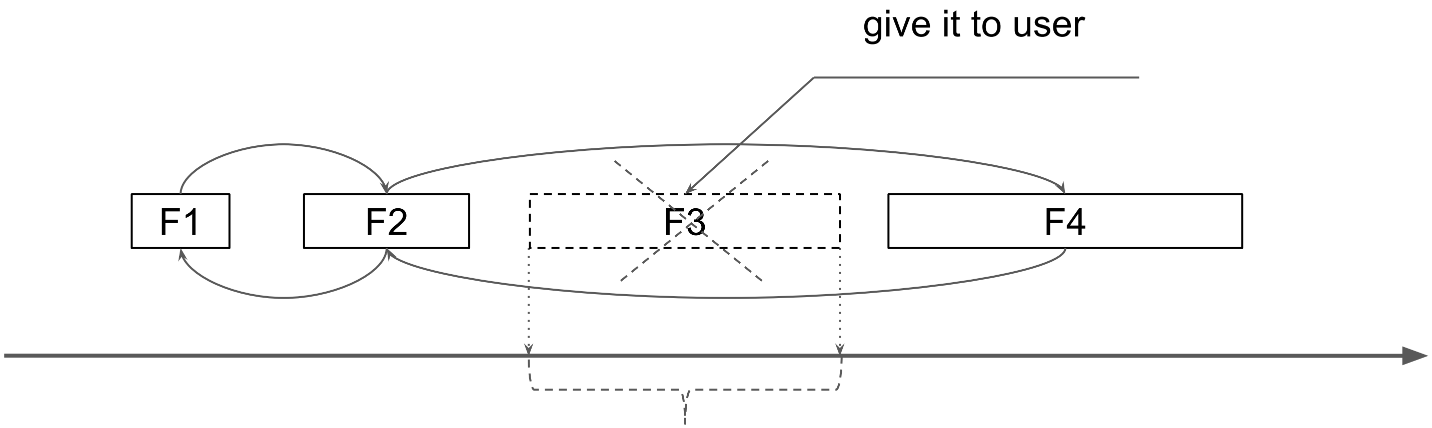
New allocation scheme(cont.)

First case: Requested size is 3 PAGES. If F1/F2 are small and F3 is bigger than 3 PAGES, we just shrink F3 to remaining size.



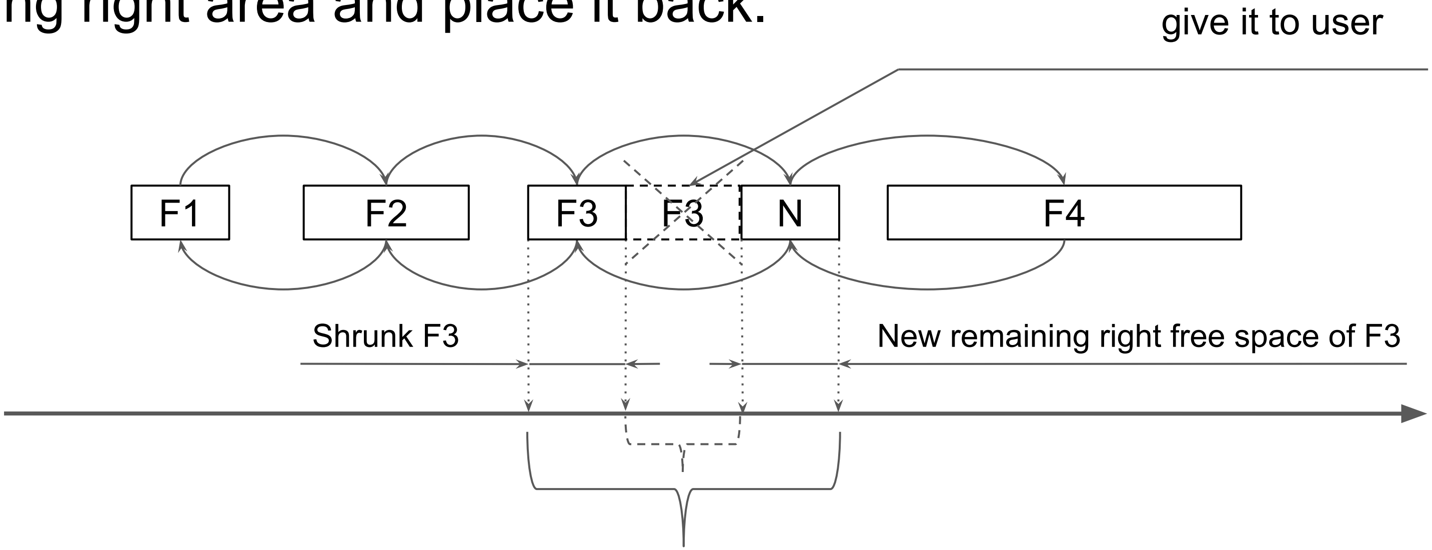
New allocation scheme(cont.)

Second case: Requested size is 3 PAGES. If F1/F2 are small and F3’s size is 3 PAGES, we just remove F3 from our internal data structures.



New allocation scheme(cont.)

Third case: Requested size is 3 PAGES. If F1/F2 are small, F3 is bigger than 3 PAGES and the requested size and alignment does not fit left nor right edges. In this case during splitting we build a new remaining right area and place it back.



New allocation scheme(cont.)

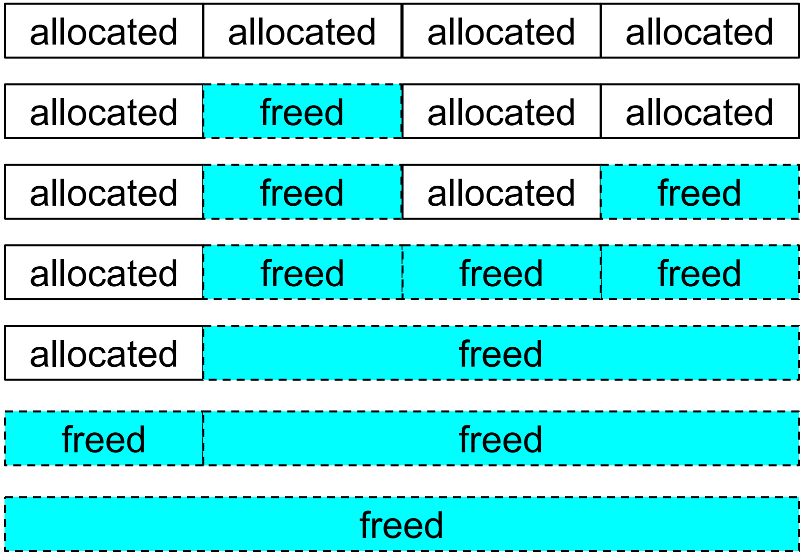
Summarizing. A “subtree-max-size” is populated back(upper levels) when block:

- is split(allocation path);
- is inserted to the tree(free path);
- is increased(merging path).

Please note that, it does not mean that upper parent nodes and their “subtree-max-size” are recalculated all the time up to the root node.

New allocation scheme(cont.)

De-allocation: red-black tree allows efficiently find a spot in the tree whereas a linked list allows fast merge of de-allocated memory chunks with existing free blocks creating large coalesced areas.



Performance analysis

- Developed special microbenchmark to analyse impact
- Available since 5.1 kernel
- Integrated with kernel self-tests
- Available under **tools/testing/selftests/vm/**
- The name is “test_vmalloc.sh”
- Is a kernel module
- The test driver has two modes
  - Performance analysis mode
  - Stressing mode

Performance test results

I use the **test_vmalloc.sh** that can simulate random allocations on all CPUs.
Please have a look at time taken by my **i5-3320M** machine to complete the test:

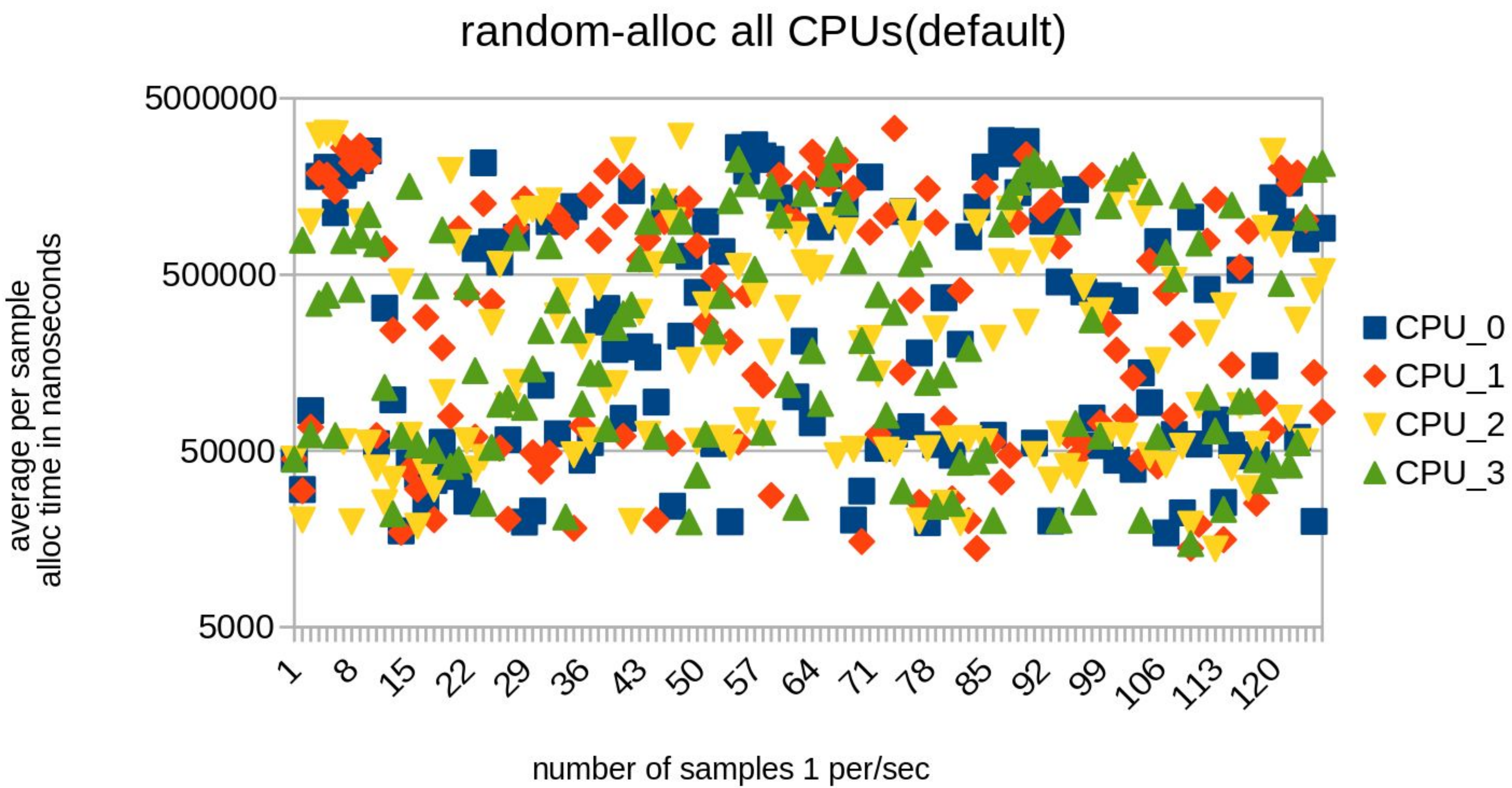
Default

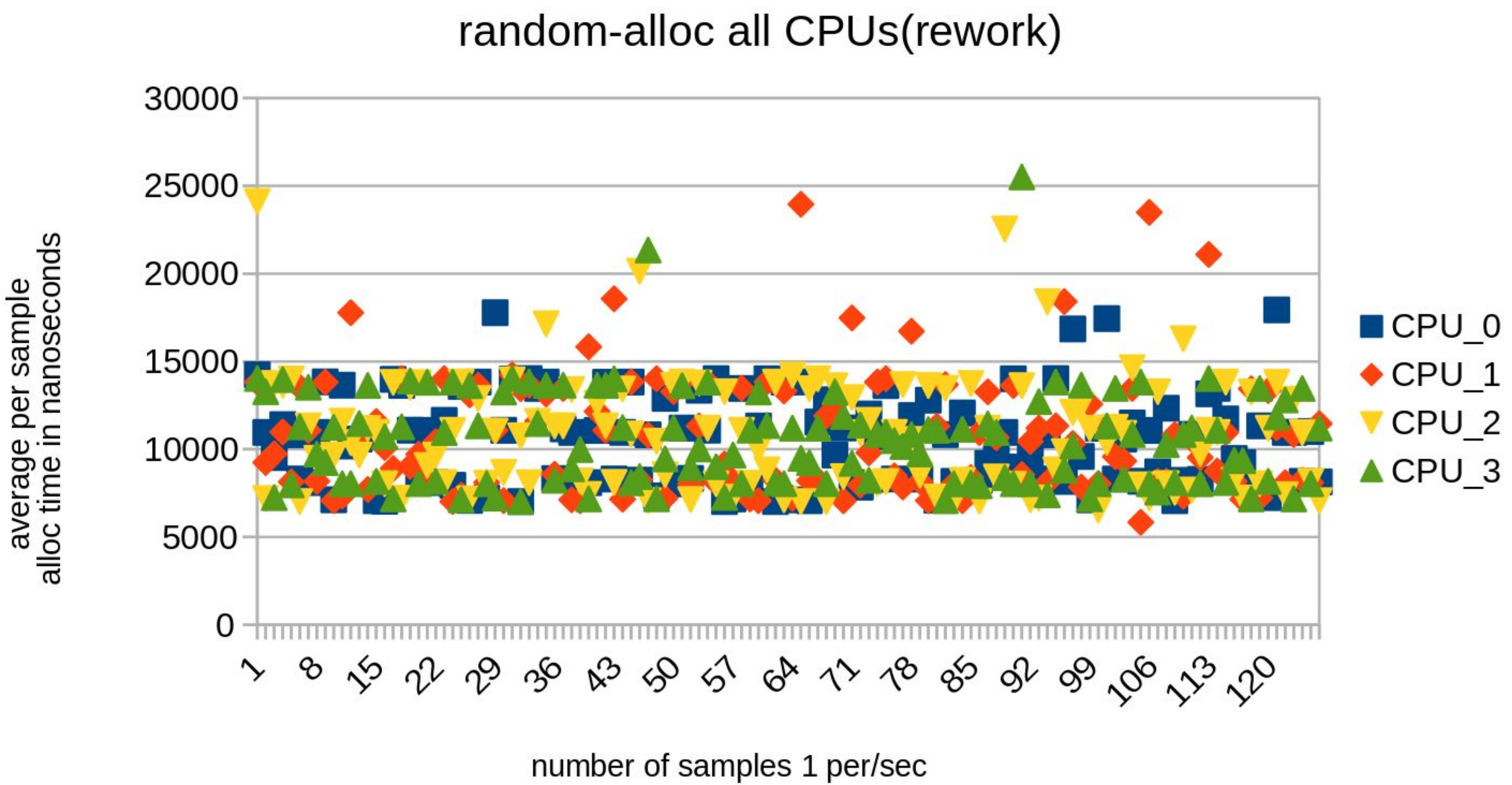
```
urezki@pc637:~$ time sudo ./test_vmalloc.sh test_repeat_count=1
   116m58.38s real     0m00.09s user     0m00.00s system
urezki@pc637:~$
```

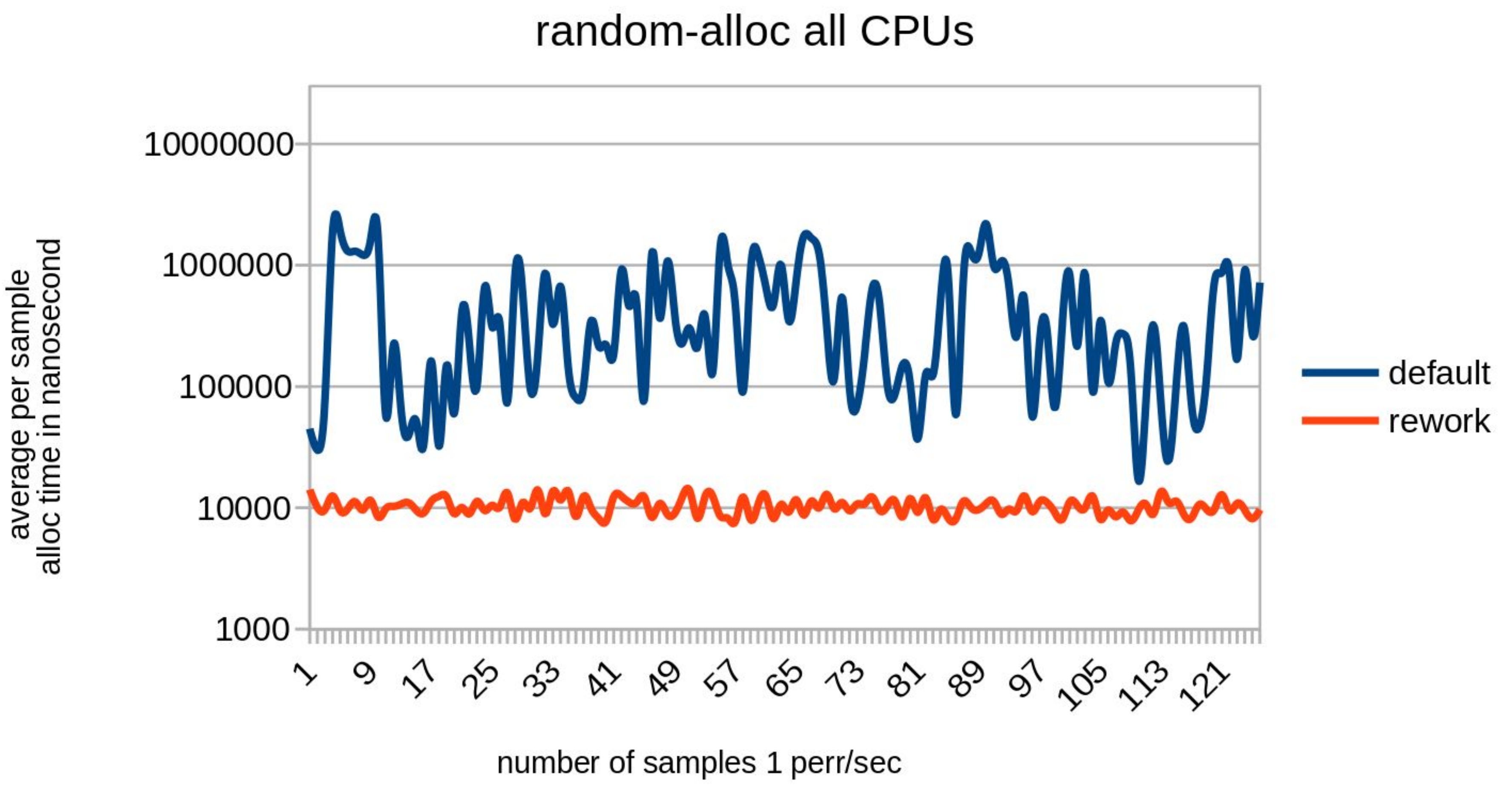
Rework

```
urezki@pc638:~$ time sudo ./test_vmalloc.sh test_repeat_count=1
   3m37.78s real     0m00.02s user     0m00.00s system
urezki@pc638:~$
```

116 minutes against 3 minutes. Rework ~39 times faster!


random-alloc all CPUs(default)
5000000
500000
50000
5000
average per sample
alloc time in nanoseconds
1
8
15
22
29
36
43
50
57
64
71
78
85
92
99
106
113
120
number of samples 1 per/sec
CPU_0
CPU_1
CPU_2
CPU_3


random-alloc all CPUs(rework)
30000
25000
20000
15000
10000
5000
0
average per sample
alloc time in nanoseconds
1
8
15
22
29
36
43
50
57
64
71
78
85
92
99
106
113
120
number of samples 1 per/sec
CPU_0
CPU_1
CPU_2
CPU_3


random-alloc all CPUs
average per sample
alloc time in nanosecond
10000000
1000000
100000
10000
1000
1
9
17
25
33
41
49
57
65
73
81
89
97
105
113
121
number of samples 1 perr/sec
default
rework

Contribution

Vmalloc benchmark and stress-test suite is in 5.1:

https://git.kernel.org/pub/scm/linux/kernel/git/next/linux-next.git/commit/?id=153178edc7819b5c550e5d498d50697ff9d5f223

https://git.kernel.org/pub/scm/linux/kernel/git/next/linux-next.git/commit/?id=3f21a6b7ef207892841feecc3b9216e1a29c745f

https://git.kernel.org/pub/scm/linux/kernel/git/next/linux-next.git/commit/?id=a05ef00c97900f69f6e69d88e8a657b7a4ef8cbd

https://git.kernel.org/pub/scm/linux/kernel/git/next/linux-next.git/commit/?id=6bc3fe8e7e172d5584e529a04cf9eec946428768

Stability fixes are in 5.1(was found by test driver):

https://git.kernel.org/pub/scm/linux/kernel/git/next/linux-next.git/commit/?id=afd07389d3f4933c7f7817a92fb5e053d59a3182

https://git.kernel.org/pub/scm/linux/kernel/git/next/linux-next.git/commit/?id=3319f8b3a38be63ff5bd31368a6996dfde0efab9

https://git.kernel.org/pub/scm/linux/kernel/git/next/linux-next.git/commit/?id=287819acc18b30c528d1c76b5b54e28e42ee54cc

Contribution(cont.)

The new KVA rework is in 5.2:

https://github.com/torvalds/linux/commit/a6cf4e0fe3e740ed7af39fdda721e1ac12247dd3#diff-1662e6f7a8ab98f610f1f19d89b78c9f

https://github.com/torvalds/linux/commit/bb850f4dae4abb18c5ee727bb2d6df9ca47ede49#diff-1662e6f7a8ab98f610f1f19d89b78c9f

https://github.com/torvalds/linux/commit/68ad4a3304335358f95a417f2a2b0c909e5119c4#diff-1662e6f7a8ab98f610f1f19d89b78c9f

https://github.com/torvalds/linux/commit/4d36e6f8040486f5945a3ba8a741eafe9d1d023a#diff-1662e6f7a8ab98f610f1f19d89b78c9f

https://github.com/torvalds/linux/commit/68571be99f323c3c3db62a8513a43380ccefe97c#diff-1662e6f7a8ab98f610f1f19d89b78c9f

https://github.com/torvalds/linux/commit/afd07389d3f4933c7f7817a92fb5e053d59a3182#diff-1662e6f7a8ab98f610f1f19d89b78c9f

https://github.com/torvalds/linux/commit/153178edc7819b5c550e5d498d50697ff9d5f223#diff-1662e6f7a8ab98f610f1f19d89b78c9f

...

Todo-list

Reduce lock contention

- Get rid of one global spin lock
    - split the **vmap_area_lock** to
        a. “busy tree” protection(allocated areas)
        b. “free tree” protection(free space)
        c. “lazily-freed” areas protection

Because of new approach the splitting is possible since a vmap_area object can only be in one of the three different states: ***a, b, c***

Todo-list(cont.)

Reduce lock contention(cont.)

- To use more efficient data structure
  - B-tree for organizing free memory layout
  - Splay-tree
  - etc.
- To implement “lazy” tree fixups
- Cache last accessed node to optimize traversal

Intel(R) Xeon(R) W-2135 CPU @ 3.70GHz 12xCPUs

```
23060734@seldlx26551:~# ./test_vmalloc.sh sequential_test_order=1&
23060734@seldlx26551:~# perf top -a -U
```

```
82.58%  [kernel]        [k] native_queued_spin_lock_slowpath
 1.85%  [kernel]        [k] alloc_vmap_area
 1.43%  [kernel]        [k] clear_page_erms
 1.26%  [kernel]        [k] _raw_spin_lock
 1.17%  [kernel]        [k] get_page_from_freelist
 1.12%  [kernel]        [k] __alloc_pages_nodemask
 0.78%  [kernel]        [k] insert_vmap_area.constprop.49
 0.75%  [kernel]        [k] vunmap_page_range
 0.66%  [kernel]        [k] vmap_page_range_noflush
 0.61%  [kernel]        [k] find_vmap_area
 0.59%  [kernel]        [k] free_vmap_area_noflush
 0.56%  [kernel]        [k] remove_vm_area
 0.43%  [kernel]        [k] _extract_crng
 0.41%  [kernel]        [k] rb_erase
 0.39%  [kernel]        [k] __free_pages
 0.39%  [kernel]        [k] __purge_vmap_area_lazy
 0.36%  [kernel]        [k] memset_erms
 0.35%  [kernel]        [k] free_unref_page
 0.25%  [kernel]        [k] chacha_permute
```

```
<annotate native_queued_spin_lock_slowpath>
       │     test    %eax,%eax
       │   ↓ jne     18d
       │   rep_nop():
 72.63 │184:  pause
       │   __read_once_size():
  9.95 │     mov     0x8(%rdx),%eax
       │   native_queued_spin_lock_slowpath():
  0.01 │     test    %eax,%eax
  0.62 │   ↑ je      184
       │   __read_once_size():
<annotate native_queued_spin_lock_slowpath>
```

Reworking of KVA allocator in Linux kernel

Linux Kernel Engineer
Sony Mobile(Lund, Sweden)
Vlad Rezki
urezki@gmail.com

Lisbon-2019