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[1] which eventually calls into __alloc_pages_nodemask[2], 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[3] 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[4], 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

[1] hugetlb.c
[2] page_alloc.c
[3] hugetlb.c
[4] hugetlb.c

...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...

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

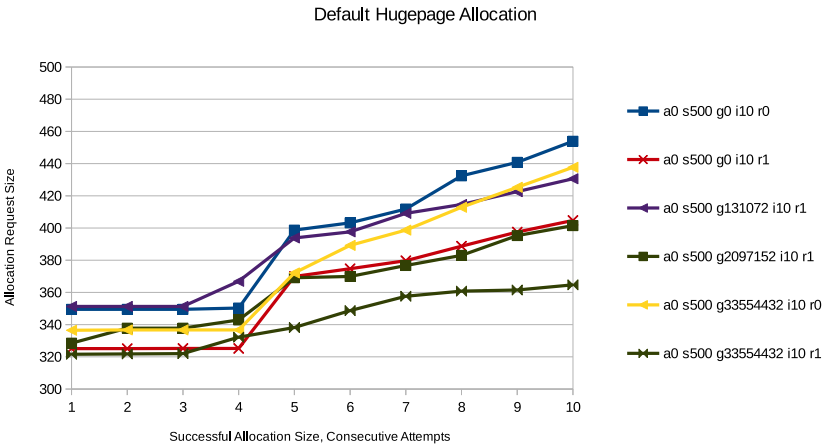
To demonstrate the issues with the original algorithm, as above, we provide two figures which describe 6 different scenarios for the default and new allocation method (hereafter called aggressive). The scenarios are a set of configuration options for the test run in question, where a test run is a certan amount of setup and then an allocation request of the kernel. All scenarios were run directly after a complete reboot, against a 64-bit 3.10 linux kernel[5]. The figures are of the average of 10 runs[6]. These scenarios are encoded as the first letter of each option, followed by a number, and are as follows:

- (a)ggressive: Whether to use aggressive alloc.
- (s)et: The number of pages to allocate. In all of our tests this number was 500, due to the fact that it was near the maximum suppliable by the system and thus demonstrated the problems with large requests best.
- (g)rab: Whether to cause the test application to allocate memory (to disrupt hugepage allocation), and how much if so. 0 means none was allocated. This allocation is done in a series of blocks, rather than in one large request, in order to spread the memory allocations to a variety of nodes and to better simulate the real memory allocation patterns of an application.
- (i)terations: 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.
- (r)eset: Whether to re-set the number of hugepages to 0 after each iteration. 0 means do not reset. We ran 4 scenarios which reset and 2 which did

[5]Linux 3.10.0-327.4.5.el7.x86_64.debug #22 SMP x86_64 GNU/Linux
[6]All run data availble in appendix D

not reset. In many real world scenarios an application making a hugepage request that fails will simply want to back off the allocation and attempt it again later, as it needs the complete request to be fulfilled. However it is also possible that the application will be able to incrementally make use of the hugepages or will be designed to re-request until the allocation is fulfilled, which is likely a necessary choice when requesting large allocations even with these changes. Not resetting the hugepages significantly eases the difficulty of fulfilling the allocation request for the kernel, while resetting demonstrates the efficacy of an intermittent allocation workload.

We use these parameters to define our tests in order to simulate a variety of real world use cases.



As this chart shows, the default allocation strategy generally fails, on average, 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 between 320 and 350 pages of the 500 requested. The next key point is the inflection that occurs at attempt 3 and 4, when all except the most stringent scenario achieved a slightly higher allocation. However, none pass the 400 page mark, and over the course of the next four attempts most barely improve. Much of this improvement can be attributed to anonymous huge pages slowly aggregating all of the memory we are requesting and then re-supplying it to us every time we ask for an allocation. The one scenarion that has not made dramatic improvements is the the g33445532 r1, which is the most demanding scenario. The third key point is the final allocation attempt. At this juncture three scenarios are at about 450 pages (two of which are r0), having improved by roughly 10 pages or so each attempt, though none have completely fulfilled the

request. g33554432 r0 has slowly accumulated more pages but is only marginally greater than it was 6 attempts ago, and the two remaining have finally achieved about 400, roughly the same as the 3 best 6 attempts ago. One thing to note is that past success is a strong predictor of future success, as the trend lines demonstrates. In summary, none of these scenarios averaged complete success, and half were at only 400 out of 500 pages after 10 consecutive allocation attemps. Total improvement for the scenarios was capped at roughly 150 pages. Now that we have examined the default scenarios, we will discuss the proposed changes and show the results of the identical tests run against them.

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[7] (which frees up pages on that node which can be swapped out). If this succeeds, we will once again attempt to allocate on this node. This is important because often allocation on a node is blocked by pages that are still in memory but do not need to be, i.e., they can be swapped out immediately by the kernel. This will effect the performance of other applications on the system but it still follows the default kernel policy (in our case LRU) and therefore disruption should be relatively minimal.

[7] http://lxr.free-electrons.com/source/mm/vmscan.c#L2826

```
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 }
```

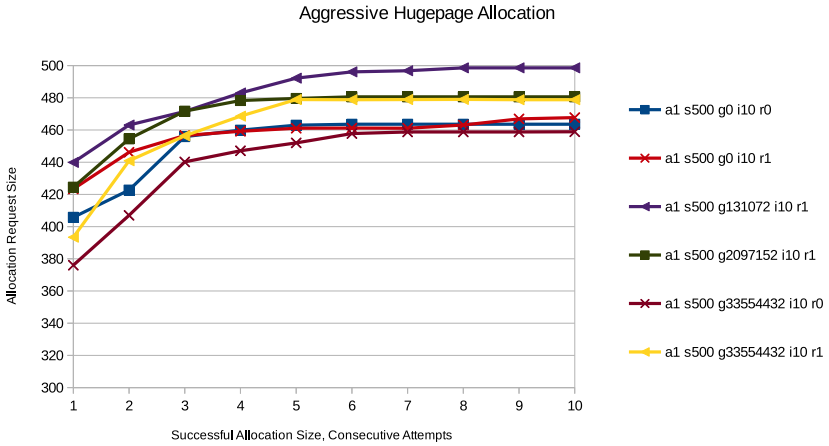
Second, if we succeed in freeing pages then we re-attempt the allocation on that node. Additionally, we made a change to the calling algorithm, set_max_huge_pages, where we will retry until either we succeed in fulfilling the allocation request or we fail at allocating any huge pages twice in a row. Technically we could

simply attempt to allocate until there are no available pages left at all, and this would guarantee completion of the request if it is possible, however we decided to make a compromise between efficacy and hogging kernel runtime. An even more relaxed failure tolerance would likely yield even better results but increase the kernel contention significantly.

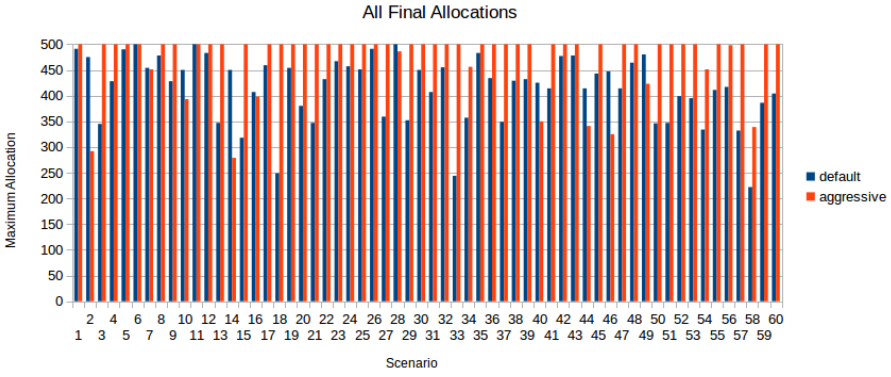
```
1643                    spin_unlock(&hugetlb_lock);
1644                    if (hstate_is_gigantic(h)) {
1645                            ret = alloc_fresh_gigantic_page(h, nodes_allowed);
1646            } else {
1647                            ret = alloc_fresh_huge_page(h, nodes_allowed);
1648            }
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. On average, the initial allocation is more than 100 pages higher for all scenarios. Rather than taking 4 consecutive attempts to reliably achieve more than the initial allocation, by the 3rd attempt all scenarios are within 10 pages of their maximum. As can be seen, though only one scenario achieves the total of 500, all scenarios achieve significantly higher allocations significantly faster, with all scenarios above 460 pages at final attempt, which was not achieved by even the least stringent default scenario. Additionally, unlike the default strategy, the scenario configuration had significantly less impact on the success of the allocation, because the aggressive alloc is able to work around large memory allocation and page resets significantly better. Gathered statistics[8] demonstrate the importance of both of the fault tolerance and retry changes we introduced.

[8] see appendix C



Next, we present a chart of all scenarios' last result to compare ultimate efficacy. Note that scenarios can achieve a maximum result much sooner than the final attempt and in all such cases continued to allocate the maximum number until the end of the scenario. Each column of the chart is the exact same scenario's final result for default and aggressive allocations, for all scenarios run (i.e., 10 runs of each scenario configuration). 46 of 50 of the aggressive scenarios actually completed the 500 page request, but the averages in the charts are lowered by test runs where multiple consecutive nodes are both full and lack enough freeable pages. Conversely, only 3 of 50 the default scenarios achieved 500 pages allocated. Of note is that almost all scenarios for both aggressive and default had at least one test run where the final total failed to achieve a significant improvement over its the initial attempted allocation. We will discuss a potential solution to this issue in Future Works.



5 Future Works

The one major algorithmic change that could yield even better results than obtained is modifying the set_max_huge_pages algorithm further. Two different strategies could be employed. First, we could avoid the linear search and instead introduce a progressive or randomized starting node choice. This would allow us to eventually overcome a large series of nodes all of which are unusable, however it would place the burden on the requestor to re-ask enough times to fulfill the request. Instead of this, we could make set_max_huge_pages simply skip over unusable nodes until the allocation is completed or it has judged that all suitable nodes are exhausted.

6 Postamble

In summary, we have presented a use case for the hugetlb mechanism of the kernel, made changes and demonstrated their efficacy. To do so, we have collected and analyzed data regarding both the current implementation and the aggressive implementation. The data unequivocally show that the new strategy is superior in its ability to fulfill large hugepage allocation requests. We have examined the kernel changes that yielded the results and made suggestions for what could be pursued next. Finally, all data and code used for this paper is available at the public github repository (https://github.com/rivenwyrm/htlb_project).

Appendix 1: Table

A Data Table

Table 1: Hugepage Allocation Data Points

Appendix 2: List of Figures

List of Figures