The What, The Why and the Where To of Anti-Fragmentation

Mel Gorman
IBM Corp. and Uni. of Limerick
mel@csn.ul.ie

Andy Whitcroft
LTC, IBM Corp.
andyw@uk.ibm.com

Abstract

Linux® uses a variant of the binary buddy allocator that is fast but suffers badly from external fragmentation and is unreliable for large contiguous allocations. We begin by introducing two cases where large contiguous regions are needed: the allocation of HugeTLB pages during the lifetime of the system and using memory hotplug to on-line and off-line memory on demand in support of changing loads. We also mention subsystems that may benefit from using contiguous groups of pages. We then describe two anti-fragmentation strategies, discuss their strengths and weaknesses and examine their implementations within the kernel. We cover the standardised tests, the metrics used, the system architectures tested in the evaluation of these strategies and conclude with an examination of their effectiveness at satisfying large allocations. We also look at a page reclamation strategy that is suited to freeing contiguous regions of pages and finish with a look at the future direction of anti-fragmentation and related work.

1 Introduction

The page allocator in any operating system is a critical component. It must be fast and have the ability to satisfy all requests to avoid subsystems building reserve page pools [4]. Linux uses a variant of the binary buddy allocator that is known to be fast in comparison to other allocator types [3] but behaves poorly in the face of fragmentation [5].

Fragmentation is a space-efficiency problem affecting all dynamic memory allocators and comes in two varieties; internal and external. Internal fragmentation occurs when a larger free block than necessary is granted for a request, such as allocating one entire page to satisfy a request for 32 bytes. Linux uses a slab allocator for small requests to address this issue. External fragmentation refers to the inability to satisfy an allocation because a suitably large block of memory is not free even though enough memory may be free overall [6]. Linux deals with external fragmentation by rarely requiring larger (*high order*) pages. Although this works well in general, Section 2 presents situations where it performs poorly.

To be clear, anti-fragmentation is not the same as defragmentation, which is a mechanism to reduce fragmentation by moving or reclaiming pages to have contiguous free space. Anti-fragmentation enables a system to conduct a partial defragmentation using the existing page reclamation mechanism. The remainder of this paper is arranged as described in the abstract.

2 Motivation for Low Fragmentation

HugeTLB pages are contiguous regions that match a large page size provided by an architecture, which is 1024 small pages on x86 and 4096 on PPC64. Use of these large pages reduces both expensive TLB misses [2] and the number of *Page Table Entries (PTEs)* required to map an area, thus increasing performance and reducing memory consumption. Linux keeps a HugeTLB freelist in the HugeTLB page pool. This pool is sized at boot time, which is a problem for workloads requiring different amounts of HugeTLB memory at different times. For example, workloads that use large in-memory data sets, such as X Windows, High-Performance Computing (HPC), many Java applications, and some desktop applications (e.g. Konqueror) require variable amounts of memory depending on the input data and type of usage. It is not possible to guess their needs at boot time. Instead it would be better to maintain low fragmentation so that their needs could be met as needed at run-time.

Contiguous regions are also required when a section of memory needs to be on-lined and then off-lined later. For example, a virtual machine running a service like a web server may require more memory due to a spike in usage, but later need to return the memory to the host. Some architectures can return memory to a *hypervisor* using a *balloon driver* but this only works when memory can be off-lined at the page granularity. The minimum sized region that can be off-lined is the same as the size of a *memory section* defined for the SPARSEMEM memory model. This model mandates that the memory section size be a power-of-two number of pages and the architecture selects a size within that constraint. On the PPC64, the minimum sized region of memory that can be off-lined is 16MiB which is the minimum size OpenFirmware uses for a *Logical Memory Block (LMB)*. On x86, the minimum sized region is 64MiB. This is the smallest DIMM size taken by the IBM xSeries® 445 which supports the memory hot-add feature. Low fragmentation increases the probability of finding regions large enough to off-line.

A third case where contiguous regions are desired, but not required, is for drivers that use DMA but do not support scatter/gather IO efficiently or do not have an IO-MMU available. These drivers must spend time breaking up the DMA request into page-sized units. Ideally, drivers could ask for a page-aligned block of memory and receive a list of large contiguous regions. With low fragmentation, the expectation is that the driver would have a better chance of getting one contiguous block and not need to break up the request.

3 External Fragmentation

The extent of fragmentation depends on the number of free blocks[1] in the system, their size and the size of the requested allocation. In this section, we define two metrics that are used to measure the ability of a system to satisfy an allocation and the degree of fragmentation.

We measure the fraction of available free memory that can be used to satisfy allocations of a specific size using an *unusable free space index*, F_u.

$$F_u(j) = \frac{TotalFree - \sum_{i=j}^{i=n} 2^i k_i}{TotalFree}$$

[1] A free block is a single contiguous region stored on a freelist. In rare cases with the buddy allocator, two free blocks are adjacent but not merged because they are not buddies.

where $TotalFree$ is the number of free pages, 2^n is the largest allocation that can be satisfied, j is the order of the desired allocation and k_i is the number of free page blocks of size 2^i. When $TotalFree$ is 0, we define F_u to be 1. A more traditional, if slightly inaccurate[2], view of fragmentation is available by multiplying $F_u(j)$ by 100. At 0, there is 0% fragmentation, at 1, there is 100% fragmentation, at 0.25, fragmentation is at 25% and 75% of available free memory can be used to satisfy a request for 2^j contiguous pages.

$F_u(j)$ can be calculated at any time, but external fragmentation is not important until an allocation fails [5] when $F_u(j)$ will be 1. We further define a *fragmentation index*, $F_i(j)$, which determines if the failure to allocate a contiguous block of 2^j pages is due to lack of memory or to external fragmentation. The higher the fragmentation of the system, the more free blocks there will be. At the time of failure, the ideal number of blocks shall be related to the size of the requested allocation. Hence, the index at the time of an allocation failure is

$$F_i(j) = 1 - \frac{TotalFree/2^j}{BlocksFree}$$

where $TotalFree$ is the number of free pages, j is the order of the desired allocation and $BlocksFree$ is the number of contiguous regions stored on freelists. When $BlocksFree$ is 0, we define $F_i(j)$ to be 0. A negative value of $F_i(j)$ implies that the allocation can be satisfied and the fragmentation index is only meaningful when an allocation fails. A value tending towards 0 implies the allocation failed due to a lack of memory. A value tending towards 1 implies that the failure is due to fragmentation. Obviously the fewer times the F_i are calculated, the better.

[2]Discussions on fragmentation are typically concerned with internal fragmentation where the percentage represents wasted memory. A percentage value for external fragmentation is not as meaningful because it depends on the request size.

4 Allocator Placement Policies

It is common for allocators to exploit known characteristics of the request stream to improve their efficiency. For example, allocation size and the relative time of the allocation have been used to heuristically group objects of an expected lifetime together [1]. Similar heuristics cannot be used within an operating system as it does not have the same distinctive phases as application programs have. There is also little correlation between the size of an allocation and its expected use. However, operating system allocations do have unique characteristics that may be exploited to control placement thereby reducing fragmentation.

First, certain pages can be freed on demand; saved to backing storage; or discarded. Second, a large amount of kernel allocations are for caches, such as the buffer and inode caches which may be reclaimed on demand. Since it is known in advance what the page will be used for, an anti-fragmentation strategy can group pages by *allocation type*. We define three types of reclaimability

Easy to reclaim (EasyRclm) pages are allocated directly for a user process. Almost all pages mapped to a userspace page table and disk buffers, but not their management structures, are in this category.

Kernel reclaimable (KernRclm) pages are allocated for the kernel but can often be reclaimed on demand. Examples include inodes, buffer head and directory entry caches. Other examples, not applicable to Linux, include kernel data and PTEs where the system is capable of paging them to swap.

Kernel non-reclaimable (KernNoRclm) pages are essentially impossible to reclaim on demand.

To distinguish among the reclamation types, additional GFP flags are used when calling `alloc_pages()`. For simplicity, the strategies presented here treat KernNoRclm and KernRclm the same so we use only one flag `GFP_EASYRCLM` to distinguish between user and kernel allocations. Variations exist that deal with all three reclamation types, but the resulting code is relatively more complex.

Allocation requests that specify the `GFP_EASYRCLM` flag include requests for buffer pages, process faulted pages, high pages allocated with `alloc_zeroed_user_highpage` and shared memory pages. The strategies principally differ in the semantics of the GFP flag and its treatment in the implementation.

5 Anti-Fragmentation With Lists

The binary buddy allocator maintains *max_order* lists of free blocks of each power-of-two from 2^0 to $2^{max\_order-1}$. Instead of one list at each order, this strategy uses two lists by extending `struct free_area`. At each order, one list is used to satisfy EasyRclm allocations and the second list is used for all other allocations. `struct per_cpu_pages` is similarly extended to have one list for EasyRclm and one for kernel allocations.

The difference in design between the standard and list-based anti-fragmentation allocator is illustrated in Figure 1. Where possible, allocations of a specified type use their own freelist but can steal pages from each other in low memory conditions. When allocated, `SetPageEasyRclm()` is called for EasyRclm allocations so that they will be freed back to the correct lists. The two lists mean that a page's buddy is likely to be of the same reclaimability. The success of this strategy depends on there being a large enough number of EasyRclm pages and that there are no prolonged bursts of requests for kernel pages leading to excessive stealing.

One advantage of this strategy is that a high order kernel allocation can push out EasyRclm pages to satisfy the allocation. The assumption is that high-order allocations during the lifetime of the system are short-lived. Performance regressions tests did not show any problems despite the allocator hot paths being affected by this strategy.

A disadvantage is related to the advantage. As kernel order-0 allocations can use the EasyRclm freelists, the strategy can break down if there are prolonged periods of small allocations without frees. The likelihood is also that long-term light loads, such as desktops running for a number of days will allow kernel pages to slowly leak to all areas of physical memory. Over time, the list-based strategy would have similar success rates to the standard allocator.

6 Anti-Fragmentation With Zones

The Linux kernel splits available memory into one or more *zones*, each representing memory with different usage limitations as shown in Figure 2. On a typical x86, we have `ZONE_DMA` representing memory capable of use for Direct Memory Access (DMA), `ZONE_NORMAL` representing memory which is directly accessible by the kernel, and `ZONE_HIGHMEM` covering the remainder. Each zone has its own set of lists for the buddy allocator to track free memory within the zone.

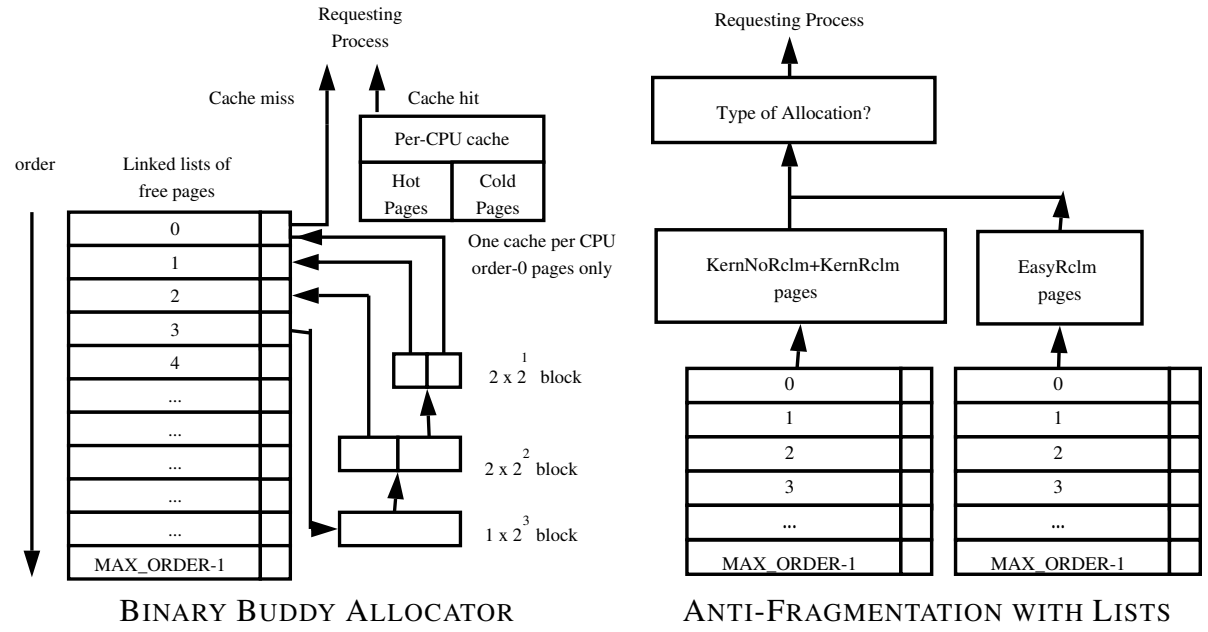


Figure 1: Comparison of the standard and list-based anti-frag allocators

This strategy introduces a new memory zone, ZONE_EASYRCLM, to contain EasyRclm pages as illustrated in Figure 3. EasyRclm allocations that cannot be satisfied from this zone fallback to regular zones, but non-EasyRclm allocations cannot use ZONE_EASYRCLM. This is a crucial difference between the list-based and zone-based strategies for anti-fragmentation as list-based allows stealing in both directions.

While booting, the system memory is split into portions required by the kernel for its operation and that which will be used for EasyRclm allocations. The size of the kernel portion is defined by the system administrator via the kernelcore= kernel parameter, which bounds the memory placed in the standard zones; the remaining memory constitutes ZONE_EASYRCLM. If kernelcore= is not specified, no pages are placed in ZONE_EASYRCLM.

The principal advantage of this strategy are that it provides a high likelihood of being able to reclaim appropriately sized portions of ZONE_EASYRCLM for any higher order allocation if the high-order allocation is also easily reclaimable. Another significant advantage is that ZONE_EASYRCLM may be used for HugeTLB page allocations as they do not worsen the fragmentation state of the system in a meaningful way. This allows us to use the ZONE_EASYRCLM as a "soft allocation" zone from the HugeTLB pool to expand into.

One disadvantage is similar to the HugeTLB pool sizing problem because the usage of the system must be known in advance. Sizing is workload dependant and performance may suffer if an inappropriate size is specified with kernelcore=. The second major disadvantage is that the strategy does not provide any help for high-order kernel allocations.

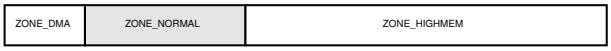


Figure 2: Standard Linux kernel zone layout

CPU	Xeon® 2.8GHz
# Physical CPUs	2
# CPUs	4
Main Memory	1518MiB

X86-BASED TEST MACHINE

CPU	Power5® PPC64 1.9GHz
# Physical CPUs	2
# CPUs	4
Main Memory	4019MiB

POWER5-BASED TEST MACHINE

Figure 4: Specification of Test Machines

Figure 3: Easy Reclaim zone layout

7 Experimental Methodology

The strategies were evaluated using five tests, two related to performance and three related to the system's ability to satisfy large contiguous allocations. The system is cleanly booted at the beginning of a single set of tests. Each of the five tests are run in order without intervening reboots to maximise the chances of the system suffering fragmentation. The tests are as follows

kbuild is similar to kernbench and it measures the time taken to extract and build a kernel. The test gives an overall view of the performance of a kernel, including the rate the kernel is able to satisfy allocations.

AIM9 is a micro-benchmark that includes tests for VM-related operations like page allocation and the time taken to call `brk()`. AIM9 is a good barometer for performance regressions. Crucially, it is sensitive to regressions in the page allocator paths.

HugeTLB-Capability is a kernel compile based benchmark. For every 250MiB of physical memory, a kernel compile is executed (in parallel, simultaneously). During the compile, one attempt is made to grow the HugeTLB page pool from 0 by echoing a large number to `/proc/sys/vm/nr_hugepages`. After the re-size attempt, the pool is shrunk back to 0. The kernel compiles are then stopped and an attempt is made to grow the pool while the system is under no significant load. A zero-filled file that is the same size as physical memory is then created with `dd`, then deleted, before a third attempt is made to re-size the HugeTLB pool. This test determines how capable the system is of allocating HugeTLB pages at run-time using the conventional interfaces.

Highalloc-Stress is a kernel compile based benchmark. Kernel compiles are started as in the HugeTLB-Capability test, plus `updatedb` is also run in the background. A kernel module is loaded to aggressively allocate as many HugeTLB-sized pages as the system has by calling `alloc_pages()`. These persistent attempts force `kswapd` to start reclaiming as well as triggering direct reclaim which does not occur when resizing the HugeTLB pool via `/proc/sys/vm/nr_hugepages`. $F_u(hugetlb\_order)$ is calculated at each allocation attempt and $F_i(hugetlb\_order)$ is calculated at each failure (see Section 3). The results are graphed at the end of the test. This test indicates how many HugeTLB pages could be allocated under the best of circumstances.

HotRemove-Capability is a memory hotplug remove test. For each section of memory reported in `/sys/devices/system/memory`, an attempt is made to off-line the memory. Assuming the kernel supports hotplug-remove, a report states how many sections and what percentage of memory was off-lined. The base kernel used for this paper was `2.6.16-rc6`

which did not support hotplug remove, so no results were produced and it will not be discussed further.

All of these benchmarks were run using driver scripts from VMRegress 0.36[3] in conjunction with the same system that generates the reports on `http://test.kernel.org`. Two machines were used to run the benchmarks based on the x86 and Power5® architectures as detailed in Figure 4. In both cases, the tests were run and results collected with scripts to minimise variation and prevent bias during testing. Four sets of configurations were run on each architecture

1. List-based strategy under light load
2. List-based strategy under heavy load
3. Zone-based with no `kernelcore` specified giving a `ZONE_EASYRCLM` with zero pages.
4. Zone-based with `kernelcore=1024MB` on x86 and `kernelcore=2048MB` on PPC64.

The list-based strategy is tested under light and heavy loads to determine if the strategy breaks down under pressure. We anticipated the results of the benchmarks to be similar if no breakdown was occurring. The zone-based strategy is tested with and without `kernelcore` to show that `ZONE_EASYRCLM` is behaving as expected and that the existence of the zone does not incur a performance penalty. The choice of 2048MB on PPC64 is 50% of physical memory. The choice of 1024MB on x86 is to give some memory to `ZONE_EASYRCLM`, but to leave some memory in `ZONE_HIGHMEM` for PTE use as `CONFIG_HIGHPTE` was set.

[3] http://www.csn.ul.ie/~mel/projects/vmregress/vmregress-0.37.tar.gz

8 Results

On the successful completion of a test run, a summarised report is generated similar[4] to the one shown in Figure 11. These reports get aggregated into the graphs shown in Figures 12 and 13. For each architecture the graphs show how the two strategies compare against the base allocator in terms of performance and the ability to satisfy HugeTLB allocations. These graphs will be the focus of our discussion on performance in Section 8.1.

Figures 5 and 6 shows the values of $F_u(hugetlb\_order)$ at each allocation attempt during the Highalloc-Stress Test while the system was under no load. Note that in all cases, the starting value of $F_u(hugetlb\_order)$ is close to 1 indicating that free memory was not in large contiguous regions after the kernel compiles were stopped. The value drops over time as pages are reclaimed and buddies coalesce. Kernels using anti-fragmentation strategies had a higher rate of decline for the value of $F_u(hugetlb\_order)$, which implies that the anti-fragmentation strategies had a measure of success. These figures will be the focus of our discussion on the ability of the system to satisfy requests for contiguous regions in Section 8.2.

Finally, Figures 7 and 8 show the value of $F_i(hugetlb\_order)$ at each allocation failure during the Highalloc-Stress Test while the system was under no load. These illustrate the root cause of the allocation failures and are discussed in Section 8.3.

8.1 Performance

On both architectures, absolute performance was comparable. The "KBuild Comparison"

[4] Edited to fit

graphs in Figures 12 and 13 show the timings were within seconds of each other and this was consistent among runs. The "AIM9 Comparison" graphs show that any regression was within 3% of the base kernel's performance. This is expected as that test varies by a few percent in each run and the results represent one run, not an average. This leads us to conclude that neither list-based nor zone-based has a significant performance penalty on either x86 or PPC64 architectures, at least for our sample workloads.

8.2 Free Space Usability

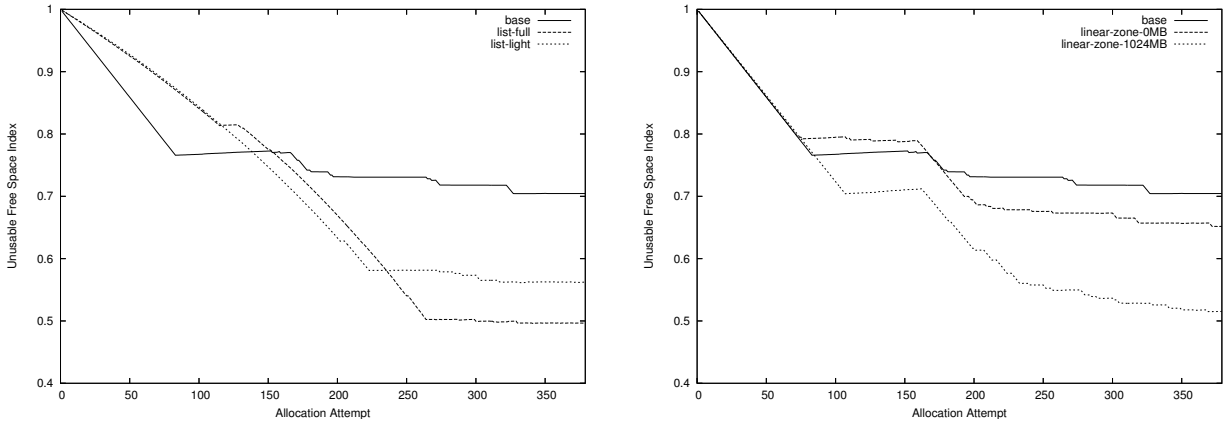
In general, zone-based was more predictable and reliable at providing contiguous free space. On both architectures, the zone-based anti-fragmentation kernels were able to allocate almost all of the pages in ZONE_EASYRCLM at rest after the tests. As shown on Figure 6, 0.66 was the final value of $F_u(hugetlb\_order)$ on PPC64 with half of physical memory in ZONE_EASYRCLM. We would expect it to reach 0.50 after multiple HugeTLB allocation attempts. Without specifying `kernelcore`, the scheme made no difference to absolute performance or fragmentation as ZONE_EASYRCLM is empty.

The list-based strategy was potentially able to reduce fragmentation throughout physical memory. On x86, list-based anti-fragmentation kept overall fragmentation lower than zone-based but it was only fractionally better on the PPC64 than the standard allocator. An examination of the x86 "High Allocation Stress Test Comparison Test" report in Figure 12 hints why. On x86, advantage is being taken of the existing zone-based groupings of allocation types in Normal and HighMem. Effectively, it was using a simple zone-based anti-fragmentation that did not take PTEs into account. The list-based strategy succeeds on x86 because it keeps the PTE pages in HighMem grouped together in addition to some success in ZONE_NORMAL. Nevertheless, the strategy clearly breaks down in ZONE_NORMAL due to large amounts of kernel allocations falling back to the EasyRclm freelists in low-memory situations. The breakdown is is illustrated by the different values of $F_u(hugetlb\_order)$ after the different loads where similar values would be expected if no breakdown was occurring. Figure 5 shows that the light-load performed *worse* than full-load due to the unpredictability of the strategy. On an earlier run, the list-based strategy under light load was able to allocate 119 HugeTLB pages from ZONE_NORMAL but only 77 after full-load.

Under load, neither scheme was significantly better than the other at keeping free areas contiguous. This is because we were depending on the LRU-approximation to reclaim a contiguous region. Under load, zone-based was generally better because page reclaim was able to reclaim within ZONE_EASYRCLM but the list-based strategy did not have the same focus. With either anti-fragmentation strategy, LRU simply is not suitable for reclaiming contiguous regions and an alternative strategy is discussed in Section 10.

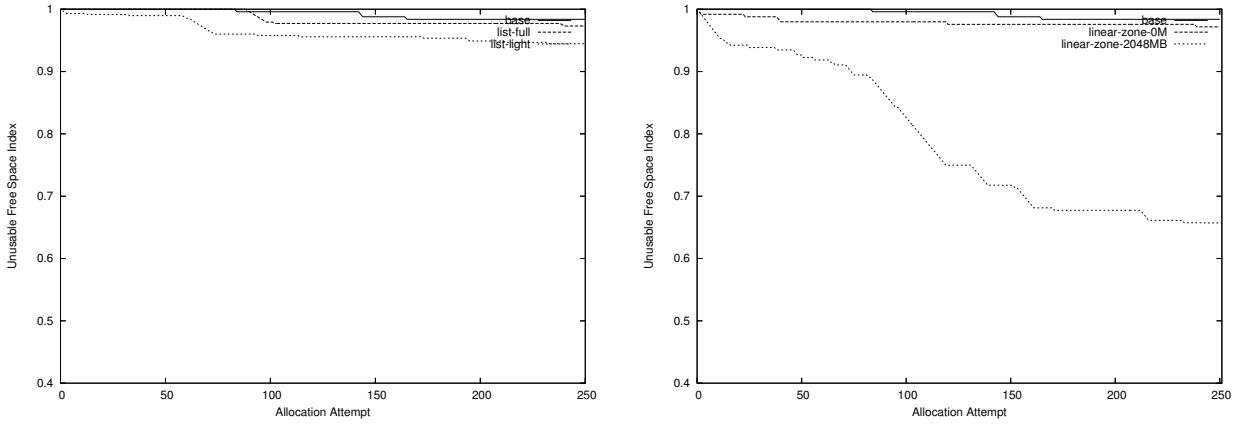
8.3 Fragmentation Index at Failure

Figures 7 and 8 clearly show that allocations failed with both strategies due to fragmentation and not lack of memory. By design, the zone-based strategy does not reduce fragmentation in the kernel zones. When an allocation fails at rest, it is because ZONE_EASYRCLM is likely nearly depleted and we are looking at the high fragmentation in the kernel zones. The figures for list-based implied that, under load, fragmentation had crept into all zones which means the strategy broke down due to excessive stealing.



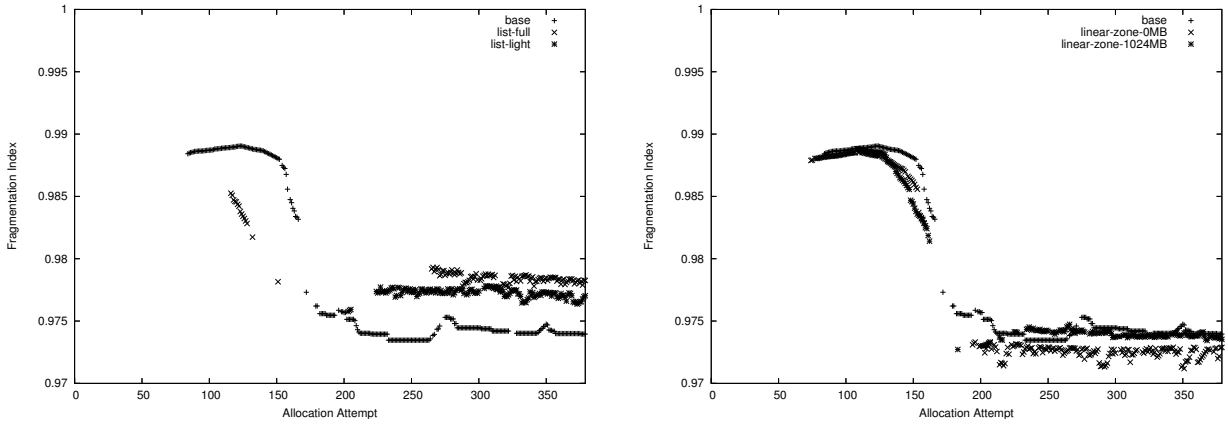
X86 LIST-BASED X86 ZONE-BASED

Figure 5: x86 Unusable Free Space Index During Highalloc-Stress Test, System At Rest



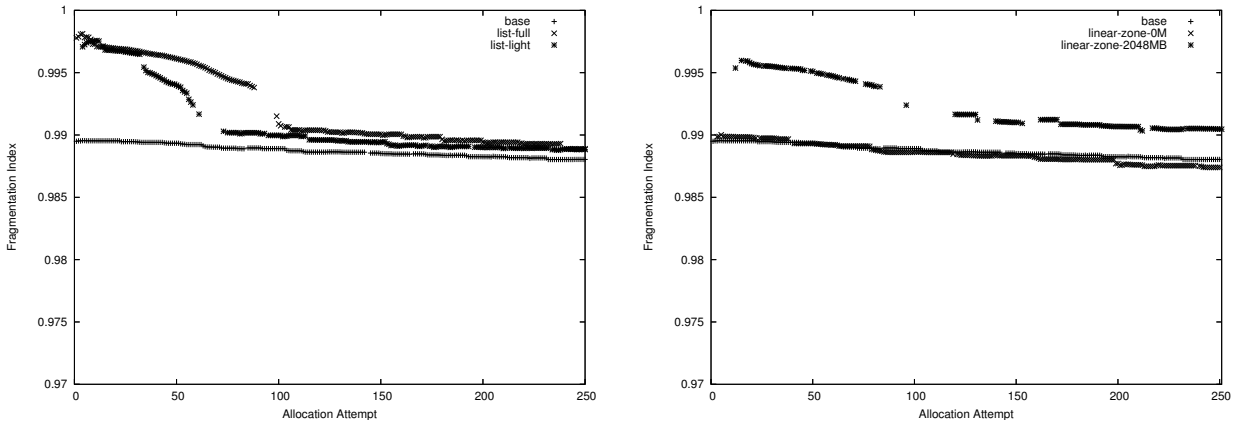
PPC64 LIST-BASED PPC64 ZONE-BASED

Figure 6: PPC64 Unusable Free Space Index During Highalloc-Stress Test, System At Rest



X86 LIST-BASED X86 ZONE-BASED

Figure 7: x86 Fragmentation Index at Allocation Failures During Highalloc-Stress Test



PPC64 LIST-BASED PPC64 ZONE-BASED

Figure 8: PPC64 Fragmentation Index at Allocation Failures During Highalloc-Stress

9 Results Conclusions

The two strategies had different advantages and disadvantages but both were able to increase availability of HugeTLB pages. The fact that list-based does not require configuration and works on all of memory makes it desirable but our figures show that it breaks down in its current implementation. Once correctly configured, zone-based is more reliable even though it does not help high-order kernel allocations.

The zone-based strategy is currently the best available solution. In the short-to-medium term, the zone-based strategy creates a soft-area that can satisfy HugeTLB allocations on demand. In the long-term, we intend to develop a strategy that takes the best from both approaches without incurring a performance regression.

10 Linear Reclaim

Anti-fragmentation improves our chances of finding contiguous regions of memory that may be reclaimed to satisfy a high order allocation. However, the existing LRU-approximation algorithm for page reclamation is not suitable for finding contiguous regions.

In the *worst-case* scenario, the LRU list contains randomly ordered pages across the system so the release of pages will also be in random order. To free a contiguous region of 2^j pages within a zone containing N pages, we may need to release $F_r(j)$ pages in that zone where

$$F_r(j) = (\frac{N}{2^j} * (2^j - 1)) + 1$$

The table in Figure 9 shows the relative proportion of memory we will need to reclaim before we can guarantee to free a contiguous region of sufficient size for the specified order. We can see that beyond the lowest orders we need to reclaim most pages in the system to guarantee freeing pages of the desired order. Order 10 and 12 are interesting as they represent the HugeTLB page sizes for x86 and PPC64 respectively. The average case is not this severe, but a detailed analysis of the average case is beyond the scope of this paper.

We introduced an alternative reclaim algorithm called *Linear Reclaim* designed to target larger

Order	Percentage
1	50.00
2	75.00
3	87.50
4	93.75
5	96.88
6	98.44
10	99.90
12	99.98

Figure 9: Reclaim Difficulty

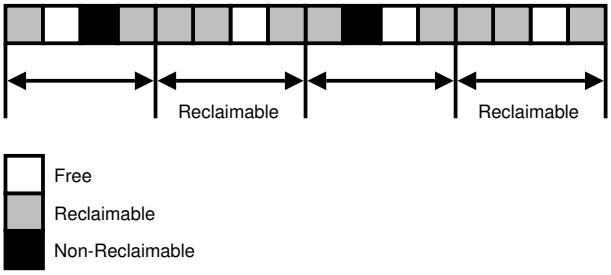


Figure 10: Linear Reclaim

contiguous regions of pages. It is used when the failing allocation is of order 3 or greater. With linear reclaim we view the entire memory space as a set of contiguous regions, each of the size we are trying to release. For each region, we check if all of the pages are likely to be reclaimable or are already free. If so, the allocated pages are removed from the LRU and an attempt is made to reclaim them. This continues until a proportion of the contiguous regions have been scanned.

In our example in Figure 10, linear reclaim will only attempt to reclaim pages in the second and fourth regions, applying reclaim to all the pages in the selected region at the same time. It is clear that in the case where reclaim succeeds we should be able to free the region by releasing just its pages which is significantly less than that required with LRU-based reclaim.

An early proof-of-concept implementation of linear reclaim was promising. A HugeTLB-capability test was run on the x86 machine. Under load, a clean kernel was able to allocate 6 HugeTLB pages, the zone-based anti-fragmentation allocator was able to allocate 10 HugeTLB pages and with both zone-based anti-fragmentation and linear-reclaim, it was able to allocate 41 HugeTLB pages. We do not have detailed timing information but early indications are that linear reclaim is able to satisfy allocation requests faster but spends more time scanning than the existing page reclamation policy before a failure. In summary, linear reclaim is promising, but needs further development.

11 Future Work

We intend to develop the zone-based anti-fragmentation strategy further. The patches that exist at the time of writing include some complex architecture-specific code that calculate the size of `ZONE_EASYRCLM`. As the code for sizing zones and memory holes in each architecture is similar, we are developing code to calculate the size of zones and holes in an architecture-independent fashion. Our initial patches show a net reduction of code.

Once an anti-fragmentation strategy is in place, we would like to develop the linear reclaim scanner further as LRU reclaims far too much memory to satisfy a request for a contiguous region. Our current testing strategy records how long it takes to satisfy a large allocation and we anticipate linear reclaim will show improvements in those figures.

In a perfect world, with everything in place, the plan is to work on the transparent support of HugeTLB pages in Linux. Although there are known applications that benefit from this such as database and java-based software, we would

also like to show benefits for desktop software such as X.

We will then determine if there is a performance case for the use of higher-order allocations by the kernel. If there is, we will revisit the list-based approach and determine if a more general solution can be developed to control fragmentation throughout the system, and not just in pre-configured zones.

Acknowledgements

We would like to thank Nishanth Aravamudan for reviewing a number of drafts of this paper and his suggestions on how to improve the quality. We would like to thank Dave Hansen for his clarifications on the content, particularly on the size of SPARSEMEM memory sections on x86. Finally, we would like to thank Paul McKenney for his in-depth commentary on an earlier version of the paper and particularly for his feedback on Section 3.

Legal Statement

References

[1] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In *PLDI*, pages 187–196, 1993.

[2] J. B. Chen, A. Borg, and N. P. Jouppi. A simulation based study of TLB performance. In *ISCA*, pages 114–123, 1992.

[3] D. G. Korn and K.-P. Bo. In search of a better malloc. In *Proceedings of the Summer 1985 USENIX Conference*, pages 489–506, 1985.

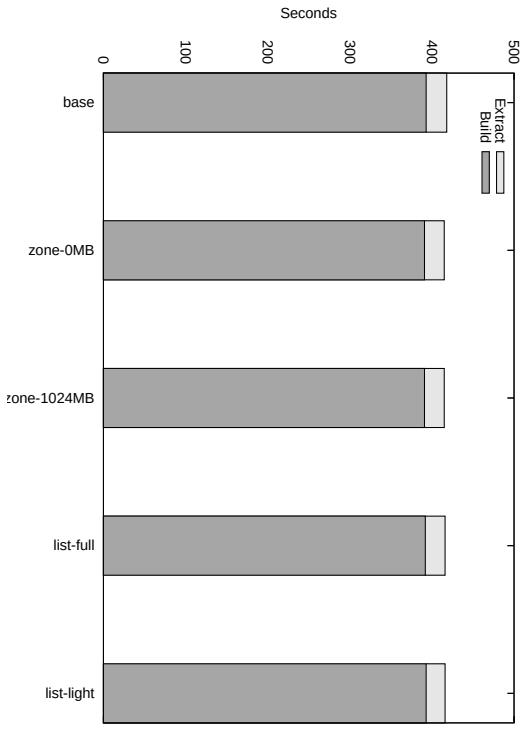
[4] M. K. McKusick. *The design and implementation of the 4.4BSD operating system*. Addison-Wesley, 1996.

[5] J. L. Peterson and T. A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

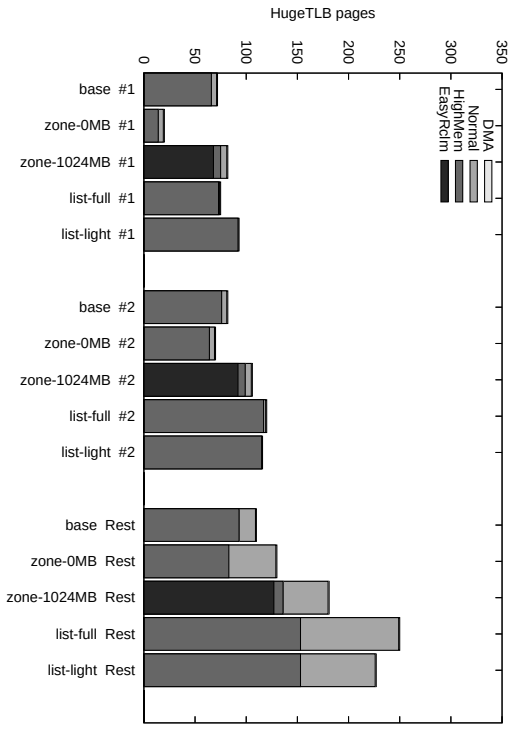
[6] B. Randell. A note on storage fragmentation and program segmentation. *Commun. ACM*, 12(7):365–369, 1969.

```
Kernel comparison report
------------------------
Architecture:             x86
Huge Page Size:           4 MB
Physical memory:          1554364 KB
Number huge pages:        379

KBuild Comparison
-----------------
                              2.6.16-rc6-clean  2.6.16-rc6-zone-0MB  2.6.16-rc6-zone-1024MB
Time taken to extract kernel:               25                   24                      24
Time taken to build kernel:                393                  391                     391

AIM9 Comparison
---------------
                 2.6.16-rc6-clean    zone-0MB         zone-1024MB
 1 creat-clo           105965.67   105866.67 -0.09%    106500.00  0.50% File Creations and Closes/s
 2 page_test           259306.67   271558.07  4.72%    258300.28 -0.39% System Allocations & Pages/s
 3 brk_test           1666572.24  1866883.33 12.02%   1880766.67 12.85% System Memory Allocations/s
 4 jmp_test          14805650.00 13949966.67 -5.78%  15088700.00  1.91% Non-local gotos/second
 5 signal_test         286252.29   280183.33 -2.12%    282950.00 -1.15% Signal Traps/second
 6 exec_test              131.79      131.98  0.14%       131.68 -0.08% Program Loads/second
 7 fork_test             3857.69     3842.69 -0.39%      3862.69  0.13% Task Creations/second
 8 link_test            21291.90    21693.58  1.89%     21499.37  0.97% Link/Unlink Pairs/second

High Allocation Stress Test Comparison
--------------------------------------
HighAlloc Under Load Test Results Pass 1
                          2.6.16-rc6-clean  2.6.16-rc6-zone-0MB  2.6.16-rc6-zone-1024MB
Order                                   10                   10                      10
Success allocs                          72                   20                      82
Failed allocs                          307                  359                     297
DMA zone allocs                          1                    1                       1
Normal zone allocs                       5                    5                       6
HighMem zone allocs                     66                   14                       7
EasyRclm zone allocs                     0                    0                      68
% Success                               18                    5                      21
HighAlloc Under Load Test Results Pass 2
                          2.6.16-rc6-clean  2.6.16-rc6-zone-0MB  2.6.16-rc6-zone-1024MB
Order                                   10                   10                      10
Success allocs                          82                   70                     106
Failed allocs                          297                  309                     273
DMA zone allocs                          1                    1                       1
Normal zone allocs                       5                    5                       6
HighMem zone allocs                     76                   64                       7
EasyRclm zone allocs                     0                    0                      92
% Success                               21                   18                      27
HighAlloc Test Results while Rested
                          2.6.16-rc6-clean  2.6.16-rc6-zone-0MB  2.6.16-rc6-zone-1024MB
Order                                   10                   10                      10
Success allocs                         110                  130                     181
Failed allocs                          269                  249                     198
DMA zone allocs                          1                    1                       1
Normal zone allocs                      16                   46                      44
HighMem zone allocs                     93                   83                       9
EasyRclm zone allocs                     0                    0                     127
% Success                               29                   34                      47

HugeTLB Page Capability Comparison
----------------------------------
                                 2.6.16-rc6-clean  2.6.16-rc6-zone-0MB  2.6.16-rc6-zone-1024MB
During compile:                                 5                    5                       5
At rest before dd of large file:               51                   52                      48
At rest after  dd of large file:               67                   64                      92
```

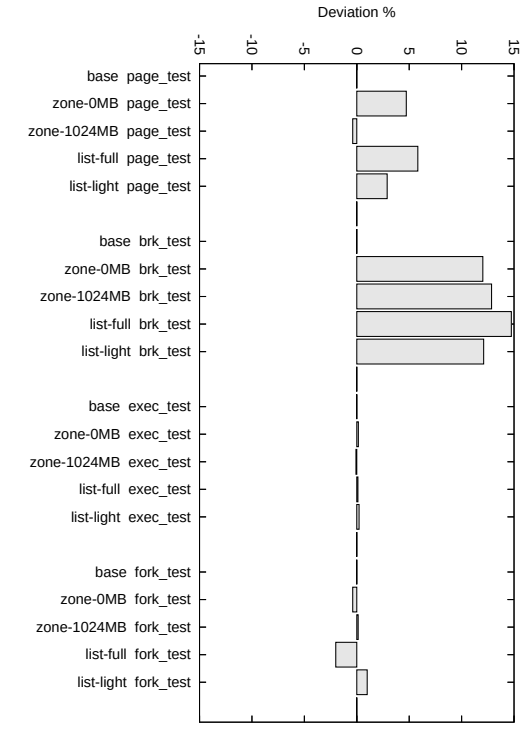
Figure 11: Example Kernel Comparison Report



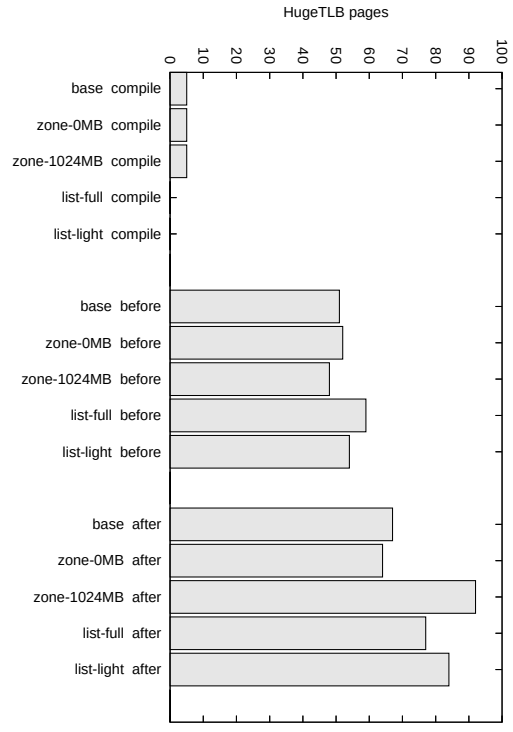
KBUILD COMPARISON



HIGH ALLOCATION STRESS TEST COMPARISON

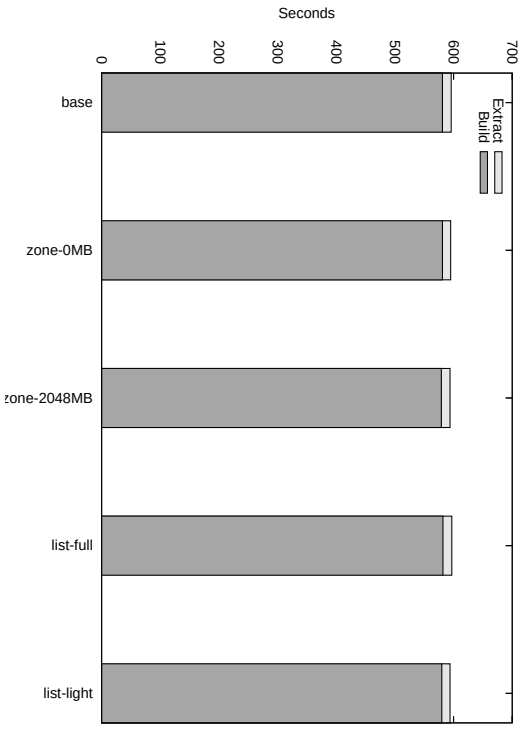


AIM9 COMPARISON

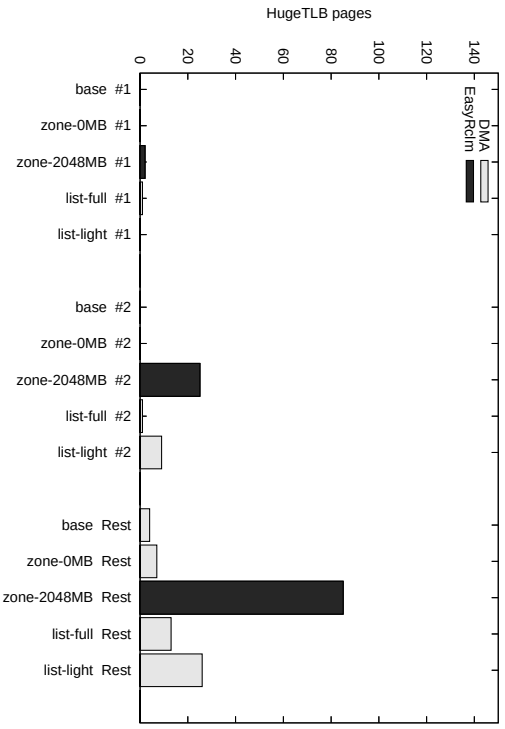


HUGETLB PAGE CAPABILITY COMPARISON

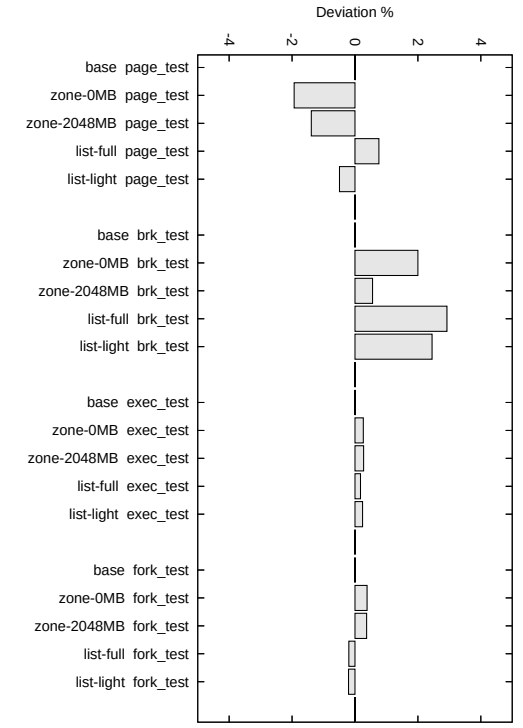
Figure 12: Anti-Fragmentation Strategy Comparison on x86



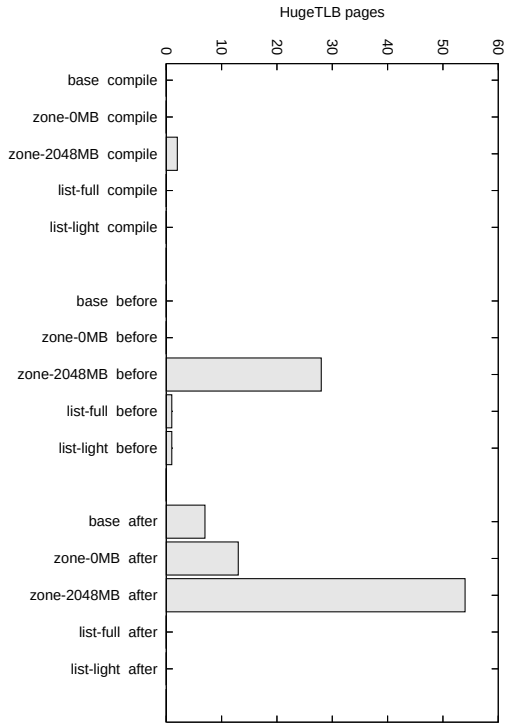
KBUILD COMPARISON



HIGH ALLOCATION STRESS TEST COMPARISON



AIM9 COMPARISON



HUGETLB PAGE CAPABILITY COMPARISON

Figure 13: Anti-Fragmentation Strategy on PPC64

Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin