*Reprinted from the*

# Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

# Bazillions of Pages

The Future of Memory Management under Linux

Christoph Lameter
*Silicon Graphics, Inc.*
christoph@lameter.com

## Abstract

A new computer system running Linux is likely equipped with at least 4GB of memory. The Linux VM manages this memory in 4KB chunks. So the Linux VM has to manage 1 million memory chunks. There are some people who already run configurations with tens or hundreds of gigabytes of memory. As time progresses these large memory sizes are going to become more and more common. The Linux VM will have to manage more and more pages. For effective memory reclaim these pages may have to be repeatedly scanned in order to determine the least recently used pages.

It is not surprising that the VM starts to struggling with the increasing amount of work. At 8 to 16GB one can observe live lock situations with certain loads. A number of possible solutions to this problem are considered: One is Rik van Riel's work of optimizing the way pages are handled in the VM, another is Andrea Arcangeli's increase in the base page size. And yet another is to make the page size dynamic in order to allow subsystems to choose the page size that is most beneficial for a given load.

## 1 Introduction

### 1.1 4 megabytes are lots of memory

My first Linux installation was done using a set of floppy disks containing Slackware (1.0.4) and a strange kernel version that was numbered 0.99 followed by some trailing letters and numbers that I cannot remember. It was 1993 and I had to download 12 disk images through a dial up line which took almost a week. The download was using an advanced file transfer protocol named Z-modem. The Slackware software was installed on a machine that had a 386SX processor (no floating point unit) and 4 megabytes of memory. The machine was already a big step forward from my first computer, a PET 2001 (Commodore Business Machines) that I gained access to in 1978. The PET had 4KB of memory. The 386SX machine was great and I thought really had plenty of memory especially since MS-DOS could only use 640 kilobytes.

### 1.2 More and more memory

Fast forward to today and now I work at Silicon Graphics on making Linux run well on Supercomputers. This adds another strange twist since I get to work with machines that have thousands of times more memory than an average computer. SGI has customers with machines equipped with several petabytes of RAM. And given the way the capacities develop: It may take less than a decade until we get to machines that have memory sizes in the exabyte range.[1]

So there is the chance of seeing how Linux handles super large memory sizes years before they are available in smaller computers for everyone. The time delay is about a decade or so before these become available in an average computer. Linux servers with one terabyte of RAM will likely become available around 2010, memory sizes of a petabyte may be possible by 2018. Machines like that will also have a couple of hundred processors (cores?) accessing that memory.[2]

I see how Linux runs with large memory sizes years before the same memory sizes are available to the general public. If there is a hard issue related to memory then it frequently ends up on my desk. The advantage for Linux is that we have a chance to prepare the Linux kernel for the upcoming memory issues years before they become a problem for the general users of Linux.

[1]One exabyte has a million gigabytes and a petabyte a thousand gigabytes.

[2]See Intel's plans for large numbers of cores[6].

One of the areas of concern is that memory sizes keep growing while processor speeds and memory access speeds are mostly stagnating. Memory management in Linux occurs by managing a small piece of meta data (the `struct page`) for each 4KB chunk of memory (a *page*.) Whenever the kernel needs to perform I/O, when a page fault occurs, or when the kernel is handling memory in some other way then the meta data in `struct page` is used for synchronization and for tracking the state of the page.

## 2 The memory problem

The available computing resources to manage the meta data in the `struct page` are shrinking because memory sizes grow faster than processing speeds. Each new generation of hardware grows the number of 4KB pages that have to be managed. The VM therefore has to deal with an ever growing number of pages (bazillions of pages because bazillions is an undetermined large number.) It looks like the trend will continue for the foreseeable future.[3]

As a result we see critical OS activities like memory reclaim taking a larger and larger portion of computing resources. For memory reclaim, the page expiration is based on examining all the the meta data structures (in `struct page`) at some point. I/O bottlenecks also develop because each 4KB page has to be handled separately for DMA transfers.[4]

The larger the dataset that is streamed through the system becomes, the larger the scaling issues that we will encounter due to the meta data that has to be kept for each 4KB page that is processed by the I/O layer.

Table 1 shows the development of memory sizes, processor capabilities and memory access speed for an average computer (simplified.) The number of pages increases faster than the number of cores and the speed of memory. Various hardware tricks are used in memory subsystems to improve speed because we cannot change the basic physical limitations of how DRAM works. Many of the optimizations are based on the assumption of linear memory accesses, other optimizations are increasing the number of bits that can be fetched simultaneously. Processors have to manage larger and larger CPU caches in multiple layers (L1, L2, L3, maybe L4 soon?) to avoid the penalty of memory accesses. Memory accesses become more expensive as the distance between the processor and memory increases. The optimizations make a relatively small amount of memory accessible without long latencies and favor linear accesses to memory.

| Year | Mem | Pages | ClckFreq | Cores | MSpeed |
|---|---|---|---|---|---|
| 1993 | 4MB | 1024 | 16Mhz | 1 | 70ns |
| 2001 | 32MB | 8192 | 300Mhz | 1 | 60ns |
| 2005 | 1GB | 256000 | 1-2 Ghz | 1 | 55ns |
| 2008 | 4GB | 1 million | 2-3Ghz | 2 | 50ns |
| 2010? | 64GB | 16 million | 2-3Ghz | 4-8 | 45ns? |
| 2020? | 128TB | 32 billion | 3-4Ghz | 128 | 40ns? |

Table 1: Memory sizes and processors

In such an environment memory locality becomes an important consideration for improving the speed of the computer system as a whole. The management of ever larger lists of `struct page` referring to pages which over time become distributed seemingly randomly all over the memory of the system does not have a beneficial effect on performance.

The ratio of cache to memory size is also falling following a similar trajectory. The cost of a random memory access will become more and more expensive over time since the chance of hitting an object in the CPU cache by chance is reduced.

The situation for Supercomputer configurations is worse (the following data is for SGI Altix 3700/4700) because the memory sizes are much larger while the processor and memory are similar to other contemporary hardware. On the other hand the number of processors is larger but the number of processors also grows slower than the total amount of memory. There is the additional complexity of scaling the synchronization methods:

Table 2 shows the development of the sizes for Supercomputers. To some extent the massive amount of page meta data was reduced on Itanium by increasing the page size. A 16KB page size means that there are only one fourth of the pages to worry about. The I/O subsystems can perform linear DMA transfers for 16KB

[3]Maybe there will be a plateau when the limits of what is addressable within a 64 bit address space (at 8 exabytes) is reached. Not likely to occur until 2024 or so for the average Linux server but I would expect memory sizes like that to be reached by 2014 in the supercomputer area.

[4]Supercomputer I/O is expected to be able to saturate the links to the storage subsystem which contains large RAID configurations (thousands of disks.) Being able to just saturate the bandwidth of a single hard disk is certainly not acceptable.

| Year | Mem | Procs | PageSize | Pages |
|---|---|---|---|---|
| 2004 | 8TB | 512 | 16K (ia64) | 512 million |
| 2006 | 16TB | 1024 | 16K | 1 billion |
| 2007 | 4PB | 2048 | 16k | 256 billion |
| 2008 | 4PB | 4096 | 64k | 64 billion |
| 2008 | 16TB | 4096 | 4k(x86) | 4 billion |
| 2014? | 1EB? | 16384? | 4k(x86) | 256 trillion |

Table 2: Supercomputer memory sizes and processors

chunks which reduces the number of scatter gather entries that have to be managed by the storage subsystem. The increase to a page size of 64KB in 2008 decreases the management overhead again but the overall number of pages still stays comparatively high. We noticed that configurations with over 2 thousand processors only run reliably with 64KB pages. Otherwise the VM regularly gets into fits while handling large queues of pages even if running a HPC load that requires minimal I/O.

The successor to the Altix series will no longer be based on Itanium processors but on x86 architecture. We can no longer utilize large page sizes, but have to use the only page size that x86 supports which is 4KB. Ironically these systems are intended to support even larger memory sizes but the current physical addressing capabilities of the Xeon line limit the amount of physical memory that a single Linux instance will be able to access. The address size limitations effectively reduce the number of pages to be managed by a Linux instance but will force the construction of large shared memory machines running a number of Linux instances that each live in their own 16TB memory segment. Then we need specialized hardware that bridges between the Linux instances. Not a nice picture but the limit on the physical memory that can be addressed by a processor also limits the number of page structs that have to be managed by the kernel.

The address space sizes are likely to be increased as processor development continues. The future generations of processor will then be able to reach similar memory capacities as the Itanium based systems are capable of today. The number of page structs that have to be managed by the VM makes a jump of several orders for similar memory sizes on Itanium versus Xeon. There are at least 4 times more page structs compared to an Itanium system with a 16k page size or even 16 times more page structs for systems currently running 64KB page size.

## 2.1 The VM heart attack

Let's consider a typical scenario that can lead to a system appearing to live lock. The more processors and memory are involved, the more likely these are to occur.

The Linux VM uses lists of pages in order to determine which 4KB pages contain information that is no longer worth keeping in memory. These lists establish the least used memory in the system and then the Linux VM can reclaim that memory for other uses. For each page it has to be determined if it was used since the last scan by checking a referenced bit. The VM **must** therefore visit all pages regularly in order to correctly expire pages from memory. The more memory there is, the larger these lists become. The more allocations are performed, the more aggressive the VM has to scan and then trim pages via these lists.

As long as there is no lock contention and reclaim passes only take a small amount of time everything will be okay. The system is not under much memory pressure and has a reasonable chance to find freeable memory with short scans.

If memory allocations continue to occur and multiple processes start to expire memory then extensive scanning may result. Since we keep on adding more processors (due to the increasing use of multi-core technology) lock contention may also result because multiple processors attempt to reclaim memory simultaneously from the same memory range. Each has to wait for the lock in order to be allowed to scan the list. These lists are protected by spin locks and so other processors are waiting by spinning on the lock. The more processors the more likely live lock scenarios can develop due to starvation or simple slowdown because processors have to wait for locks that are held for a long time.

On NUMA it is typical that things take a turn for the worse when direct reclaim is beginning to occur concurrently. At that point the VM is walking down potentially long zone lists. Most of those are remote and memory accesses are especially expensive. If concurrent reclaim occurs within the same zone from multiple remote processors then excessive latencies will slow reclaim down further. At the end a majority of the processors may be in reclaim and only minimal processor time may become available for user processes to make progress with data processing.

### 2.2 Randomized memory references

The larger the memory the higher the chances of TLB misses which may also slow the machine. TLBs can map 4KB or 2MB sections of memory. The number of TLBs that a processor can cache is limited. If the processor can handle 512 4KB TLB entries (like in the upcoming Nehalem processor) then the processor can access only 2M of memory without a TLB miss. The number of supported TLBs for 2 megabyte entries is only 64 which allows the access to 128 megabyte of memory. TLB misses can be fast if the page table entries are held in the processor cache but a cache miss of the page table entry can introduce significant latencies, and the larger the amount of memory the higher the chance of these misses.

With increased memory the accesses to data locations in memory will be more sparse if data is not allocated closely together. It is therefore becoming expensive to follow pointers to memory that has not recently been accessed, and TLB misses become more and more likely to occur. There is the increasing chance of CPU cache misses, and the memory architectures are optimized for neighboring memory accesses which cannot handle sparse memory accesses effectively.

There are therefore multiple reasons why it is advantageous to use memory that is near other memory that was recently accessed. However, the arrays of pointers to `struct page` that we currently use for various purposes in the VM have a problem here because they have no locality. [5] Pointers may go to arbitrary pages all over memory. For all practical purposes these pointer lists may degenerate to accesses that are seemingly random, defeating the highly developed logic put in the processor to prefetch memory. In the worst case these lists may cause a TLB miss for each page struct on the list.

### 2.3 I/O fragmentation

The result of degeneration of page lists to access to random locations in memory has another important consequence for I/O. The pages that are sent down to the I/O subsystem cannot be coalesced into larger linear chunks (which is typically possible for some time after boot because pages that follow each other are allocated in order.) And therefore the I/O devices have to do I/O via scatter/gather entries to bazillion of pages in seemingly random locations in memory. I/O devices suitable for Linux must support an ever increasing number of scatter gather entries. The scatter gather list complexity becomes a potential I/O performance bottleneck.

[5]The situation for page structs is still better than references to objects in the pages because page structs are placed in a special memmap area whereas other objects can be placed anywhere in memory.

## 3 Solutions for handling large amounts of memory

The main problem here is that the VM has to sift through too much meta data for key operations like memory allocation, reclaim, and I/O. Plus the meta data is seemingly randomly distributed over all of memory reducing optimizations that the memory subsystem could make. The following solutions are focusing on reducing the scanning effort, reducing the amount of memory references for key VM operations, and increasing the locality of access in order for CPU caches, TLBs, and memory subsystems to be able to optimize memory accesses.

### 3.1 Reclaim improvements

One approach is to look at the problems that are emerging with memory reclaim in the VM. Rik van Riel has, over the last years, investigated a variety of methods to improve memory reclaim. These methods result in a better determination of which pages to evict from memory utilizing new reclaim algorithms developed an academic setting. Among them are: ARC, ClockPro, CAR, and LIRS. [6] These improvements would allow faster expiration of pages by reducing scan time and allow a more accurate prediction of pages that may be needed in the future.

The other aspect of Rik and other developers work on reclaim is that methods are developed to exclude pages that are unreclaimable from the reclaim lists and lists are created that contain easily reclaimable pages. These methods reduce the scanning overhead and may reduce the problems that we currently see. They are particularly good for specialized loads that result in large numbers of unreclaimable pages. The current reclaim in the Linux VM has to scan unreclaimable pages again and again which is obviously good to avoid.

[6]See http://linux-mm.org for more information about these approaches.

These are interesting approaches that enhance page reclaim and allow us to manage even more page structs in a better way. However, they do not address the fundamental issue that there is too much meta data for the VM to handle. The advanced reclaim methods still require eventually scanning through all the pages. The number of pages is not reduced but keeps on growing while we make minimal progress in getting these advanced reclaim algorithms working in the Linux VM.

### 3.2 Increasing the default page size

The solution that we have adopted for Itanium is to change the default page size. On Itanium this is supported by the hardware, so it's easy to do and the increase in page size has a significant effect in reducing the VM overhead for those large machines.

The x86 platform only supports 4KB and 2MB (64-bit) page table entries. We could work with that and increase the default page size by installing multiple 4KB page table entries in order to simulate e.g. a 16KB or 64KB "page" like is done on IA64.

But it is not clear that one actually would want a larger default page size. The 4KB page size is appropriate for the executables and small files that are needed by the operating system. The main use of larger sized pages are for applications that either perform a large amount of I/O (databases, enterprise applications) or need large amounts of memory (HPC applications.)

The binary format is also affected. Current binaries are formatted to have data aligned on 4KB boundaries. If that is no longer the case then we either have to change the binary format or provide some sort of layer that allows 4KB aligned access although the default page size is larger. The easy solution out of that may be to simply redefine the binary format and rebuild a completely new distribution. But that would require some work on binutils, the linker, and the loader.

Another idea that avoids multiple 4KB PTEs per large page is to set the default page size to the next higher page size which is 2 megabytes on x86. Essentially we are using a PMD for a PTE. Such a specialized version of Linux could perhaps run as a guest inside a virtualized environment (KVM, Lguest) in order to allow the special HPC or Enterprise class applications to run. The applications would have to be compiled for an environment that has a 2MB base page size and we would need a minimal distribution to create essential binaries that are necessary for the application.

### 3.3 Optional Support for larger page sizes

Optional support for larger page sizes means that the binary format can be left intact. User space works as it always has. Without enabling additional options the behavior of the kernel does not change. Optional large page size support means that all existing kernel APIs to user space stay as they are. Large page support can be switched on for special purposes by–for example–formatting a disk with a larger block size than 4KB. Or one could create a pseudo file system in memory with a larger page size that is then mapped into a processes memory.

However, larger pages means that the VM must now support more frequent allocations of contiguous memory larger than a 4KB. Requests for contiguous memory may vary in size. Memory fragmentation in Linux may increase.

Linux already has fragmentation avoidance logic implemented by Mel Gorman and one active defragmentation method (lumpy reclaim.) Both are measures to keep contiguous memory available. Both increase the chance of being able to obtain contiguous memory beyond the size of a single page but neither can guarantee that a large contiguous memory chunk is available at any point in time. Any user of contiguous memory beyond 4KB must implement fallback measures that kick in if large contiguous memory is not available. In fallback we loose the localization of memory accesses and increase the cache footprint. Fallback usually is realized by managing lists of small 4KB pages.

These fallbacks are currently rare but with the increase of demand for larger allocations the fallbacks may become more frequent. Additional defragmentation measures could be needed to produce more contiguous memory to avoid fallbacks to lists of pages.

## 4 Virtualizable Compound Pages

*Virtualizable Compound Pages* are a first step to support allocations of varying sizes of larger pages in the kernel. An allocation request for a Virtualizable Compound Page will first attempt to allocate linear physical memory of the requested size (a real Compound Page.) Typically these allocations will be successful resulting in a large page useful to allow the localization of multiple objects, the use of large stacks, or temporary storage for various purposes.

If the page allocator does not have sufficient linear memory available then the fallback logic will allocate a series of 4KB pages and use the vmalloc functionality to allow the memory provided to be used as virtually contiguous memory. This increases the overhead because the processor needs to use a page table to look up the memory location of each 4KB chunk but the page table handling logic is usually integrated in the processor and therefore fast and entries are cached. However, memory may be physically dispersed which means the optimizations for linear access of the memory subsystem and I/O subsystems may not be triggered. An array of pointers to the pages has to be maintained as well.

Virtualizable Compound Pages have the disadvantage that the user of these pages must always be aware that the linear memory that was provided by the allocator may be virtual and that actual physical memory may not be contiguous. If for example, a device needs to operate on the memory of a Virtualized Compound Page then the device needs to perform scatter gather operations to the physical pages that constitute the Virtualized Compound Page.

Additional problems result if the processor needs to have TLB entries loaded via a processor trap (like on IA64.) In that case access to the virtualized memory requires the ability to handle the faults in the contexts that the virtualized compound is used. In the case of IA64 the use of Virtualizable Compounds for stacks is impossible because the trap mechanism itself depends on the availability of the stack. If the processor implements TLB lookups in hardware (like x86) then the use of Virtualizable Compounds for stack areas is possible.

Virtualizable Compound Pages allow to optimize two typical usage scenarios in the kernel:

### 4.1 Avoid vmalloc

The use of Virtualizable Compound Pages allows the reduction of the use of vmalloc'ed memory. If a Virtualizable Compound is used instead of vmalloc then the page allocator will typically be able to provide a contiguous physical memory area. No vmalloc is necessary unless memory is significantly fragmented and the antifragmentation measures have not produced enough linear memory. Therefore overall vmalloc use of the kernel is reduced. The need to go through a page table can be avoided and access to memory becomes more effective.

### 4.2 Fallback for higher order allocations

The other use of Virtual Compounds is to avoid higher order allocations that may fail. If higher order allocation requests are converted to Virtual Compounds then the kernel code can transparently handle situations in which memory is fragmented and no higher order pages are available. A typical use case are large buffers and stacks (would e.g. allow the use of significantly larger stack areas than currently possible.)

## 5 Variable order slab caches

It is easy to make slab allocations use various sizes of pages if the maximum number of objects is stored with each page. A variable order slab cache therefore does not need virtual mappings like provided through Virtualizable Compound Pages. Slab allocations can then be tuned to use more or less large pages depending on their availability. Allocation sizes can be cut back to lower sizes if memory fragmentation demands this. Allocations of large units that fail can be retried with a smaller allocation unit. This means that in extreme cases the effectiveness of the antifragmentation and defragmentation methods determines the extent to which large allocation units can be used. Therefore the speed and the locality of slab object allocations may depend on effective defragmentation.

The size of the allocation units also increases the likelihood that slab objects are allocated near one another. The resulting object locality reduces the TLB pressure commonly coming from pointer chasing because objects are distributed all over memory. An optimal configuration can be obtained if the allocation unit is made to fit the TLB size used for the kernel data segment. On x86_64 this is 2 megabytes, so if the allocation unit is set to 2 megabytes then most objects taken out from a given slab on one processor will come from the same 2 megabyte area that can be covered with a single TLB entry.

A larger allocation unit is particularly useful for slab caches with larger object sizes (1–3 kilobyte objects) because a larger allocation allows more effective placement with less wasted memory. Objects may not fit well into smaller sized allocations. In addition to more efficient placement, allocation requests are also able to use the fast path for a higher percentage of allocations. As a result, calls to the page allocator become less frequent.

The allocation using varying page sizes puts more stress on the antifragmentation and defragmentation methods of the page allocator. If a high number of allocations fail and requires a retry with a smaller allocation size then this method is not effective and it would be better to switch back to smaller allocation units.

## 6 Larger I/O buffers

One of the limitations under which Linux file systems suffer is that I/O must be managed in 4KB chunks because the maximum buffer size is constrained by the page size of the operating system. This size is 4KB on x86 and as a result file systems are basically on their own if they want to manage data in larger chunks of memory. Some file systems implement a layer that allows the management of larger buffers (as in XFS) using virtualized mapping which means managing a list of pages for each larger buffer. The additional overhead reduces the potential performance gain and results in potentially non localized memory accesses.

The small I/O buffers also limit the amount of contiguous I/O that can be reliably submitted via DMA transfers to devices. Small 4KB chunks of memory require the management of large scatter gather lists which may be a limiting factor in the I/O throughput of a device.

Some file systems (like the ext file systems) are limited in the size of the volumes they currently support because they track allocations in page size chunks. If the chunk size can be increased through the use of larger I/O buffers then the size of volumes can be increased. The meta data that has to be managed for a given volume size is reduced significantly which accelerates the operation of the file system. The scaling possible with larger buffers can breathe new life into old file systems that can now overcome their size and performance limitations.

Large page support requires modification to the way that page size is handled in functions that provide page cache operations. The page size is stored in the mapping structure that exists for each opened file in the system. The mapping structure must then be consulted during each page cache operation to determine the block size to use for a particular operation. The page size or buffer size can then be configured dynamically for each open file.

However, the use of large pages should not change the user API. In particular `mmap` semantics that are exposed to user space should not change. It needs to be possible to continue the mapping in 4KB chunks. This becomes possible if we allow mapping of 4KB segments of larger pages into an address space. The semantics of `mmap` are then preserved. One has to realize though that state for multiple of these 4KB chunks is kept in a single page struct. Only the large page as a whole can be dirtied or locked. Write and read operations must be performed using the block size set in the mapping.

Typically large pages up to 64KB can be supported by Linux file systems since the file systems already support platforms (IA64, Powerpc) that allow a 64KB base page size. The file system meta data structures are therefore already prepared to support block sizes up to 64KB. Support for larger sizes will require modifications to the file systems. The ability to set the page size per mapping may allow the design of entirely new file systems with support for a block size that can be configured per directory or per file.

The changes necessary for large page support are typically transparent for file systems. The `set_blocksize()` function will allow setting larger sizes than 4KB pages which will affect the raw block device.

Fallback occurs using Virtualizable Compound Pages. File systems can access the buffer data via linear access through the page tables in the fallback case. However devices may have to check if a virtually contiguous page is passed to the driver and then set up DMA with a scatter gather list of the physical pages that constitute the Virtualized Compound Page.

## 7 Memory Fragmentation

Memory fragmentation has been an issue for a long time for the Linux kernel. Over time the 4KB pages used for processes tend to get randomly distributed over memory which also has the effect of making allocations of large contiguous chunks of memory impossible. There is no problem as long as only 4KB page allocations are performed because memory of the same size is freed and allocated. However, larger allocations than 4KB have always been performed for the stacks on x86 (8KB, so two contiguous pages are required) and for certain slab caches where objects would have caused too much memory wastage if they would have been placed in a 4KB page. The risk of failing such an allocation was judged to be negligible. If such an 8KB allocation cannot be satisfied then the page allocator will continue reclaiming until two consecutive pages are available.

The larger the chunk of memory becomes, the larger the risk becomes that an allocation cannot be satisfied. The page allocator will typically attempt to continue page reclaim in order to generate contiguous pages for allocations up to order 3 (32KB.) Even larger allocations will fail after a single reclaim pass that failed to generate a sufficiently sized page.

### 7.1 Antifragmentation Measures

Antifragmentation measures avoid fragmentation by classifying the allocation according to object lifetimes and reclaimability. One result of antifragmentation measures is that reclaimable pages are allocated in special memory areas. A series of neighboring pages can then be reclaimed to obtain large contiguous regions. So there is some level of guarantee that reclaim will be able to open up contiguous areas of memory because there are no unreclaimable pages in the way.

Antifragmentation measures were added to support allocation of huge pages even after the system has been running for awhile and after system memory has become fragmented. Huge pages are becoming more important for applications since they allow the localization of memory accesses and the reduction of TLB pressure by the use of a single TLB entry for 2MB of memory. Huge pages are a crude way to reach optimal speed of a machine as long as we have no large page support in the VM.

Problems still exist with allocations that are marked unmovable. These allocations are mostly page tables and certain slab allocations. The situation could be improved by making page table pages movable.[7] The slab defrag functionality allows making slab objects movable but a support function must be provided for each slab cache to provide such functionality for each slab. A significant reduction of the number of unmovable allocations would be possible with these two measures.

[7]See Ross Biro's work on relocatable page table pages.

Somewhat fewer problems exist with allocations marked reclaimable. Reclaimable allocations are mainly used for slab allocations that can be reclaimed using shrinkers. The slab defrag measures add methods to these slabs that allow the targeted reclaim of objects in these caches. So these are already movable to some extend and the movability of these objects will increase as the reclaim methods for reclaimable slabs mature.

Antifragmentation measures cannot guarantee the availability of larger allocations. Antifragmentation measures only increase the likelihood that a large allocation will be successful. In the worst case the situation can degenerate into a state in which the categorization of allocations fails. Antifragmentation will never move pages but only sort the allocations according to its reclaimability which means that antifragmentation can be supported with minimal overhead.

### 7.2 Defragmentation

Defragmentation measures move pages or target specific pages that are in the way of generating a large contiguous section of memory. Defragmentation therefore involves more overhead than antifragmentation measures.

The one defragmentation method currently implemented in the kernel is *lumpy reclaim*. Lumpy reclaim works with movable pages, during reclaim we check if the neighboring pages could be freed. The freeing of adjacent pages then allows the merging of free pages to large contiguous chunks that could be used for large page allocations for the page cache, etc. However, lumpy reclaim does not apply to unmovable allocations and reclaimable allocations. Some additional work could make lumpy reclaim like methods work for reclaimable allocations.

Mel Gorman has a patch set that implements full fledged defragmentation. Defragmentation has much in common with memory hot plug. In both cases an area of memory is scanned and memory is then moved elsewhere. We could have a defragmentation solution in the kernel if we wanted to or needed to have such support.

## 8 The need for a better page allocator

The development of new functionality in the page allocator has been slow since the merge of the antifragmentation measures which was a controversial decision that took years to make.

There are a number of known problems:

### 8.1 Slow 4KB page allocations

The current buffering mechanism for 4KB pages (which one would expect to be of superior speed given the importance of 4KB allocations to the VM) is suffering from bloat and is inferior to the allocation speed of the slab allocators by some orders of magnitude. The result is that 4KB allocations frequently use the slab allocators instead of the page allocator. Various subsystems compensate by having their own buffering schemes to avoid the page allocator. All of that code could be avoided if the page allocator fast path could be made competitive in performance to what the slab allocators can do.

Ironically 4KB page allocations are often about 5% slower (uncontended case) than 8KB sized allocations. 8KB allocations bypass the 4KB buffering mechanism and therefore can avoid the list management overhead. Performance wise it seems to be best for a subsystem to allocate a large chunk of memory from the page allocator and then cut it into 4KB pieces on its own.

### 8.2 Issues with lock contention from multiple processors

Higher order allocations have a disadvantage: Access to the buddy free lists requires taking a zone lock which is—for most systems—a global lock. So multiple processors cannot simultaneously allocate memory from the page allocator. For 4KB sizes we have a buffering mechanism that avoids the locking (but creates overhead that hurts elsewhere.)

If multiple processors allocate memory continuously from the page allocator then we may end up with bouncing cache lines for the zone locks. This contention can even be observed with 4KB allocations if they are frequent because even the 4KB buffering scheme needs to go to the free lists once in a while to check out a new batch of pages.

### 8.3 More effective support for order N allocations

If the page allocator is presented with varying orders of allocations then it would be best if these would be satisfied from several different areas. If allocations of the same order came from the same memory area then fragmentation would be reduced. Such a scheme is an extension of the antifragmentation method of sorting the allocations according to their lifetime. We would also sort them by size.

### 8.4 Scaling memory reclaim

One important aspect of larger page support is that it addresses the reclaim issue. If the pages on the reclaim lists have a larger size then there are fewer of these pages for a given amount of memory. The number of page structs that have to be processed is reduced and therefore reclaim works in a more effective way.

Reclaim is currently problematic on a multitude of platforms. Even desktop loads can start to suffer from reclaim scaling if applications are pushed into heavy reclaim. Swapping of large applications can make the system feel sluggish permanently. One wonders if it would not be better to simply fail if there is not enough memory rather than have the system become so sluggish that it takes a long time even if one attempts to simply reboot the system to get rid of the memory reclaim problems.[8] In the HPC area it is already fairly common to abort an application if heavy reclaim occurs because the applications becomes unacceptably slow.

## 9 Transparent Huge and Giant page support

Support for varying sizes of pages for the page cache would allow transparent support for huge pages with minimal effort. Most of the page cache functions could be used directly by the huge page subsystem. The VM could be optimized to install PMDs instead of PTEs if the PTEs would fill the complete page table page at the lowest layer.

Ultimately it would be possible to get rid of the current huge page support. A small skeleton could be retained for backward compatibility. Having transparent huge page support would clean up special casing in the VM and make it easy for applications to use huge pages for various purposes without the use of special libraries.

Giant pages are 1GB sized mappings that are currently only supported by the most recent AMD processors. Transparent huge page support could be extended to

[8]The problem is in no way unique to Linux

also support the 1GB PUDs that these processors provide without the need to add yet another subsystem with special reservations. 1GB support would be a way to effectively manage memory for applications that may use several terabytes of memory.

## 10 Conclusion

Memory sizes are going to continue to increase, while processor speeds will continue to not make much headway. Further parallelization will occur by processor manufacturers increasing symmetric (multi core) and asymmetric (coprocessors) parallelism on the die.

Concurrency issues will therefore continue to dominate the development of operating systems. It is likely that we will see a ratio of over 4GB of memory per core. A single processor may have to handle about 1 million pages for reclaim or for I/O if we stay with the current scheme of handling memory in the VM. We will have to deal with this situation in some way. Either we need to develop ways to handle bazillions of pages or we need to reduce their number. However, optimal performance will only be reached through an effective reduction of the number of entities that the kernel has to handle.

The trend to processor specialization will continue since binding a task to a processor will allow effective use of the CPU caches and speed the operation of actions necessary repeatedly. This means that limiting I/O submission for a given cached dataset to a few processors makes sense. Also it may be useful to dedicate certain processors to the operating system for reclaim, defragmentation and similar memory intensive operations that would contaminate the caches of other processors executing mostly in user context.

## 11 References

## References

[1] Brim, Michael J. & James D. Speirs, *The Processor-Memory Gap: Current and Future Memory Architectures*. 2002. `http://pages.cs.wisc.edu/~mjbrim/personal/classes/752/report.ps`.

[2] Mahapatra, Nihar R. & Balakrishna Venkatrao, “The Processor-Memory Bottleneck: Problems and Solutions.” *Crossroads*, Volume 6, Issue 3es. ACM: New York, 1999.

[3] Marathe, Jaydeep P. *METRICS: Tracking Memory Bottlenecks via Binary Rewriting*. Master Thesis: North Carolina University, 2003. `http://www.lib.ncsu.edu/theses/available/etd-07132003-161530/unrestricted/etd.pdf`.

[4] “Optoelectronic Integration Overcoming Processor Bottlenecks” in *Science Daily*, August 4th, 2005. `http://www.sciencedaily.com/releases/2005/08/050804053723.htm`.

[5] Sutter, Herb, “The Free Lunch is Over: A Fundamental Turn toward Concurrency in Software.” *Dr. Dobb’s Journal*, (30)3, March 2005.

[6] Bokar, Shekhar, Pradeep Dubey, Kevin Kahn, David Kuck, Hans Mulder, Steve Pawlowski and Justin Rattner ”Platform 2015: Intel Processor and Platform Evolution for the Next Decade.” *Technology Intel Magazine*, Intel Corporation: March 2005.