HugeTLB Improvement to Reduce Post-Boot Allocation Failures

Joshua Nicolas Verburg-Sachs

2016-02

Contents

1 Preamble

This research was inspired by many frustrated attempts to utilize the excellent huge page facilities of the Linux kernel during runtime. Huge pages are a mechanism whereby the Linux kernel can allocate memory not in 4kb pages (or whatever your distribution uses) but page sizes ranging from many megabytes to a gigabyte or more (though we touch only on the megabyte sized pages in this paper). This gives a variety of advantages, one of the most important being a reduction in the number of page entries that the kernel must manage for your executable. This can, for example, significantly increase the speed of large mmaps and munmaps.

Most of the current use cases and documentation stipulate that you should only attempt to allocate huge pages immediately after boot, in order to avoid (very common) failures due to memory fragmentation and memory use by other executables. However this is not always a convenient option and in some situations (such as real time systems with uptime requirements) it is basically untenable.

Therefore, this paper will describe a method whereby the Linux kernel can be improved to more robustly allocate huge pages during runtime, even when other applications have consumed a significant amount of available memory.

2 Amble

There are two primary causes of hugepage allocation failure. In order to illuminate them and discuss the solution we will describe the original algorithm here:

```
967 static int alloc_fresh_huge_page(struct hstate *h, nodemask_t *nodes_allowed)
968 {
969         struct page *page;
970         int nr_nodes, node;
971         int ret = 0;
972
973         for_each_node_mask_to_alloc(h, nr_nodes, node, nodes_allowed) {
974                 page = alloc_fresh_huge_page_node(h, node);
975                 if (page) {
976                         ret = 1;
977                         break;
978                 }
979         }
980
981         if (ret)
982                 count_vm_event(HTLB_BUDDY_PGALLOC);
983         else
984                 count_vm_event(HTLB_BUDDY_PGALLOC_FAIL);
985
986         return ret;
987 }
```

As originally written, the algorithm iterates over the given allowed nodes, calling alloc_fresh_huge_page_node which eventually calls into __alloc_pages_nodemask in page_alloc.c, where the kernel employs a variety of tools to attempt to enough memory to fulfill the request from this exact node.

...set_max_huge_pages...

```
1574                 spin_unlock(&hugetlb_lock);
1575                 if (hstate_is_gigantic(h))
1576                         ret = alloc_fresh_gigantic_page(h, nodes_allowed);
1577                 else
1578                         ret = alloc_fresh_huge_page(h, nodes_allowed);
1579                 spin_lock(&hugetlb_lock);
1580                 if (!ret)
1581                         goto out;
```

...set_max_huge_pages...

set_max_huge_pages is called by the kernel to allocate the total requested

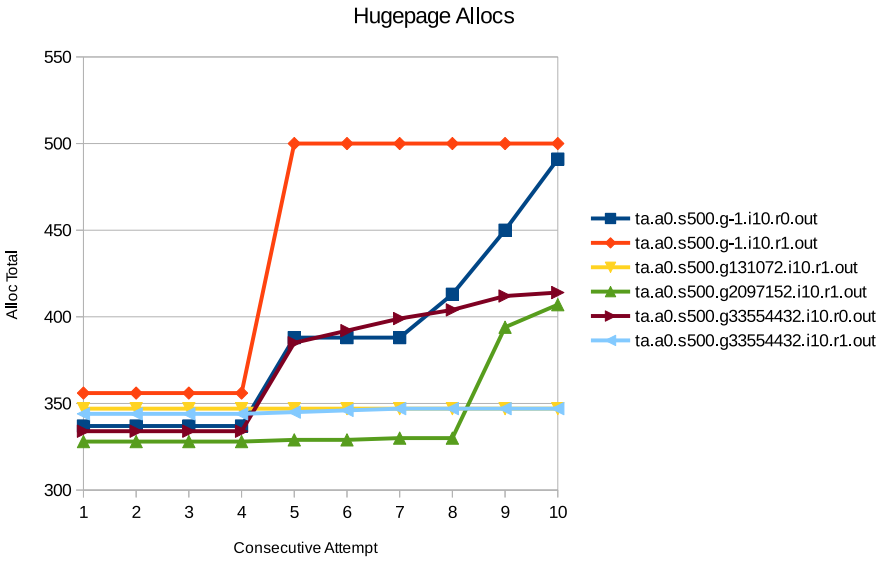
number of pages. Note that if we fail even a single call of alloc_fresh_huge_page, we immediately end the attempt to allocate. This can sometimes be worked around by asking the kernel repeatedly to allocate the number of pages, but the allocation will be biased towards the initial nodes searched since we iterate linearly, and eventually they will be exhausted. A situation will quickly arise where the allocation of further huge pages is not limited by available memory but instead by specific nodes which cannot be be passed over by the allocation algorithm. Additionally, the code in page_alloc.c primarily checks the freelist and then eventually wakes the kswapds. However if the system is under load, this is unlikely to be able to provide enough to fulfill a significant request.

3 Mosey

To demonstrate the issues with the original algorithm, as above, we provide two figures which describe 6 different scenarios for the default and new allocation method (hereafter called aggressive). The scenarios are a set of configuration options for the test run in question, where a test run is a certan amount of setup and then an allocation request of the kernel. All scenarios were run directly after a complete reboot. These scenarios are encoded as the first letter of each option, followed by a number, and are as follows:

- Aggressive: Whether to use aggressive alloc.
- Set: The number of pages to allocate.
- Grab: Whether to cause the test application to allocate memory (to disrupt hugepage allocation), and how much if so. -1 means none was allocated.
- Iterations: How many times the test application should attempt to set the number of huge pages. In all test runs this was 10, as it proved sufficient for demonstrating performance.
- Reset: Whether to re-set the number of hugepages to 0 after each iteration. 0 means do not reset.

As this chart shows, the default allocation strategy often struggles to fulfill a large huge page request at runtime. There are three key points on the graph to examine. The first is the initial allocation which for all attempts achieved roughly 350 pages of the 500 requested. The next key point is on consecutive attempt four, when 3 of the scenarios achieved a slightly higher allocation. The three in question are the two g-1s and the g33554432r0. Only the g-1r1 scenario achieves 500 at this point, and from thereafter the memory allocation succeeds, due to transparent hugepages keeping the pages available. The third key point is the 10th and final allocation attempt. At this juncture g-1r0 has almost, though not quite, fulfilled the 500 page request. g33554432r0 has slowly accumulated more pages but is only marginally greater than it was 6 attempts ago, and



g2097152r1 has almost caught up with g33554432r0. Only the scenarios where no additional memory was taken by the test application actually achieved near 500 pages, and only two other allocations even achieved more than 350 pages allocated. When they did, it still took a large number of repeated requests to achieve only a small fraction more memory than the initial fulfillment. In order to demonstrate the efficacy of the proposed changes to the Linux kernel, we will now examine the proposed changes and then the data for the test scenarios.

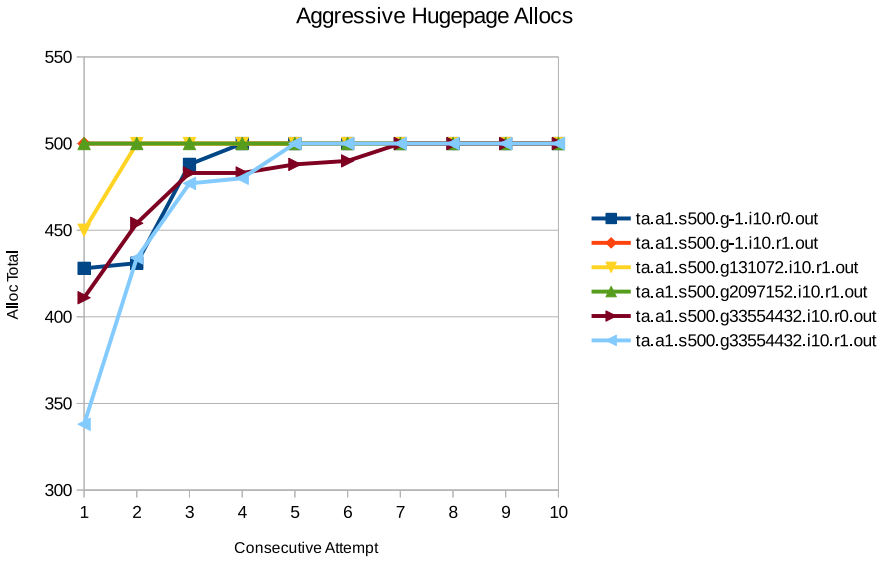
4 Wander

There are two important changes to the alloc_fresh_huge_page algorithm. First, if we completely fail to allocate any pages, we call try_to_free_pages (which frees up pages on that node which can be swapped out). Second, if we succeed in freeing pages then we re-attempt the allocation on that node. Additionally, we made a change to the calling algorithm, set_max_huge_pages, where we will retry until either we succeed in fulfilling the allocation request or we fail at allocating any huge pages twice in a row.

```
1013 static unsigned int alloc_fresh_huge_page(struct hstate *h, nodemask_t *nodes_al
1014 {
1015     struct page *page = NULL;
1016     int nr_nodes, node, ret = 0, num_nodes = 0;
1017
1018     for_each_node_mask_to_alloc(h, nr_nodes, node, nodes_allowed) {
1019         num_nodes++;
1020         page = alloc_fresh_huge_page_node(h, node);
1021         if (page) {
1022             goto out;
1023         }
1024     }
1025
1026     if (!hugepages_aggressive_alloc) {
1027         goto out;
1028     }
1029     for_each_node_mask_to_alloc(h, nr_nodes, node, nodes_allowed) {
1030         gfp_t gfp_mask = htlb_alloc_mask|__GFP_COMP|__GFP_THISNODE|
1031                 __GFP_REPEAT|_GFP_NOWARN;
1032         struct zonelist *zonelist = node_zonelist(node, gfp_mask);
1033
1034         if (try_to_free_pages(zonelist, h->order, gfp_mask, nodes_allowed) > 0)
1035             page = alloc_fresh_huge_page_node(h, node);
1036             if (page) {
1037                 count_vm_event(HTLB_BUDDY_PGALLOC_RETRY_SUCCESS);
1038                 break;
1039             }
1040             count_vm_event(HTLB_BUDDY_PGALLOC_RETRY_FAIL);
1041         }
1042     }
1043  out:
1044     if (page) {
1045         count_vm_event(HTLB_BUDDY_PGALLOC);
1046         ret = 1;
1047     } else {
1048         count_vm_event(HTLB_BUDDY_PGALLOC_FAIL);
1049         ret = 0;
1050     }
1051     return ret;
1052 }
```

```
1643                    spin_unlock(&hugetlb_lock);
1644                    if (hstate_is_gigantic(h)) {
1645                            ret = alloc_fresh_gigantic_page(h, nodes_allowed);
1646            } else {
1647                            ret = alloc_fresh_huge_page(h, nodes_allowed);
1648            }
1649            spin_lock(&hugetlb_lock);
1650            if (hstate_is_gigantic(h) || !hugepages_aggressive_alloc) {
1651                if (!ret)
1652                    goto out;
1653            } else {
1654                /* If two successive allocs failed, bail */
1655                if (last_ret == 0) {
1656                    if (!ret) {
1657                        count_vm_event(HTLB_BUDDY_PGALLOC_SUBSEQ_FAIL);
1658                        goto out;
1659                    } else {
1660                        count_vm_event(HTLB_BUDDY_PGALLOC_SUBSEQ_SUCCESS);
1661                    }
1662                }
1663                last_ret = ret;
1664            }
```

The next chart shows the difference in success rates for this new strategy. The initial allocation is higher for all but one of the tests. This is a direct result, as shown by gathered statistics (reference gathered stats here?), of both the fault tolerance and retry changes we introduced.



5 Postamble

6 Future Works

Appendix 1: Table

A Data Table

Table 1: Hugepage Allocation Data Points

Appendix 2: List of Figures

List of Figures