Supporting the Allocation of Large Contiguous Regions of Memory

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

Andy Whitcroft
IBM
andyw@uk.ibm.com

Abstract

Some modern processors such as later Opterons® and Power® processors are able to support large pages sizes such as 1GiB and 16GiB. These page sizes are impractical to reserve at boot time because of the amount of memory that is potentially wasted. Currently, Linux® as it stands is not well suited to support multiple page sizes because it makes no effort to satisfy allocations for contiguous regions of memory. This paper will discuss features under development that aim to support the allocation of large contiguous areas.

This paper begins by discussing the current status of mechanisms to reduce external fragmentation in the page allocator. The reduction of external fragmentation results in sparsely populated superpages that must be reclaimed for contiguous allocations to succeed. We describe how poor reclaim decisions offset the performance benefits of superpages in low-memory situations, before introducing a mechanism for the intelligent reclaim of contiguous regions. Following a presentation of metrics used to evaluate the features and the results, we propose a memory compaction mechanism that migrates pages from sparsely populated to dense regions when enough memory is free to avoid reclaiming pages. We conclude by highlighting that parallel allocators prevent contiguous allocations by taking free pages from regions being reclaimed. We propose a method for addressing this by making pages temporarily unavailable to allocators.

1 Introduction

Any architecture supporting virtual memory is required to map virtual addresses to physical addresses through an address translation mechanism. Recent translations are stored in a cache called a *Translation Lookaside Buffer (TLB)*. *TLB Coverage* is defined as memory addressable through this cache without having to access the master tables in main memory. When the master table is used to resolve a translation, a *TLB Miss* is incurred. This can have as significant an impact on *Clock cycles Per Instruction (CPI)* as CPU cache misses [3]. To compound the problem, the percentage of memory covered by the TLB has decreased from about 10% of physical memory in early machines to approximately 0.01% today [6]. As a means of alleviating this, modern processors support multiple page sizes, usually up to several megabytes, but gigabyte pages are also possible. The downside is that processors commonly require that physical memory for a page entry be contiguous.

In this paper, mechanisms that improve the success rates of large contiguous allocations in the Linux kernel are discussed. Linux already supports two page sizes, referred to as the *base page* and the *huge page*. The paper begins with an update on previous work related to the placement of pages based on their mobility [2]. A percentage of pages allocated by the kernel are movable due to the data being referenced by page tables or trivially discarded. By grouping these pages together, contiguous areas will be moved or reclaimed to satisfy large contiguous allocations. As a bonus, pages used by the kernel are grouped in areas addressable by fewer large TLB entries, reducing TLB misses.

Work on the intelligent reclaim of contiguous areas is then discussed. The regular reclaim algorithm reclaims base pages but has no awareness of contiguous areas. Poor page selections result in higher latencies and lower success rates when allocating contiguous regions.

Metrics are introduced that evaluate the placement policy and the modified reclaim algorithm. After describing the test scenario, it is shown how the placement policy keeps a percentage of memory usable for contiguous allocations, and the performance impact is discussed. It is then shown how the reclaim algorithm satisfies the allocation of contiguous pages faster than the regular reclaim.

The remainder of the paper proposes future features to improve the allocation of large contiguous regions. Investigation showed that contiguous areas were sparsely populated and that compacting memory would be a logical step. A memory compaction mechanism is proposed that will be rarely triggered, as an effective placement policy significantly reduces the requirement for compaction [1] [4]. Additional investigation revealed that parallel allocators use pages being reclaimed, preempting the contiguous allocation. This is called the *racing allocator* problem. It is proposed that pages being reclaimed be made unavailable to other allocators.

2 External Fragmentation

External fragmentation refers to the inability to satisfy an allocation because a sufficiently sized free area does not exist despite enough memory being free overall [7]. Linux deals with external fragmentation by rarely requiring contiguous pages. This is unsuitable for large, contiguous allocations. In our earlier work, we defined metrics for measuring external fragmentation and two mechanisms for reducing it. This section will discuss the current status of the work to reduce external fragmentation. It also covers how page types are distributed throughout the physical address space.

2.1 Grouping Pages By Mobility

Previously, we grouped pages based on their ability to be reclaimed and called the mechanism *anti-fragmentation*. Pages are now grouped based on their ability to be moved, and this is called *grouping pages by mobility*. This takes into account the fact that pages `mlock()`ed in memory are movable by page migration, but are not reclaimable.

The standard buddy allocator always uses the smallest possible block for an allocation, and a minimum number of pages are kept free. These free pages tend to remain as contiguous areas until memory pressure forces a split. This allows occasional short-lived, high-order allocations to succeed, which is why setting `min_free_kbytes` to 16384[1] benefits Ethernet cards using jumbo frames. The downside is that once split, there is no guarantee that the pages will be freed as a contiguous area. When grouping by mobility, the minimum number of free pages in a zone are stored in contiguous areas. Depending on the value of `min_free_kbytes`, a number of areas are marked `RESERVE`. Pages from these areas are allocated when the alternative is to fail.

Previously, bits from `page→flags` were used to track individual pages. A bitmap now records the mobility type of pages within a `MAX_ORDER_NR_PAGES` area. This bitmap is stored as part of the `struct zone` for all memory models except the `SPARSEMEM`, where the memory section is used. As well as avoiding the consumption of `page→flags` bits, tracking page type by area controls fragmentation better.

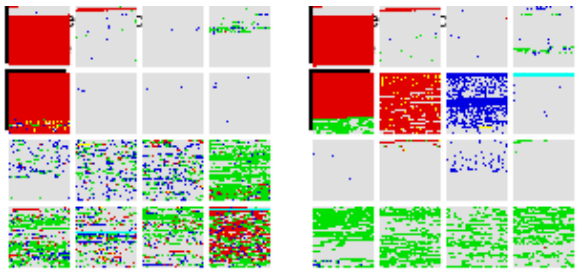
Previously effective control of external fragmentation required that `min_free_kbytes` be 5% to 10% of physical memory, but this is no longer necessary. When it is known there will be bursts of high-order atomic allocations during the lifetime of the system, `min_free_kbytes` should be increased. If it is found that high order allocations are failing, increasing `min_free_kbytes` will free pages as contiguous blocks over time.

2.2 Partitioning Memory

When grouping pages by mobility, the maximum number of superpages that may be allocated on demand is dependent on the workload and the number of movable pages in use. This level of uncertainty is not always desirable. To have a known percentage of memory available as contiguous areas, we create a zone called `ZONE_MOVABLE` that only contains pages that may be migrated or reclaimed. Superpage allocations are not movable or reclaimable but if a **sysctl** is set, superpage allocations are allowed to use the zone. Once the zone is sized, there is a reasonable expectation that the superpage pool can be grown at run-time to at least the size of `ZONE_MOVABLE` while leaving the pages available for ordinary allocations if the superpages are not required.

The size in bytes of the `MAX_ORDER` area varies between systems. On x86 and x86-64 machines, it is 4MiB of data. On PPC64, it is 16MiB; on an IA-64 supporting huge pages, it is often 1GiB. If a developer requires a 1GiB superpage on Power there is no means to providing it, as the buddy allocator does not have the necessary free-lists. Larger allocations could be satisfied by increasing `MAX_ORDER`, but this is a compile-time change and not desirable in the majority of cases. Generally, supporting allocations larger than `MAX_ORDER_`

[1] A common recommendation when using jumbo frames.



Standard Allocator Grouping By Mobility

Figure 1: Distribution of Page Types

NR_PAGES requires adjacent MAX_ORDER_NR_PAGES areas to be the same mobility type. The memory partition provides this guarantee.

The patches to group pages by mobility and partition memory were merged for wider testing in the Linux kernel version 2.6.21-rc2-mm2 despite some scepticism on the wider utility of the patches. Patches to allocate adjacent MAX_ORDER_NR_PAGES are planned.

2.3 Comparing Distribution

Figure 1 shows where pages of different mobility types are placed with the standard allocator and when grouping by mobility. Different colours are assigned to pages of different mobility types and each square represents MAX_ORDER_NR_PAGES. When the snapshot was taken the system had been booted, a 32MiB file was created and then deleted. It is clear in the figure that MAX_ORDER_NR_PAGES areas contain pages of mixed types with the standard allocator, but the same type when when grouping by mobility. This "mixing" in the standard allocator means that contiguous allocations are not likely to succeed even if all reclaimable memory is released.

It is clear from the figure that grouping pages by mobility does not guarantee that all of memory may be allocated as contiguous areas. The algorithm depends on a number of areas being marked MOVABLE so that they may be migrated or reclaimed. To a lesser extent it depends on RECLAIMABLE blocks being reclaimed. Reclaiming those blocks is a drastic step because there is no means to target the reclamation of kernel objects in a specific area.

Despite these caveats, the result when grouping pages by mobility is that a high percentage of memory may be allocated as contiguous blocks. With memory partitioning, a known percentage of memory will be available.

3 Reclaim

When an allocation request cannot be satisfied from the free pool, memory must be reclaimed or compacted. Reclaim is triggered when the free memory drops below a watermark, activating **kswapd**, or when available free memory is so low that memory is reclaimed directly by a process.

The allocator is optimised for base page-sized allocations, but the system generates requests for higher order areas. The regular reclaim algorithm fares badly in the face of such requests, evicting significant portions of memory before areas of the requested size become free. This is a result of the *Least Recently Used (LRU)* reclaim policy. It evicts pages based on age without taking page locality into account.

Assuming a random page placement at allocation and random references over time, then pages of similar LRU age are scattered throughout the physical memory space. Reclaiming based on age will release pages in random areas. To guarantee the eviction of two adjacent pages requires 50% of all pages in memory to be reclaimed. This requirement increases with the size of the requested area tending quickly towards 100%. Awareness is required of the size and alignment of the allocation or the performance benefits of superpage use are offset by the cost of unnecessarily reclaiming memory.

3.1 Linear Area Reclaim

To fulfil a large memory request by reclaiming, a contiguous aligned area of pages must be evicted. The resultant free area is then returned to the requesting process. Simplistically this can be achieved by linearly scanning memory, applying reclaim to suitable contiguous areas. As reclaim completes, pages coalesce into a contiguous area suitable for allocation.

The linear scan of memory ignores the age of the page in the LRU lists. But as previously discussed, scanning based on page age is highly unlikely to yield a contiguous area of the required size. Instead, a hybrid of these two approaches is used.

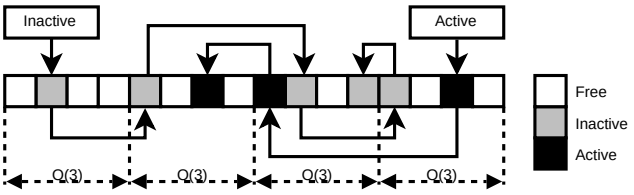


Figure 2: Area Reclaim area selection

3.2 LRU-Biased Area Reclaim

LRU-Biased Area Reclaim (Area Reclaim)[2] is a hybrid reclaim algorithm incorporating linear area reclaim into the regular LRU-based aging algorithm. Instead of linearly scanning memory for a suitable area, the tails of the active and inactive LRU lists are used as a starting point.

Area Reclaim follows the regular approach of targeting pages in the LRU list order. First, an area in the active list is selected based on the oldest page in that list. Active pages in that area are rotated to the inactive list. An area in the inactive list is then selected based on the oldest page in that list. All pages on the LRU lists within this area are then reclaimed. When there is allocation pressure at a higher order, this tends to push groups of pages in the same area from the active list onto the head of the inactive list increasing the chances of reclaiming areas at its tail in the future. On the assumption there will be future allocations of the same size, **kswapd** applies pressure at the largest size required by an in-progress allocation.

Figure 2 shows an example memory layout. As an example, consider the application of Area Reclaim at order-3. The active and inactive lists work their way through the pages in memory indicating the LRU-age of those pages. In this example, the end of the active list is in the second area and the end of the inactive list is in the third area. Area Reclaim first applies pressure to the active list, targeting the second area. Next it applies pressure to the inactive list, targeting area three. All pages in this area are reclaimable or free and it will coelesce.

While targeting pages in an area, we maintain the age-based ordering of the LRU to some degree. The downside is that younger, active and referenced pages in the same area are all targeted prematurely. The higher the order of allocation, the more pages that are unfairly treated. This is unavoidable.

[2]This was initially based on Peter Zijlstra's modifications to last year's Linear Reclaim algorithm, and is commonly known as Lumpy Reclaim.

x86-64 Test Machine

CPU	Opteron® 2GHz
# Physical CPUs	2
# CPUs	4
Main Memory	1024MiB

PPC64 Test Machine

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

Figure 3: Specification of Test Machines

4 Experimental Methodology

The mechanisms are evaluated using three tests: one related to performance, and two that are known to externally fragment the system under normal conditions. Each of the tests is run in order, without intervening reboots, to maximise the chances of the system being fragmented. The tests are as follows.

kernbench extracts the kernel and then builds it three times. The number of jobs **make** runs is the number of processors multiplied by two. The test gives an overall view of the kernel's performance and is sensitive to changes that alter scalability.

HugeTLB-Capability is unaltered from our previous study. For every 250MiB of physical memory, a kernel compile is executed in parallel. Attempts are made to allocate hugepages through the kernel's `proc` interface under load and at the end of test.

Highalloc-Stress builds kernels in parallel, but in addition, **updatedb** runs so that the kernel will make many small unmovable allocations. Persistent attempts are made to allocate hugepages at a constant rate such that **kswapd** should not queue more than 16MiB/s I/O. Previous tests placed 300MiB of data on the I/O queue in the first three seconds, making comparisons of reclaim algorithms inconclusive.

All benchmarks were run using driver scripts from VM-Regress 0.80[3] in conjunction with the system that generates the reports on `http://test.kernel.org`. Two machines were used to run the benchmarks based on 64-bit AMD® Opteron and POWER5® architectures, as shown in Figure 3.

On both machines, a minimum free reserve of $5 * MAX\_ORDER\_NR\_PAGES$ was set, representing 2% of physical memory. The placement policy is effective with a lower reserve, but this gives the most predictable results. This is due to the low frequency that page types are mixed under memory pressure. In contrast, our previous study required fives times more space to reduce the frequency of fallbacks.

5 Metrics

In this section, five metrics are defined that evaluate fragmentation reduction and the modified reclaim algorithm. The first metric is *system performance*, used to evaluate the overhead incurred when grouping pages by mobility. The second metric is overall allocator *effectiveness*, measuring the ability to service allocations. Effectiveness is also used as the basis of our third metric, *reclaim fitness*, measuring the correctness of the reclaim algorithm. The fourth metric is *reclaim cost*, measuring the processor and I/O overheads from reclaim. The final metric is *inter-allocation latency*, measuring how long it takes for an allocation to succeed.

5.1 System Performance

The performance of the kernel memory allocator is critical to the overall system. Grouping pages by mobility affects this critical path and any significant degradation is intolerable. The **kernbench** benchmark causes high load on the system, particularly in the memory allocator. Three compilation runs are averaged, giving processor utilisation figures to evaluate any impact of the allocator paths.

5.2 Effectiveness

The effectiveness metric measures what percentage of physical memory can be allocated as superpages

$$E = (A_r * 100)/A_t$$

where A_r is the number of superpages allocated and A_t is the total number of superpages in the system. During the **Highalloc-Stress** test, attempts are made to allocate superpages under load and at rest, and the effectivess is measured. Grouping pages by mobility should show an increase in the effectiveness when the system is at rest at the end of a test. Under load, the metric is a key indicator of the intelligent reclaim algorithm's area selection.

5.3 Reclaim Fitness

Reclaim fitness validates the reclaim algorithm. When applied to 100% of memory, any correct algorithm will achieve approximately the same effectiveness. A low variance implies a correct algorithm. An incorrect algorithm may prematurely exit or fail to find pages.

5.4 Reclaim Cost

The reclaim cost metric measures the overhead incurred by scanning, unmapping pages from processes, and swapping out. Any modification to the reclaim algorithm affects the overall cost to the system. Cost is defined as:

$$C = log_{10}((C_s * W_s) + (C_u * W_u) + (C_w * W_w))$$

where C_s is the number of pages scanned, C_u is the number of pages unmapped, and C_w is the number of pages we have to perform I/O for in order to release them. The scaling factors are chosen to give a realistic ratio of each of these operations. W_s is given a weight of 1, W_u is weighted as 1000 and W_w is weighted at 1,000,000.

The cost metric is calculated by instrumenting the kernel to report the scan, unmap, and write-out rates. These are collected while testing effectiveness. This metric gives a measure of the cost to the system when reclaiming pages by indicating how much additional load the system experiences as a result of reclaim.

[3] `http://www.csn.ul.ie/~mel/projects/vmregress/vmregress-0.80.tar.gz`

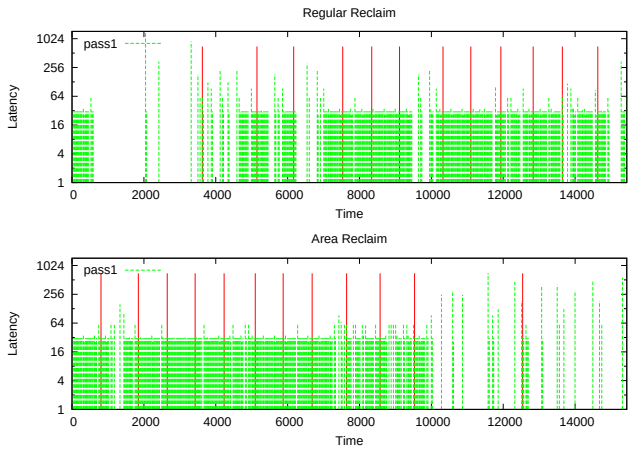


Figure 4: Inter-Allocation Latencies

x86-64 Test Machine

CPU Time	Mobility		Delta (%)
	Off	On	
User	85.83	86.78	1.10
System	35.92	34.07	–5.17
Total	121.76	120.84	–0.75

PPC64 Test Machine

CPU Time	Mobility		Delta (%)
	Off	On	
User	312.43	312.24	–0.06
System	16.89	17.24	2.05
Total	329.32	329.48	0.05

Figure 5: Mobility Performance Comparison

5.5 Inter-Allocation Latency

Given a regular stream of allocation attempts of the same size, the inter-allocation latency metric measures the time between successful allocations. There are three parts to the metric. The inter-allocation latency variability is defined as the standard deviation of the inter-allocation delay between successful allocations. The mean is the arithmetic mean of these inter-allocation delays. The worst-case allocation time is simply the worst inter-allocation delay.

The inter-allocation times are recorded when measuring effectivess. Figure 4 shows the raw inter-allocation times for regular and Area Reclaim. The fixed-height vertical red lines indicate where an additional 5% of memory was allocated as superpages. A vertical green bar indicates when an allocation succeeded and the height indicates how long since the last success. The large portions of white on the left side of the regular reclaim graph indicate the time required for enough memory to be reclaimed for contiguous allocations to succeed. In contrast, Area Reclaim does not regularly fail until 50% of memory is allocated as superpages.

A key feature of a good reclaim algorithm is to provide a page of the requested size in a timely manner. The inter-allocation variability metric gives us a measure of the consistency of the algorithm. This is particularly significant during the transition from low to high allocator load, as allocations are time critical.

6 Results

Four different scenarios were tested: two sets of runs evaluating the effectiveness and performance of the page mobility changes, and two sets of runs evaluating the Regular and Area Reclaim algorithms.

Figure 5 shows the elapsed processor times when running **kernbench**. This is a `fork()`-, `exec()`-, I/O-, and processor-intensive workload and a heavy user of the kernel page allocator, making it suitable for measuring performance regressions there. On x86-64, there is a significant and measurable improvement in system time and overall processor time despite the additional work required by the placement policy. In Linux, the kernel address space is a linear mapping of physical memory using superpage *Page Table Entries (PTEs)*. Grouping pages by mobility keeps kernel-related allocations together in the same superpage area, reducing TLB misses in the kernel portion of the working set. A performance gain is found on machines where the kernel portion of the working set exceeds TLB reach when the kernel allocations are mixed with other allocation types.

In contrast, the PPC64 figures show small performance regressions. Here, the kernel portion of the address space is backed by superpage entries, but the PPC64 processor has at least 1024 TLB entries, or almost ten times the number of x86-64. In this case, the TLB is able to hold the working set of the kernel portion whether pages are grouped by mobility or not. This is compounded by running **kernbench** immediately after boot, causing allocated physical pages to be grouped together. PPC64 is expected to show similar improvements due

to reduced TLB misses with workloads that have large portions of their working sets in the kernel address space and when the system has been running for a long time.

Where there are no benefits due to reduced TLB misses in the kernel portion of a working set, regressions of between 0.1% and 2.1% are observed in **kernbench**. However, the worst observed regression in overall processor time is 0.12%. Given that **kernbench** is an unusually heavy user of the kernel page allocator and that superpages potentially offer considerable performance benefits, this minor slowdown is acceptable.

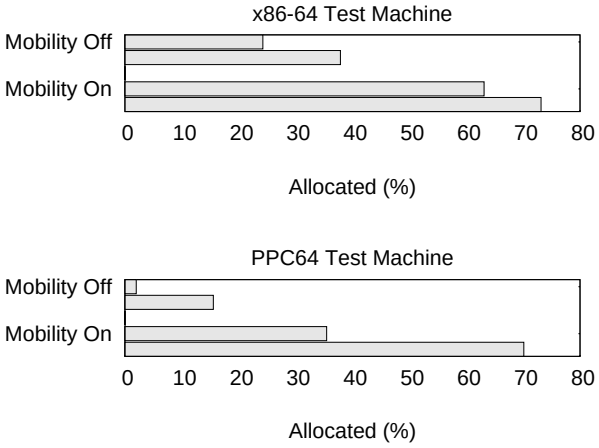


Figure 6: Mobility Effectiveness

Figure 6 shows overall effectiveness. The pair of bars show the percentage of memory successfully allocated as superpages under load during the **Highalloc-Stress** stress and at the end when the system is at rest. The figures show that grouping the pages significantly improves the success rates for allocations. In particular on PPC64, dramatically fewer superpages were available at the end of the tests in the standard allocator. It has been observed that the longer systems run without page mobility enabled, the closer to 0% the success rates are.

Figure 7 compares the percentage of memory allocated as huge pages at the end of all test for both reclaim algorithms. Both algorithms should be dumping almost all of memory, so the figures should always be comparable. The figures are comparable or improved, so it is known that Area Reclaim will find all reclaimable pages when under pressure. This validates the Area Reclaim scanner.

Arch	Reclaim		Delta (%)
	Regular	Area	
x86-64	64.53	73.15	8.62
PPC64	72.11	70.12	-1.99

Figure 7: Correctness

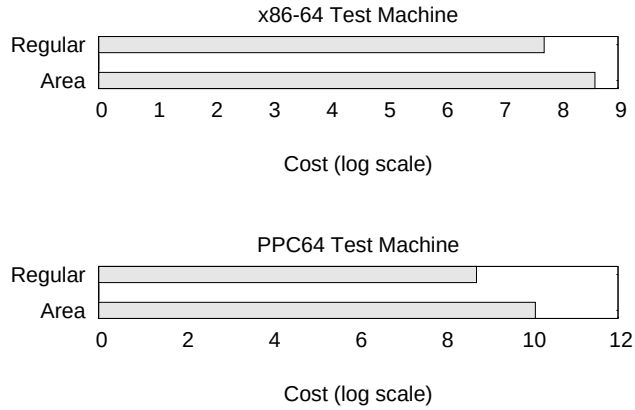


Figure 8: Cost Comparison

Figure 8 shows that the cost of Area Reclaim are higher than those of Regular reclaim, but not by a considerable margin. Later we will show the the time taken to allocate the required areas is much lower with Area Reclaim and justifies the cost. Cost is a trade-off. On one hand the algorithm must be effective at reclaiming areas of the requested size in a reasonable time. On the other, it must avoid adversely affecting overall system utility while it is in operation.

The first row of graphs in Figure 9 shows the inter-allocation variability as it changes over the course of the test. Being able to deliver ten areas 20 seconds after they are requested is typically of no use. Note that in both algorithms, the variability is worse towards the start, with Area Reclaim significantly out-performing Regular Reclaim.

The second row of graphs in Figure 9 shows the mean inter-allocation latency as it changes over the course of the test. It is very clear that the Area Reclaim inter-allocation latency is more consistent for longer, and generally lower than that for Regular reclaim.

The third row of graphs in Figure 9 shows the worst-case inter-allocation latency. Although the actual worst-case latencies are very similar with the two algorithms, it is clear that Area Reclaim maintains lower latency for much longer. The worst-case latencies for Area Reclaim are at the end of the test, when very few potential contiguous regions exist.

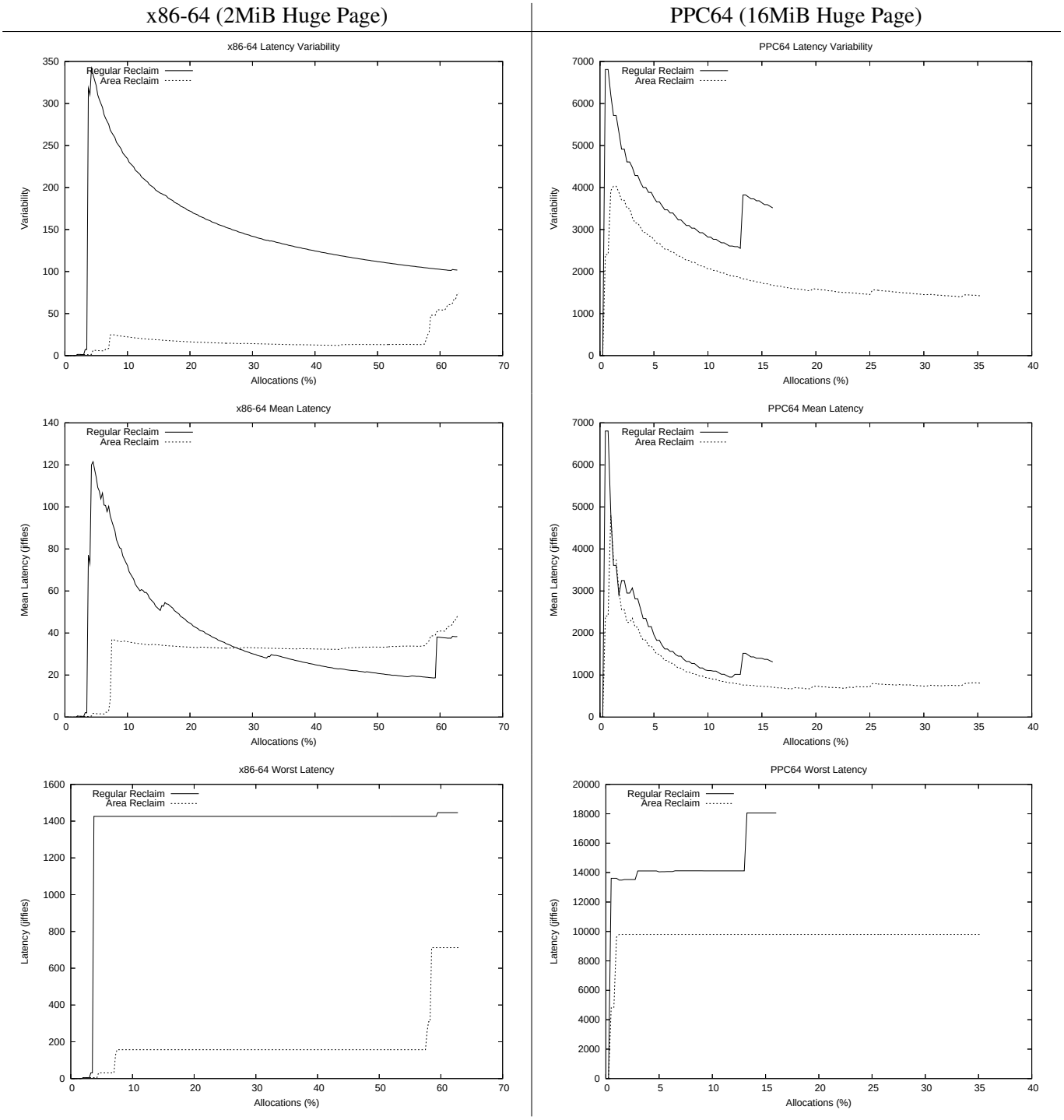


Figure 9: Latency Comparison

The superpage size on PPC64 is eight times larger than that on x86-64. Increased latencies between the architectures is to be expected, as reclaiming the required contiguous area is exponentially more difficult. Inspection of the graphs shows that Area Reclaim allocates superpages faster on both x86-64 and PPC64. The increased effectiveness of the Area Reclaim algorithm is particularly noticeable on PPC64 due to the larger superpage size.

6.1 Results Summary

The effectiveness and performance comparisons show that grouping pages by mobility and Area Reclaim is considerably better at allocating contiguous regions than the standard kernel, while still performing well. The cost and latency metrics show that for a moderate increase in work, we get a major improvement in allocation latency under load. A key observation is that early allocations under load with Area Reclaim have considerably lower latency and variability in comparison to the standard allocator. Typically a memory consumer will not allocate all of memory as contiguous areas, but allocate small numbers of contiguous areas in short bursts. Even though it is not possible to anticipate bursts of contiguous allocations, they are handled successfully and with low latency by the combination of grouping pages by mobility and Area Reclaim.

7 Memory Compaction

Reclaim is an expensive operation and the cost might exceeed the benefits of using superpages, particularly when there is enough memory free overall to satisfy the allocation. This paper proposes a memory compaction mechanism that moves pages so that there are fewer, but larger and contiguous, free areas. The term defragmentation is avoided because it implies all pages are movable, which is not the case in Linux. At the time of writing, no such memory compaction daemon has been developed, although a page migration mechanism[5] does exist in the kernel.[4]

7.1 Compaction Mechanism

The compaction mechanism will use the existing page migration mechanism to move pages. The page mobility type information stored for a `MAX_ORDER_NR_PAGES` area will be used to select where pages should be migrated. Intuitively the best strategy is to move pages from sparsely to densely populated areas. This is unsuitable for two reasons. First, it involves a full scan of all pages in a zone and sorting areas based on density. Second, when compaction starts, areas that were previously sparse may no longer be sparse unless the system was frozen during compaction, which is unacceptable.

The compaction mechanism will group movable pages towards the end of a zone. When grouping pages by mobility, the location of unmovable pages is biased towards the lower addresses, so these strategies work in conjunction. As well as supporting the allocation of very large contiguous areas, biasing the location of movable pages towards the end of the zone potentially benefits the hot-removal of memory and simplifies scanning for free pages to migrate to.

When selecting free pages, the *free page scanner* begins its search at the end of a zone and moves towards the start. Areas marked `MOVABLE` are selected. The free pages contained within are removed from the free-lists and stored on a private list. No further scanning for free pages occurs until the pages on the private list are depleted.

When selecting pages to move, the *migration scanner* searches from the start of a zone and moves towards the end. It searches for pages that are on the LRU lists, as these are highly likely to be migratable. The selection of an area is different when a process or a compaction daemon is scanning. This process will be described in the next two sections.

A compaction run always ends when the two scanners meet. At that point, it is known that it is very unlikely that memory can be further compacted unless memory is reclaimed.

7.2 Direct Compaction

At the time of an allocation failure, a process must decide whether to compact or reclaim memory. The extent of external fragmentation depends on the size of the allocation request. In our previous paper, two metrics were defined that measure external fragmentation. They are reintroduced here, but not discussed in depth except as they apply to memory compaction.

[4] As implemented by Christoph Lameter.

Fragmentation index is the first metric and is only meaningful when an allocation fails due to a lack of a suitable free area. It determines if the allocation failure was due to a lack of free memory or external fragmentation. The index is calculated as

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

where $TotalFree$ is the number of free pages, j is the order of the desired allocation, and $AreasFree$ is the number of contiguous free areas of any size. When $AreasFree$ is 0, $F_i(j)$ is defined as 0.

A value tending towards 0 implies the allocation failed due to lack of memory and the process should reclaim. A value tending towards 1 implies the failure is due to external fragmentation and the process should compact. If a process tries to compact memory and fails to satisfy the allocation, it will then reclaim.

It is difficult to know in advance if a high fragmentation index is due to areas used for unmovable allocations. If it is, compacting memory will only consume CPU cycles. Hence when the index is high but before compaction starts, the index is recalculated using only blocks marked `MOVABLE`. This scan is expensive, but can take place without holding locks, and it is considerably cheaper than unnecessarily compacting memory.

When direct compaction is scanning for pages to move, only pages within `MOVABLE` areas are considered. The compaction run ends when a suitably sized area is free and the operation is considered successful. The steps taken by a process allocating a contiguous area are shown in Figure 10.

1. Attempt allocation
2. On success, return area
3. Calculate fragmentation index
4. If low memory goto 11
5. Scan areas marked MOVABLE
6. Calculate index based on MOVABLE areas alone
7. If low memory goto 11
8. Compact memory
9. Attempt allocation
10. On success, return block
11. Reclaim pages
12. Attempt allocation
13. On success, return block
14. Failed, return NULL

Figure 10: Direct Compaction

7.3 Compaction Daemon

kswapd is woken when free pages drop below a watermark to avoid processes entering direct reclaim. Similarly, **compactd** will compact memory when external fragmentation exceeds a given threshold. The daemon becomes active when woken by another process or at a timed interval.

There are two situations were **compactd** is woken up to unconditionally compact memory. When there are not enough pages free, grouping pages by mobility may be forced to mix pages of different mobility types within an area. When a non-movable allocation is improperly placed, **compactd** will be woken up. The objective is to reduce the probability that non-movable allocations will be forced to use a area reserved for `MOVABLE` because movable pages were improperly placed. When **kswapd** is woken because the high watermark for free pages is reached, **compactd** is also woken up on the assumption that movable pages can be moved from unmovable areas to the newly freed pages.

If not explicitly woken, the daemon will wake regularly and decide if compaction is necessary or not. The metric used to make this determination is called the *unusable free space index*. It measures what fraction of available free memory may be used to satisfy an allocation of a specific size. The index is calculated as

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

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, F_u is defined as 1.

By default the daemon will only be concerned with allocations of order-3, the maximum contiguous area normally considered to be reasonable. Users of larger contiguous allocations would set this value higher. Compaction starts if the unusable free space index exceeds 0.75, implying that 75% of currently free memory is unusable for a contiguous allocation of the configured size.

In contrast to a process directly reclaiming, the migrate scanner checks all areas, not just those marked MOVABLE. The compaction daemon does not exit until the two scanners meet. The pass is considered a success if the unusable free space index was below 0.75 before the operation started, and above 0.75 after it completes.

8 Capturing Page Ranges

A significant factor in the efficiency of the reclaim algorithm is its vulnerability to racing allocators. As pressure is applied to an area of memory, pages are evicted and released. Some pages will become free very rapidly as they are clean pages; others will need expensive disk operations to record their contents. Over this period, the earliest pages are vulnerable to being used in servicing another allocation request. The loss of even a single page in the area prevents it being used for a contiguous allocation request, rendering the work Area Reclaim redundant.

8.1 Race Severity

In order to get a feel for the scale of the problem, the kernel was instrumented to record and report how often pages under Area Reclaim were reallocated to another allocator. This was measured during the **Highalloc-stress** test as described in Section 4.

The results in Figure 11 show that only a very small portion of the areas targeted for reclaim survive to be allocated. Racing allocations steal more than 97% of all areas before they coalesce. The size of the area under reclaim directly contributes to the time to reclaim and the chances of being disrupted by an allocation.

Arch	Area (MB)	Allocations		Rate (%)
		Good	Raced	
x86-64	2	500	19125	2.61
PPC64	16	152	13544	1.12

Figure 11: Racing Allocations

8.2 Capture

If the reallocation of pages being reclaimed could be prevented, there would be a significant increase in success rates and a reduction in the overall cost for releasing those areas. In order to evaluate the utility of segregating the memory being released, a prototype capture system was implemented. Benchmarking showed significant improvement in reallocation rates, but triggered unexpected interactions with overall effectiveness and increased the chances of the machine going out-of-memory. More work is required.

9 Future Considerations

The intelligent reclaim mechanism focuses on the reclaim of LRU pages because the required code is well understood and reliable. However, a significant percentage of areas are marked RECLAIMABLE, usually meaning that they are backed by the slab allocator. The slab allocator is able to reclaim pages, but like the vanilla allocator, it has no means to target which pages are reclaimed. This is particularly true when objects in the dentry cache are being reclaimed. Intelligently reclaiming slab will be harder because there may be related objects outside of the area that need to be reclaimed first. This needs investigation.

Memory compaction will linearly scan memory from the start of the address space for pages to move. This is suitable for **compactd**, but when direct compacting, it may be appropriate to select areas based on the LRU lists. Tests similar to those used when reclaiming will be used to determine if the contiguous area is likely to be successfully migrated. There are some potential issues with this, such as when the LRU pages are already at the end of memory, but it has the potential to reduce stalls incurred during compaction.

The initial results from the page capture prototype are promising but, they are not production-ready and need further development and debugging. We have considered making the direct reclaim of contiguous regions synchronous to reduce the latency and the number of pages reclaimed.

10 Acknowledgements

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. We would like to thank Steve Fox, Tim Pepper, and Badari Pulavarty for their helpful comments and suggestions on early drafts

of this paper. A big thanks go to Dave Hansen for his insights into the interaction between grouping pages by mobility and the size of the PPC64 TLB. Finally, we would like to acknowledge the help of the community with their review and helpful commentary when developing the mechanisms described in this paper.

Legal Statement

References

[1] P. J. Denning. Virtual memory. *Computing Surveys*, 2(3):153–189, Sept. 1970.

[2] M. Gorman and A. Whitcroft. The what, the why and the where to of anti-fragmentation. In *Ottawa Linux Symposium 2006 Proceedings Volume 1*, pages 361–377, 2006.

[3] Henessny, J. L. and Patterson, D. A. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[4] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley Publishing Co., Reading, Massachusetts, 2 edition, 1973.

[5] C. Lameter. Page migration. *Kernel Source Documentation Tree*, 2006.

[6] J. E. Navarro. *Transparent operating system support for superpages*. PhD thesis, 2004. Chairman-Peter Druschel.

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

Proceedings of the Linux Symposium

Volume One

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

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

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*