HugeTLB Improvement to Reduce Post-Boot Allocation Failures

Joshua Nicolas Verburg-Sachs

2016-02

Contents

1 Abstract

The HugeTLB system in the Linux kernel is a useful tool for ensuring speed and performance of applications with large memory demands. Currently the allocation of hugepages is only reliable at boottime, when the kernel can easily obtain the desired memory. However not all use cases know their needs ahead of time, and sometimes those needs change during runtime. Attempts to allocate during runtime often fail. So we propose several changes to the allocation mechanism to increase the likelihood of success during runtime. These changes consist mostly of modifications to the hugetlb.c file in the Linux kernel. We have developed a test suite to allow us to examine both the default and new implementation. The results are that for large hugepage allocations, we see a success rate of 5% for the default and 76% for the new implementation, out of 60 independent test runs against each, in 6 different scenarios. Therefore, we conclude that the changes significantly improve over the default implementation. We also suggest several improvements which could be made to further increase the success rate.

2 Introduction

This research is inspired by the needs of modern (semi) real-time processes handling significant traffic loads, where their memory requirements can vary dramatically with the specific data to handle or even just time of day. 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 (HugePages Kernel Documentation), 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.

3 Original HugeTLB Allocation Implementation

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[1]:

```
Function AllocFreshHugePage
    ForEachMemoryNode(nodes)
        if AllocFreshHugePageNode(node)
            break
```

As originally written, the algorithm iterates over the given allowed nodes, calling **AllocFreshHugePageNode** which eventually calls into **AllocPagesNodemask**, where the kernel employs a variety of tools to attempt to gather enough memory to fulfill the request from this exact node.

```
Function SetMaxHugePages
    while SurplusHugePages and HugePagesAllocated() < HugePagesDesired()
        AdjustPoolSurplus
    while HugePagesAllocated() < HugePagesDesired()
        if AllocFreshHugePage() fails
            break
```

SetMaxHugePages[2] is called by the kernel to allocate the total requested number of pages. Note that if we fail even a single call of **AllocFreshHugePage**, 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.

4 Test Results For Default Implementation

To demonstrate the issues with the original algorithm, as above, we provide two figures which describe six 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

[1] See Appendix C for code
[2] See Appendix D for code

a complete reboot. The test bed was a generic Centos 7 installation on a 2 CPU 2GB RAM VMware Workstation 12 VM used only for this purpose, with a 64-bit 3.10 linux kernel[3]. The figures are of the average of 10 runs[4]. 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[5] 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 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.

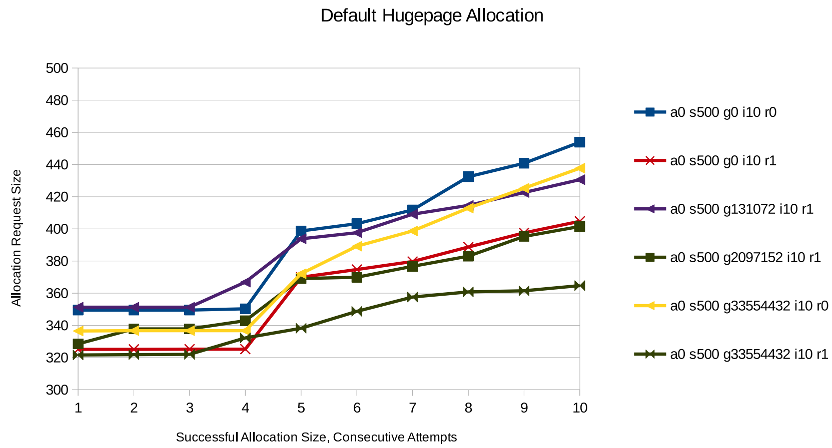
We use these parameters to define our tests in order to simulate a variety of real world use cases.

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

[3]Linux 3.10.0-327.4.5.el7.x86_64.debug #22 SMP x86_64 GNU/Linux

[4]Please see results_test_app/aggr and results_test_app/nonaggr in the GitHub repo for total test run printouts

[5]See Appendix B for system memory information.



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 scenario that has not made dramatic improvements is the the g33445532 r1, which is the most demanding of the kernel. 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 six attempts ago, and the two remaining have finally achieved about 400, roughly the same as the 3 most successful scenarios 6 attempts ago. One thing to note is that past success is a strong predictor of future success, as the graph 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.

5 'Aggressive' HugeTLB Allocation Implementation

There are two important changes to the **AllocFreshHugePage**[6] algorithm. First, if we completely fail to allocate any pages, we call **TryToFreePages** (freeing up pages on that memory node which can be swapped out). This is

[6]See Appendix E for code

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.

```
Function AllocFreshHugePage()
    ForEachMemoryNode(nodes)
        if AllocFreshHugePageNode(node)
            return
    if not aggressive
return
    ForEachMemoryNode(nodes)
        if TryToFreePages(node)
            if AllocFreshHugePageNode(node)
                break
```

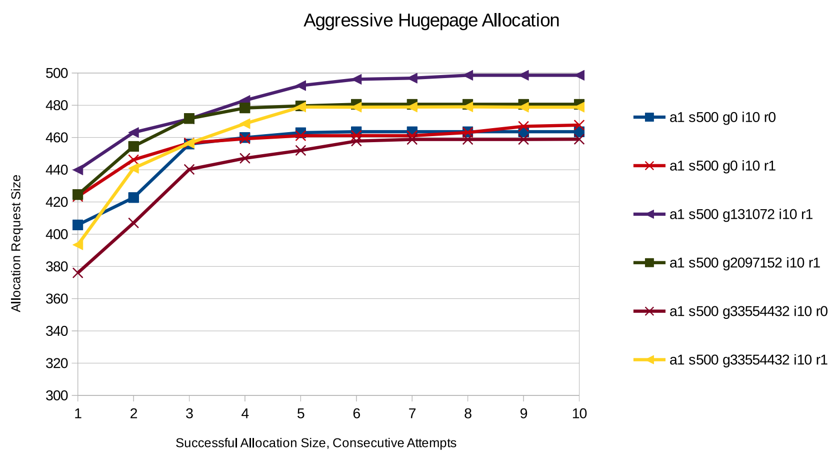
Second, if we succeed in freeing pages then we re-attempt the allocation on that node. Additionally, we made a change to the calling algorithm, **SetMaxHugePages**[7], where we will retry until either we succeed in fulfilling the allocation request or we fail at allocating any huge pages twice in a row. 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.

```
Function SetMaxHugePages
    while SurplusHugePages and HugePagesAllocated() < HugePagesDesired()
        AdjustPoolSurplus
    while HugePagesAllocated() < HugePagesDesired()
        if AllocFreshHugePage() fails
            if Failed
                break
            Failed = True
        else
            Failed = False
```

[7]See Appendix F for code

6 Test Results And Comparison for 'Aggressive' Implementation

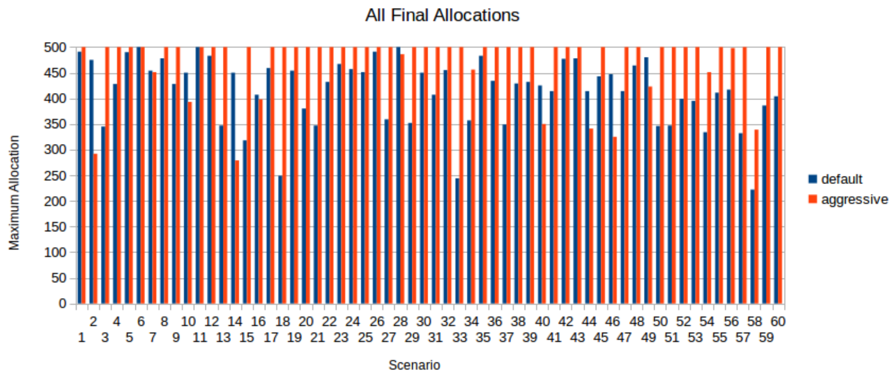
The next chart shows the difference in success rates for this new strategy. On average, the initial allocation is more than 100 pages higher for all scenarios. Rather than taking 4 consecutive attempts to reliably achieve more than the initial allocation, by the 3rd attempt all scenarios are within 10 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.



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 60 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 out of 60 of the default scenarios achieved 500 pages allocated. Of note is that almost all scenarios for both

[8]Please see Appendix A

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.



7 Future Works

The one major algorithmic change that could yield even better results than obtained is modifying **set_max_huge_pages** 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. The drawback to this strategy is that iterating over all nodes is a very lengthy process, and similarly any method to determine an appropriate level of 'exhaustion' could prove laborious if it is not simply arbitrary. However either of these would likely yield even better results than the simple changes we propose and are worth investigating.

8 Conclusion

In summary, we have presented a use case for the hugetlb mechanism of the kernel where the current implementation is not able to meet the required needs, made changes to significantly increase the likelihood of success and subsequently demonstrated their efficacy. To do so, we have collected and analyzed data regarding both the current implementation and the aggressive implementation. We show that the new strategy is unequivocally 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`).

Appendices

A Raw Data

```
./results_test_app/nonaggr/
a0 s500 g0 i10 r0,337,337,337,337,388,388,388,413,450,491,
a0 s500 g0 i10 r0,332,332,332,332,397,410,422,460,462,475,
a0 s500 g0 i10 r0,343,343,343,343,344,345,345,345,345,345,
a0 s500 g0 i10 r0,340,340,340,340,398,398,398,428,428,428,
a0 s500 g0 i10 r0,335,335,335,335,395,395,426,474,479,490,
a0 s500 g0 i10 r0,465,465,465,473,491,498,498,500,500,500,
a0 s500 g0 i10 r0,334,334,334,334,399,413,429,449,454,454,
a0 s500 g0 i10 r0,349,349,349,349,404,414,414,444,444,478,
a0 s500 g0 i10 r0,329,329,329,329,382,382,393,399,399,428,
a0 s500 g0 i10 r0,331,331,331,331,389,389,405,412,447,450,
a0 s500 g0 i10 r1,356,356,356,356,500,500,500,500,500,500,
a0 s500 g0 i10 r1,333,333,333,333,389,391,391,425,448,483,
a0 s500 g0 i10 r1,345,345,345,345,345,347,347,347,347,347,
a0 s500 g0 i10 r1,325,325,325,325,380,380,399,421,450,450,
a0 s500 g0 i10 r1,318,318,318,318,318,318,318,318,318,318,
a0 s500 g0 i10 r1,327,327,327,327,376,376,395,403,403,407,
a0 s500 g0 i10 r1,357,357,357,357,409,409,421,428,435,459,
a0 s500 g0 i10 r1,248,248,249,249,249,249,249,249,249,249,
a0 s500 g0 i10 r1,327,327,327,327,381,425,425,444,445,454,
a0 s500 g0 i10 r1,315,315,315,315,352,352,352,352,380,380,
a0 s500 g131072 i10 r1,347,347,347,347,347,347,347,347,347,347,
a0 s500 g131072 i10 r1,353,353,353,353,404,404,423,423,432,432,
a0 s500 g131072 i10 r1,342,342,342,342,393,406,426,440,462,467,
a0 s500 g131072 i10 r1,358,358,358,358,418,418,450,454,457,457,
a0 s500 g131072 i10 r1,350,350,350,350,403,403,412,423,433,451,
a0 s500 g131072 i10 r1,350,350,350,403,409,413,434,452,465,491,
a0 s500 g131072 i10 r1,358,358,358,358,359,359,359,359,359,359,
a0 s500 g131072 i10 r1,355,355,355,406,451,471,475,480,481,500,
a0 s500 g131072 i10 r1,350,350,350,350,351,352,352,352,352,352,
a0 s500 g131072 i10 r1,350,350,350,402,403,403,413,416,438,450,
a0 s500 g2097152 i10 r1,328,328,328,328,329,329,330,330,394,407,
a0 s500 g2097152 i10 r1,353,353,353,353,400,400,416,441,444,455,
a0 s500 g2097152 i10 r1,244,244,244,244,244,244,244,244,244,244,
a0 s500 g2097152 i10 r1,262,355,355,355,355,357,357,357,357,357,
a0 s500 g2097152 i10 r1,350,350,350,401,422,422,422,447,480,483,
a0 s500 g2097152 i10 r1,360,360,360,360,405,405,405,405,405,434,
a0 s500 g2097152 i10 r1,344,344,344,344,344,350,349,349,349,349,
a0 s500 g2097152 i10 r1,342,342,342,342,391,391,415,422,423,429,
a0 s500 g2097152 i10 r1,354,354,354,354,405,405,418,424,432,432,
a0 s500 g2097152 i10 r1,348,348,348,348,396,396,411,411,425,425,
```

```
a0 s500 g33554432 i10 r0,334,334,334,334,385,392,399,404,412,414,
a0 s500 g33554432 i10 r0,326,326,326,326,378,387,398,427,465,477,
a0 s500 g33554432 i10 r0,340,341,341,341,397,407,407,439,465,478,
a0 s500 g33554432 i10 r0,334,334,334,334,381,403,403,403,406,414,
a0 s500 g33554432 i10 r0,331,331,331,331,377,390,390,404,427,443,
a0 s500 g33554432 i10 r0,341,341,341,341,343,386,406,413,436,447,
a0 s500 g33554432 i10 r0,343,343,343,343,343,392,401,404,407,414,
a0 s500 g33554432 i10 r0,333,333,333,333,379,391,427,452,452,464,
a0 s500 g33554432 i10 r0,340,340,340,340,392,398,410,437,437,480,
a0 s500 g33554432 i10 r0,343,344,344,344,346,346,346,346,346,346,
a0 s500 g33554432 i10 r1,344,344,344,344,345,346,347,347,347,347,
a0 s500 g33554432 i10 r1,333,333,333,333,377,377,389,398,398,399,
a0 s500 g33554432 i10 r1,334,334,334,334,335,372,377,382,382,395,
a0 s500 g33554432 i10 r1,333,333,334,334,334,334,334,334,334,334,
a0 s500 g33554432 i10 r1,340,340,340,389,393,393,402,411,411,411,
a0 s500 g33554432 i10 r1,329,329,329,375,384,384,395,395,402,417,
a0 s500 g33554432 i10 r1,325,325,325,332,332,332,332,332,332,332,
a0 s500 g33554432 i10 r1,219,221,222,222,222,222,222,222,222,222,
a0 s500 g33554432 i10 r1,316,316,316,316,317,372,374,383,383,386,
a0 s500 g33554432 i10 r1,343,343,343,343,343,355,404,404,404,404,
./results_test_app/aggr/
a1 s500 g0 i10 r0,428,431,488,500,500,500,500,500,500,500,
a1 s500 g0 i10 r0,291,292,292,292,292,292,292,292,292,292,
a1 s500 g0 i10 r0,442,493,500,500,500,500,500,500,500,500,
a1 s500 g0 i10 r0,410,447,468,469,500,500,500,500,500,500,
a1 s500 g0 i10 r0,487,500,500,500,500,500,500,500,500,500,
a1 s500 g0 i10 r0,344,344,500,500,500,500,500,500,500,500,
a1 s500 g0 i10 r0,382,409,418,445,445,451,451,451,451,451,
a1 s500 g0 i10 r0,479,500,500,500,500,500,500,500,500,500,
a1 s500 g0 i10 r0,402,418,500,500,500,500,500,500,500,500,
a1 s500 g0 i10 r0,393,393,393,393,393,393,393,393,393,393,
a1 s500 g0 i10 r1,500,500,500,500,500,500,500,500,500,500,
a1 s500 g0 i10 r1,476,499,500,500,500,500,500,500,500,500,
a1 s500 g0 i10 r1,437,478,492,500,500,500,500,500,500,500,
a1 s500 g0 i10 r1,279,279,279,279,279,279,279,279,279,279,
a1 s500 g0 i10 r1,413,441,477,480,500,500,500,500,500,500,
a1 s500 g0 i10 r1,395,398,398,398,398,399,398,399,398,398,
a1 s500 g0 i10 r1,435,491,500,500,500,500,500,500,500,500,
a1 s500 g0 i10 r1,462,485,500,500,500,500,500,500,500,500,
a1 s500 g0 i10 r1,425,434,434,435,434,434,434,453,492,500,
a1 s500 g0 i10 r1,412,457,486,500,500,500,500,500,500,500,
a1 s500 g131072 i10 r1,450,500,500,500,500,500,500,500,500,500,
a1 s500 g131072 i10 r1,500,500,500,500,500,500,500,500,500,500,
a1 s500 g131072 i10 r1,411,467,473,480,484,484,484,500,500,500,
a1 s500 g131072 i10 r1,495,498,500,500,500,500,500,500,500,500,
a1 s500 g131072 i10 r1,431,438,473,500,500,500,500,500,500,500,
```

```
a1 s500 g131072 i10 r1,426,447,447,499,500,500,500,500,500,500,
a1 s500 g131072 i10 r1,458,488,500,500,500,500,500,500,500,500,
a1 s500 g131072 i10 r1,357,357,357,357,441,477,484,486,486,486,
a1 s500 g131072 i10 r1,445,490,500,500,500,500,500,500,500,500,
a1 s500 g131072 i10 r1,426,446,465,494,497,500,500,500,500,500,
a1 s500 g2097152 i10 r1,500,500,500,500,500,500,500,500,500,500,
a1 s500 g2097152 i10 r1,400,415,442,483,490,500,500,500,500,500,
a1 s500 g2097152 i10 r1,415,451,495,500,500,500,500,500,500,500,
a1 s500 g2097152 i10 r1,397,440,455,456,456,456,456,456,456,456,
a1 s500 g2097152 i10 r1,399,459,476,494,500,500,500,500,500,500,
a1 s500 g2097152 i10 r1,451,499,500,500,500,500,500,500,500,500,
a1 s500 g2097152 i10 r1,438,454,499,500,500,500,500,500,500,500,
a1 s500 g2097152 i10 r1,456,488,500,500,500,500,500,500,500,500,
a1 s500 g2097152 i10 r1,439,489,500,500,500,500,500,500,500,500,
a1 s500 g2097152 i10 r1,350,350,350,350,350,350,350,350,350,350,
a1 s500 g33554432 i10 r0,411,454,483,483,488,490,500,500,500,500,
a1 s500 g33554432 i10 r0,392,431,437,443,459,500,500,500,500,500,
a1 s500 g33554432 i10 r0,430,492,500,500,500,500,500,500,500,500,
a1 s500 g33554432 i10 r0,341,341,341,341,341,341,341,341,341,341,
a1 s500 g33554432 i10 r0,433,445,499,500,500,500,500,500,500,500,
a1 s500 g33554432 i10 r0,322,324,325,325,325,325,325,325,325,325,
a1 s500 g33554432 i10 r0,414,441,471,479,495,500,500,500,500,500,
a1 s500 g33554432 i10 r0,371,464,493,500,500,500,500,500,500,500,
a1 s500 g33554432 i10 r0,386,410,420,420,422,422,422,422,422,423,
a1 s500 g33554432 i10 r0,260,268,433,480,490,500,500,500,500,500,
a1 s500 g33554432 i10 r1,338,434,477,480,500,500,500,500,500,500,
a1 s500 g33554432 i10 r1,326,439,464,486,500,500,500,500,500,500,
a1 s500 g33554432 i10 r1,407,437,464,463,500,500,500,500,500,500,
a1 s500 g33554432 i10 r1,416,439,452,451,451,451,452,452,451,451,
a1 s500 g33554432 i10 r1,458,500,500,500,500,500,500,500,500,500,
a1 s500 g33554432 i10 r1,415,490,492,493,498,498,498,499,498,498,
a1 s500 g33554432 i10 r1,428,466,485,500,500,500,500,500,500,500,
a1 s500 g33554432 i10 r1,337,338,338,339,340,339,339,339,339,339,
a1 s500 g33554432 i10 r1,403,434,436,474,500,500,500,500,500,500,
a1 s500 g33554432 i10 r1,406,432,456,500,500,500,500,500,500,500,
```

B Memory Information

```
MemTotal:        1621960 kB
MemFree:          514528 kB
MemAvailable:     871008 kB
Buffers:             752 kB
Cached:           452596 kB
SwapCached:            0 kB
Active:           485364 kB
Inactive:         376344 kB
```

```
Active(anon):         409100  kB
Inactive(anon):         9068  kB
Active(file):          76264  kB
Inactive(file):       367276  kB
Unevictable:               0  kB
Mlocked:                   0  kB
SwapTotal:           2097148  kB
SwapFree:            2097148  kB
Dirty:                     8  kB
Writeback:                 0  kB
AnonPages:            408384  kB
Mapped:                97640  kB
Shmem:                  9808  kB
Slab:                 133312  kB
SReclaimable:          51128  kB
SUnreclaim:            82184  kB
KernelStack:           11168  kB
PageTables:            20824  kB
NFS_Unstable:              0  kB
Bounce:                    0  kB
WritebackTmp:              0  kB
CommitLimit:         2908128  kB
Committed_AS:        2073504  kB
VmallocTotal:    34359738367  kB
VmallocUsed:          423048  kB
VmallocChunk:    34359303168  kB
HardwareCorrupted:         0  kB
AnonHugePages:        118784  kB
HugePages_Total:           0
HugePages_Free:            0
HugePages_Rsvd:            0
HugePages_Surp:            0
Hugepagesize:           2048  kB
DirectMap4k:          100224  kB
DirectMap2M:         1996800  kB
DirectMap1G:               0  kB
```

C Original alloc_fresh_huge_page C Code

```c
static int alloc_fresh_huge_page(struct hstate *h, nodemask_t *nodes_allowed)
{
	struct page *page;
	int nr_nodes, node;
	int ret = 0;

	for_each_node_mask_to_alloc(h, nr_nodes, node, nodes_allowed) {
		page = alloc_fresh_huge_page_node(h, node);
		if (page) {
			ret = 1;
			break;
		}
	}

	if (ret)
		count_vm_event(HTLB_BUDDY_PGALLOC);
	else
		count_vm_event(HTLB_BUDDY_PGALLOC_FAIL);

	return ret;
}
```

D Original set_max_huge_pages C Code

...set_max_huge_pages...

```c
		spin_unlock(&hugetlb_lock);
		if (hstate_is_gigantic(h))
			ret = alloc_fresh_gigantic_page(h, nodes_allowed);
		else
			ret = alloc_fresh_huge_page(h, nodes_allowed);
		spin_lock(&hugetlb_lock);
		if (!ret)
			goto out;
```

...set_max_huge_pages...

E Aggressive alloc_fresh_huge_page C Code

```c
static unsigned int alloc_fresh_huge_page(struct hstate *h, nodemask_t *nodes_al
{
    struct page *page = NULL;
    int nr_nodes, node, ret = 0, num_nodes = 0;

    for_each_node_mask_to_alloc(h, nr_nodes, node, nodes_allowed) {
        num_nodes++;
        page = alloc_fresh_huge_page_node(h, node);
        if (page) {
            goto out;
        }
    }

    if (!hugepages_aggressive_alloc) {
        goto out;
    }
    for_each_node_mask_to_alloc(h, nr_nodes, node, nodes_allowed) {
        gfp_t gfp_mask = htlb_alloc_mask|__GFP_COMP|__GFP_THISNODE|
                __GFP_REPEAT|__GFP_NOWARN;
        struct zonelist *zonelist = node_zonelist(node, gfp_mask);

        if (try_to_free_pages(zonelist, h->order, gfp_mask, nodes_allowed) > 0)
            page = alloc_fresh_huge_page_node(h, node);
            if (page) {
                count_vm_event(HTLB_BUDDY_PGALLOC_RETRY_SUCCESS);
                break;
            }
            count_vm_event(HTLB_BUDDY_PGALLOC_RETRY_FAIL);
        }
    }
 out:
    if (page) {
        count_vm_event(HTLB_BUDDY_PGALLOC);
        ret = 1;
    } else {
        count_vm_event(HTLB_BUDDY_PGALLOC_FAIL);
        ret = 0;
    }
    return ret;
}
```

F Aggressive set_max_huge_pages C Code

```c
        spin_unlock(&hugetlb_lock);
        if (hstate_is_gigantic(h)) {
                        ret = alloc_fresh_gigantic_page(h, nodes_allowed);
        } else {
                        ret = alloc_fresh_huge_page(h, nodes_allowed);
        }
        spin_lock(&hugetlb_lock);
        if (hstate_is_gigantic(h) || !hugepages_aggressive_alloc) {
            if (!ret)
                goto out;
        } else {
            /* If two successive allocs failed, bail */
            if (last_ret == 0) {
                if (!ret) {
                    count_vm_event(HTLB_BUDDY_PGALLOC_SUBSEQ_FAIL);
                    goto out;
                } else {
                    count_vm_event(HTLB_BUDDY_PGALLOC_SUBSEQ_SUCCESS);
                }
            }
            last_ret = ret;
        }
```