Local and Remote Memory: Memory in a Linux/NUMA System

Jun 20th, 2006
by

Christoph Lameter, Ph.D.
christoph@lameter.com

© 2006 Silicon Graphics, Inc.

Memory seems to be so simple. If you need it then just get some and use it. On most systems, one memory page is as good as another. On a NUMA system however the distance of the memory to the executing process matters. Performance can sink dramatically if memory references are made too frequently to pages on remote nodes. Local memory is special to the execution context because it has minimal latency and optimal bandwidth characteristics.

The kernel has the task to assign memory to a process in the most optimal way so that the process can execute with the highest performance. For that purpose, the kernel has been equipped with various mechanisms that determine node locality and insure that memory is allocated in such a way that the NUMA distances are minimized. During the execution of a process the locality constraints may change due to a variety of influences (scheduler, administrator, memory allocation). The kernel therefore also needs the ability to move pages while a process is executing. This is called page migration.

Memory allocation can also be controlled by the process itself through the establishment of memory policies. This is in particular useful if it is known that processes will use more memory than one node has available. It allows the distribution of memory to optimize the performance of the application.

Table of Contents

TRADEMARKS AND ATTRIBUTIONS

Silicon Graphics and IRIX are registered trademarks of Silicon Graphics, Inc., in the United States and/or other countries worldwide. Intel and Itanium are trademarks or registered trademarks of Intel Corporation. Linux is a registered trademark of Linus Torvalds, used with permission by Silicon Graphics, Inc. Red Hat and all Red Hat-based trademarks are trademarks or registered trademarks of Red Hat, Inc. in the United States and other countries. Novell is a registered trademark, and SUSE is a trademark of Novell, Inc., in the United States and other countries. All other trademarks mentioned herein are the property of their respective owners.

1. Introduction

NUMA (Non Uniform Memory Access) technology is relatively new to Linux and currently only a few Linux architectures support NUMA functionality. However, NUMA technology is gaining more significance as systems increase the number of processors since it becomes increasingly difficult to build a large system with uniform memory access speed. The distinction of a NUMA system from a regular system with multiple processors is that some memory is local to a processor—and therefore accessible with the least latency and highest throughput—while other memory is remote and therefore performance-wise inferior to local memory since the latencies are larger and the possible throughput is less.

NUMA technology first became available in commercial Linux distributions in 2004 (SLES9 and RHEL 4). Compared to the long history of Linux and Unix, NUMA memory management is of a fairly recent origin and—while it has been available before in other operating systems such as IRIX and Tru64—it is relatively new to the Linux kernel. Within the 2.6.X kernels we have an increasing refinement of the NUMA functionality as the kernel adapts to larger memory sizes and a higher number of processor counts. Recently local reclaim and memory migration support was added to the Linux kernel. These two topics will be discussed in detail here.

In this text we will first have a look at a simple toy NUMA system (Chapter 2) and discuss some of the basic concepts of NUMA. Then we give a simple overview on how memory management works without NUMA (Chapter 2) and then discuss how Linux memory management was changed through the introduction of NUMA functionality (Chapter 3). The problems encountered providing efficient allocations follow (Chapter 4) and then we discuss the recent introduction of *local memory reclaim* (Chapter 5) and *memory migration* (Chapter 6).

Finally we discuss shortly how a Linux process can manually control memory allocation in a NUMA system. The methods used are first *memory policies* (Chapter 7) and then *cpusets* (Chapter 8). The conclusion gives a brief look at how we may continue to develop NUMA support under Linux in the future.

2. A sample NUMA system

A NUMA system is comparable in many ways to common Symmetric Multi-Processor (SMP) and single processor (UP) systems. Processors execute instructions and process data that resides in memory.

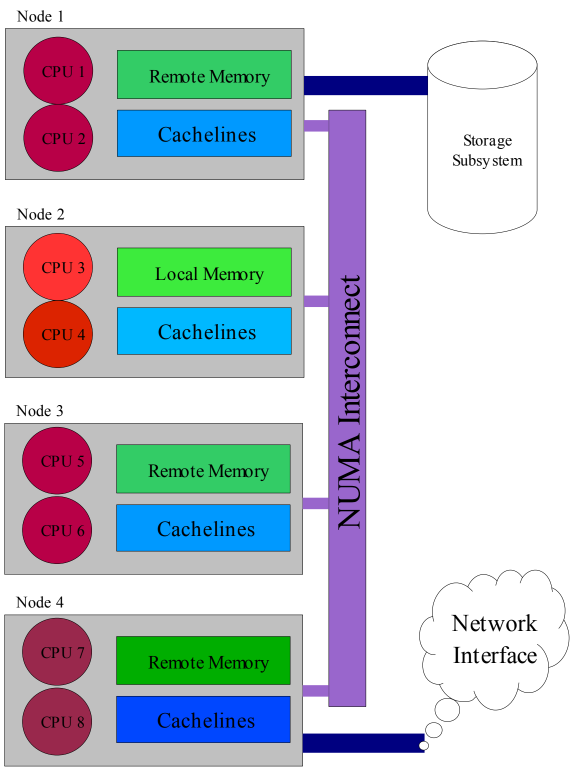
In SMP and UP systems memory is *uniform* meaning that the cost of accessing all memory locations is the same.[1] In NUMA systems the cost of accessing memory varies by the *NUMA distance*.[2] This distance typically refers to the increased latency and lower bandwidth of memory that is *off-node*. Optimization of applications running on a NUMA system requires *placement* of data structures in such a way that the application can run with the optimal performance.

The NUMA distances are recognized by the kernel in units of distance define by the ACPI (Advanced

1 Unified Memory Architecture (UMA).
2 Non-Unified Memory Architecture (NUMA).

Configuration and Power Innterface) SLIT (System Locality Information Table) definition.[3] According to the ACPI standard the distance for a regular SMP style memory access and for local NUMA distances in the SLIT table normalized to a value of 10.

The diagram to the right shows a schema for a model of a NUMA system that will be used for the following discussion. The model NUMA system has four nodes (Grey) that are connected via a NUMA interconnect (Magenta). Every node in this toy system has two processors, local memory (Green) and a set of cache lines that may have been retrieved from any memory in the system (Blue).



Drawing 1: NUMA Model

Lets say a process is running on processor 4 on node 2. The NUMA distances measure the efficiency of accessing memory in the system. If we now access *local memory* then the NUMA distance will be 10. The process is running on node 2 and therefore memory placed on node 2 is local to us and has the distance of 10, which is the lowest possible NUMA distance.

If the process running on processor 4 needs to access memory from node 1 then the access request needs to be transferred across the NUMA interlink and a result needs to be moved back. This is an *off-node* access that is slower than a local access. The NUMA distance expresses how much slower the memory access is compared to a local access. 14 is a common distance to a neighboring node. 14 is relative to 10 for a local access. So the remote access in our example is 1.4 times slower than a local access. The NUMA distances vary according to the technology and the speed of the NUMA interlink.

Typically the distances to neighboring nodes are the same. So if we would try to access memory from node 3 we would have the same distance. However, node 4 is further away and likely has a longer distance. Node 1 and 4 also have devices attached to them. Node 1 has a storage device attached. That means that I/O interrupts for retrieving data from secondary storage will typically be handled by that node. Node 4 has an attached network interface. These hardware devices give these two nodes particular advantages. Applications running on node 1 do not have to go via the NUMA interlink to write or read data from storage. Applications running on node 4 can access the network without requiring NUMA interlink traffic to occur.

The challenge for the operating system is to place the processing for applications and the memory used for processing in such a way that the overall performance of the system is optimized. In the rest of this text we will discuss how the Linux Operating System manages its memory in a NUMA system and

3 *ACPI Advanced Configuration & Power Interface*, http://www.acpi.info. Hewlett Packard, *ACPI System Locality Information Table Interface* (June 26, 2000), http://h21007.www2.hp.com/dspp/files/unprotected/Itanium/slit.pdf. Or Section 5.2.16 “System Locality Distance Information Talbe (SLIT)” in *Advanced Configuration and Power Interface Specificication.* Revision 3.0, September 2, 2004.http://www.acpi.info/DOWNLOADS/ACPIspec30.pdf.

how to use the operating system facility to place memory.

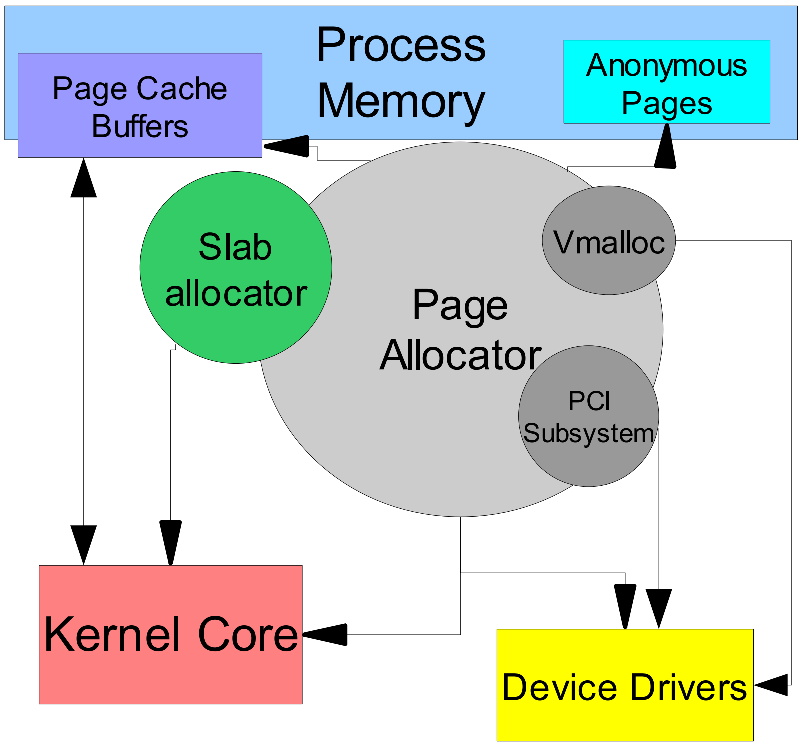
3. Linux and NUMA memory

3.1. Memory Management 101

The drawing on the right attempts to give a conceptual overview on how basic Linux memory management works. Central to all memory management is the *page allocator*. The page allocator can hand out pieces of memory in chunks of *page size* bytes.

The page size is fixed in hardware at 4 KBytes for i386, x64 and many other architectures. The page size is configurable on several platforms. On Itanium the page size is usually configured to be 16k.

All other memory allocators are in one way or another based on the page allocator and take pages out of the pool of pages managed by the page allocator. The page allocator may provide pages that are mapped into a processes virtual address space. There are two ways that pages mapped into user space are used.



Drawing 2: Linux Memory Subsystems

The first type of pages is used for *anonymous memory*. These are pages for temporary use while a process is running. They are not associated with any file and are typically used for variables, the heap and the stack. Anonymous memory is light weight and can be manged in a more efficient way than file backed pages because no mappings to disk (which require serialization to access) have to be maintained. Anonymous memory is private to a process (hence the name) and will be freed when a process terminates. Anonymous memory may be temporarily moved to disk (swapping) if memory becomes very tight. However, at that point an anonymous page acquires a reference to swap space and therefore a mapping to secondary storage, which adds overhead to the future processing of this page.

The *page cache* or *buffers* are pages that have an associated page on a secondary storage medium such as a disk. Page cache pages can be removed if memory becomes tight because their content can be restored by reading the page from disk. Most important is that the page cache contains the executable code for a process. A process may map additional files into its address space via the **mmap()** system call.

Both the executable code and the mapped files are directly addressable via virtual addresses from the user process. The operating system may also maintain buffers in the page cache that are not mapped into the address space of a process. This frequently occurs if a process manipulates files through system calls like **sys_write()**, **sys_read()** that read and modify the contents of files. The unmapped pages may be also thought as belonging in some loose form to a process. However, all page cache pages may be mapped or accessed by multiple processes and therefore the ownership of these pages cannot be clearly established.

The *kernel core* itself may need pages in order to store meta data. For example file systems may use buffers to track the location of sections of a file on disk, pages may be used to establish the virtual to physical address mappings (page tables) and so on. The kernel also needs to allocate memory for structures of varying sizes that are not in units of the page size in use on the system. For that purpose the *slab allocator* is used. The slab allocator retrieves individual pages or contiguous ranges of pages from the page allocator but then uses its own control structures to be able to hand out memory chunks of varying sizes as requested by the kernel or drivers. The slab allocator employs a variety of caching techniques that result in high allocation performance for small objects. Slab allocations are used to build up structures that maintain the current system state. This includes information about open files, recently used filenames and a variety of other state objects.

The *device drivers* utilize both the page allocator and the slab allocator to allocate memory to manage devices. There are a couple of additional variations on page sized allocations for device drivers. First there is the *vmalloc* subsystem. Vmalloc allows the allocation of larger chunks of memory that appear to be virtually contiguous within kernel context but the actual pages constituting this allocation may not be physically contiguous. Therefore vmalloc can generate a virtually contiguous memory for large chunks of memory even if the page allocator cannot satisfy request for large contiguous chunks of memory anymore because memory has become fragmented. Accesses to memory obtained via the vmalloc allocator must use a page table to translate the virtual addresses to physical addresses and may be not as efficient as using a direct physical address as handed out from the page allocator. Vmalloc memory may not be mapped into user space.

Finally, the PCI subsystem itself may can be used by a device driver to request memory that is suitable for DMA transfers for a given device via **dma_alloc_coherent()**. The way of obtaining that type of memory varies with the type of underlying hardware and therefore the allocation technique varies for each platform supported by Linux.

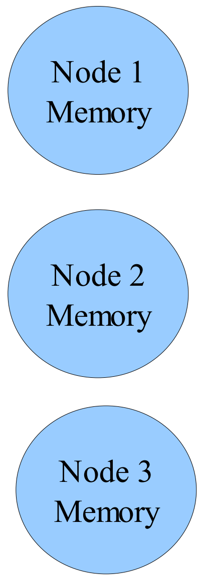
Here is an overview of the basic memory allocators under Linux:

Allocation Function	***Allocator***	***Action***
alloc_pages(flags, order)	Page allocator	Allocates 2^order contiguous pages
kmalloc(size, flags)	Slab allocator	Allocate *size* bytes of memory
kmem_cache_alloc(cache, flags)	Slab allocator	Allocate an entry for the indicated slab
vmalloc(size)	vmalloc subsystem	Allocate a virtual contiguous memory area with a minimum of *size* bytes.
dma_alloc_coherent(device, size, &addr, flags)	PCI subsystem/ architecture specific support	Allocate DMA capable memory for the indicated device.

3.2. NUMA memory

On a NUMA system memory is managed separately for each node. Essentially the page pools now exist for each node. Each node has an associated *swapper thread* that takes care of memory reclaim on a node. Each memory address range in a node is called an *allocation zone*.

The zone structure in turn contains free lists that hold available pages. Active and inactive lists are used to manage page reclaim. Pages move between the active and the inactive list to determine which pages can be reclaimed. If an inactive page is accessed then the page is moved to the active list. The swapper moves pages in the opposite direction from the active to the inactive list. If memory pressure leads to the necessity to reclaim pages then pages are freed from the end of the inactive list.



Drawing 3: Memory nodes

If an allocation request is made on a NUMA system then we will have to decide from which pool on which node we will allocate memory. We preserve the behavior of the existing allocators that were mentioned above (in the absence of memory policies or cpuset restrictions) and default to taking memory from the node on which the process is running. This is called *node-local* allocation and the intent here is to allocate memory that has the fastest access for the currently running process. It is highly likely that the memory allocated will be used soon by the task that allocated the memory, so the placing the memory on the local node is useful to minimize the access latency.

However, the kernel or device drivers may know for some allocation that the memory will be most frequently accessed from another node or it is known that the object will be accessed randomly from all nodes and we would like to spread the objects out to avoid overload situations. It is then advantageous to specify for each allocation from which node the memory should come. This required the adding of extra function calls with a **_node** suffix and led to the introduction of additional special allocators for NUMA systems:

Allocation Function	***Default action in a NUMA system***
alloc_pages(flags, order)	Allocate node-local pages
alloc_pages_node(node, flags, order)	Allocate pages from the indicated node
kmalloc(size, flags)	Allocate node-local memory
kmem_cache_alloc(cache, flags)	Allocate node-local memory for the indicated slab
kmalloc_node(size, flags, node)	Allocate chunk from indicated node
kmem_cache_alloc_node(size, flags, node)	Allocate chunk from indicated node for the slab
vmalloc(size)	Allocate node-local memory.
vmalloc_node(size, node)	Allocate pages from indicated node
dma_alloc_coherent(device, size, &addr, flags)	Allocate DMA capable memory from the node where the indicated device is locating.

4. Efficient allocations

Memory allocation in a NUMA system leads to a couple of new considerations in comparison with a system with uniform memory access speed. The existence of nodes effectively means that the system exists in multiple pieces that are more or less autonomous. Processes run best if they can stay within one of these containers and special consideration needs to be given to applications that span over multiple nodes in order to get best performance.

4.1. Optimal placement: Node local

The optimal placement of memory for a process is through *node local* allocations. In that case the memory access has the smallest latency possible and traffic on the NUMA interlink (a precious centralized resource) does not have to occur at all. However, that is only the optimal placement if all accesses continue to come from the current processor and if sufficient memory is available locally. There may be nodes with processors but with no memory and therefore no means of local allocation, so in some cases we may have to resort to off node allocation by default. Optimal placement can only be preserved if the process stays on the node on which the allocation occurred. Processes may be moved to other nodes by various means such as the scheduler. The scheduler can harm performance significantly by moving an execution thread to a different node. Memory accesses that were local before may now be remote and utilize NUMA interlink resources.

There are various means that the user has available to control memory allocation and process placement. Changing processor affinity of a running process has the same effect as the scheduler moving a process. The application may override the node local placement via memory policies. A cpuset may be configured to spread memory. All of these make the optimal placement impossible but these measures may be necessary for various other reasons that we will discuss later.

Small applications that only need to utilize the resources of a single node may be run with optimal speed by allocating all elements of the process node local. These processes can utilize no more memory than available on a single node and can have no more simultaneously executing threads than processors available for one node. Unix like operating systems—like Linux—typically run large amounts of these small processes. These may run unmodified and the system is typically able to place them optimally. So small processes may get near to the maximum performance possible in a NUMA system.

4.2. Multi node applications

The situation becomes more complicated if an application needs more memory or more processors than are available on one node. Since it is no longer possible to allocate all memory on the local node some performance trade offs need to be made. If there are no other memory restrictions then the best placements frequently depend on the way memory is used by the application:

Type	***Best Allocation***
Thread specific data	Node local since the data is only accessed from a local processor.
Shared read/write	If the shared memory areas are accessed with an equal likelihood from all nodes then it is best to distribute the data evenly across all the nodes. Typically *interleave* memory policies or *memory spreading* are used to to accomplish this.
Shared read/only	If the memory is frequently read but not written to then it is advantageous for each node to have its own copy. Data can be *replicated*. However, replicated data cannot easily be modified. Linux does not support data replication but uses interleaving or memory spreading for all shared data. The application would have to replicate the data on its own by generating a thread local copy of the shared data structure. Some means needs to exist to update the copy as needed.

Multi node applications may only have a few executing threads that fit on a couple of nodes but those threads could require more memory than that set of nodes provides. In such a case more memory may be allocated from neighboring nodes. However, if the more distant NUMA memory is used then the performance of the processes that use the memory is reduced. So an allocation strategy would need to strive to allocate that extra memory as closely as possible to the threads of the application.

All of these methods only apply for pages in direct use by the multi node application. Additional complexity is introduced through the location of pages in the page cache, locality of state information of the kernel about application's use of resources, etc. Typically the nodes an application runs on are not used exclusively by the application but also by the operating system.

Linux does use all available memory for the page cache which introduces some NUMA allocation problems that are the result of the way NUMA functionality was introduced into the kernel.

4.3. Page Cache

Linux caches pages (or blocks) from disk in memory and these pages are then said to be part of the *page cache*. However, the pages are not only used for caching. Pages can be mapped into the memory of a process as executable pages and as memory mapped pages. Page cache pages mapped into a process are not freed upon termination of the process in contrast to anonymous pages but will linger in memory in the hope that a future process will use the same page from disk.

This means that the operating system must provide a way to reclaim unmapped node memory to avoid filling up nodes with unmapped page cache pages.

Page cache allocations—like other data of a process—are best placed node local if referenced from a single processor. If a page cache page is referenced by multiple processors then the same considerations as for regular data also apply to the page cache page. The frequency of access plays an important role for freeing page cache pages. If the page will be reused soon then it is advantageous to keep the page in memory to avoid a disk access.

4.4. Memory balancing

In an optimal configuration for a multi-node application the memory on nodes local to the executing threads is distributed in such a way in that the NUMA interlink traffic is evenly spread across all memory nodes. The balancing avoids overloading one node with requests via the NUMA interlink.

If the threads are likely to execute arbitrary sections of the code and access arbitrary data then it is advantageous to spread the code pages, operating system state information and shared memory pages evenly over the nodes. Only special *thread local state* needs to be kept local to the executing processor.

In order to maintain best performance it may be advisable to monitor the access pattern of the application to the data structure it uses. The application may be optimized by rebalancing memory over available nodes if a memory area becomes a bottleneck or memory on a node short on memory is rarely used.

4.5. NUMA interlink and node hardware limits

The resources available through the NUMA interlink are typically limited. A NUMA node can only cache a limited number of cache lines from remote nodes and is only able to process a given quantity of requests per second for cache lines from other nodes. The NUMA interlink itself has a limited bandwidth. Usage of the NUMA interlink not only increases memory latency but may also delay other requests. In some cases contention on the NUMA interlink may cause severe degradation of application performance.

For example if a process starts on one node and faults in a large number of executable pages then all those pages are located optimally relative to that starting process. If the process then begins to start a large number of new threads on other nodes using the same executable then lots of remote requests for cache lines for the executable pages that were faulted in first are generated. The initial node may become a bottleneck because all those requests cannot be serviced fast enough. This may then slow down the performance of the application.

The *number of remote cache lines* kept in the local cache of a node is limited. If the cache footprint of the currently executing threads exceeds the available cache space then cache lines will have to be continually refetched, increasing NUMA interlink contention and draining the resources of the remote node. The *overall capacity* of the NUMA interlink may become a limitation if multiple applications have to mostly request cache lines from remote nodes because the most of the pages are not local to the executing processes.

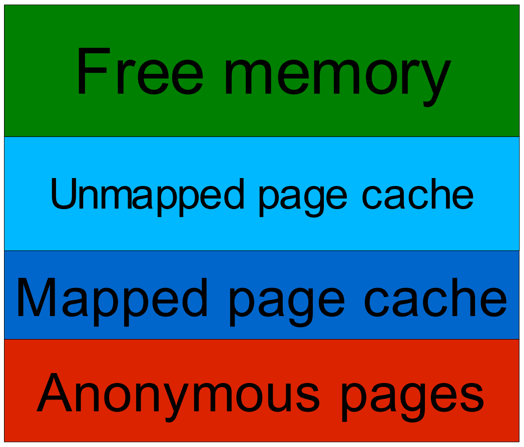
The NUMA interlink may be especially put under load by concurrent attempts to write to cache lines. Writes require exclusive access to cache lines and cause cache lines to be invalidated. Repeated reacquisition of the same cache line resulting in a bouncing cache line.

5. Memory reclaim

5.1. Reclaim in an UP or SMP system

Pages that are mapped by a process are page cache pages and anonymous pages. Anonymous pages are freed when a process terminates and mapped page cache pages become unmapped page cache pages. So over time more and more memory is used for unmapped page cache pages and the amount of free memory is gradually reduced as processes are created and terminated.

Finally, free memory will be lower than operating system limits and the swapper will begin to reclaim memory. If memory pressure is low (and the swapper is not run too frequently) then only unmapped pages from the inactive list are freed. This will usually lead to an increase in free memory and the cycle of accumulating unmapped page cache pages can start over again.



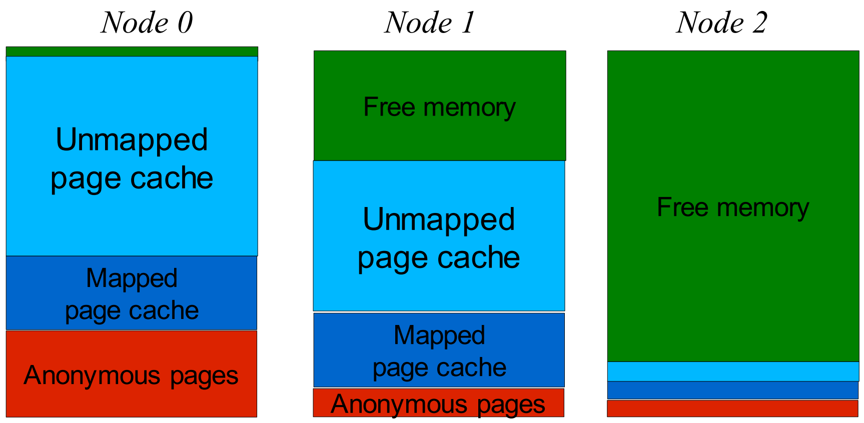
Drawing 4: Normal page reclaim

If memory pressure increases then swapping will become more aggressive. The swapper may start to unmap pages of a process in order to be able to free them or may map anonymous pages to swap space so that it will become possible to write back anonymous memory to swap space allowing more memory to be freed.

5.2. NUMA reclaim

In a NUMA system global reclaim happens infrequently since memory on all nodes must be exhausted to trigger the swapper. Reclaim is tied to a global shortage of memory and not a local shortage on a node.

Lets say we only run processes on node 0 and never on the other two nodes. Then the unmapped page cache will gradually consume all memory on node 0.



Drawing 5: Reclaim in a NUMA system

When the memory on node 0 is exhausted then the standard reclaim (swapper) will *not* run because there is still a large amount of memory free on the two other nodes. So globally memory is available while it is exhausted locally.

When new processes start on node 0 then pages from the node 1 may now have to be used for the executable pages and for the anonymous pages of the process. This means that the process cannot run with optimal performance anymore since most memory accesses will be off node accesses. Gradually the memory of node 1 and node 2 will fill up with unmapped page cache pages until all memory is exhausted. At that point the system will then free a small portion of memory. Having some memory free will then allow a short time of local allocations for the processes running on node 0. Global reclaim leads to allocations that are not optimal for processes in a NUMA system since local memory is not intentionally freed.

The more nodes exist the less frequent the triggering of global reclaim becomes because the larger the system the more difficult it becomes to drain all nodes of memory. With more nodes come longer paths to remote nodes and therefore longer NUMA latencies. Therefore the larger the system, the worse the penalty for processes running on those systems.

One technique that was used in the past in this situation was simply to remove all unmapped page cache pages from all nodes.[4] However, that could penalize processes running on other nodes and could require disk reads to retrieve frequently used unmapped pages that were just removed from memory.

In Linux 2.6.16 a form of *local reclaim* was introduced. *Zone reclaim* begins to do light reclaim (meaning removing unmapped page cache pages) if the memory in a zone is getting low. If zone reclaim is enabled then the system will notice that node local allocation cannot be satisfied because of a memory shortage in a local zone. It will then first attempt local reclaim of unmapped pages before allowing the allocation from off node memory

Zone reclaim can be controlled through the following **/proc/sys/vm** settings:

Setting	***Purpose***
/proc/sys/vm/zone_reclaim_mode	Switches zone reclaim on and off and allows the setting of special parameters. *zone_reclaim_mode* is set to 0 (off) for systems with insignificant NUMA distances between nodes. It is set to 1 if NUMA distances between nodes are 1.5 times slower than node local accesses.
/proc/sys/vm/zone_reclaim_interval	If we fail to reclaim enough memory to avoid going off node then avoid scanning for more reclaimable memory for this time interval. The default interval is 30 seconds. Reducing the interval increases the overhead of page reclaim but may reduce the number of off node allocations.

Zone reclaim frees unmapped pages from the list of inactive pages while allocations occur. There may be a stream of continual reclaims as long as the system is able use zone reclaim to satisfy local allocations. However, if the system finds that reclaiming memory is not enough to satisfy local allocations then the continual reclaim stops for short time periods (30 seconds by default). Local reclaim requires scanning of the inactive list for unmapped pages. It would lead to significant overhead if frequent unsuccessful scans for unmapped pages were to be performed.

4 SLES9 has a tool called **bcfree** for that purpose.

If a node is filled up then the timeout period determines the interval in which reclaim attempts are performed in order to find local reclaimable memory. The timeout period defines the overhead that we incur through scanning during off node allocation periods. A larger timeout period can be used to reduce the frequency of the checks and a smaller period allows more reclaim in situations were memory is rapidly allocated and deallocated.

Zone reclaim may operate in two advanced modes. First, one may want to *throttle write out processes*. If an application is rapidly writing to disk then dirty pages in the page cache will rapidly overflow memory in all nodes of the system because the I/O subsystem will not be able to write these pages out fast enough. Unmapped dirty pages are cannot be freed by regular zone reclaim because the information still has to be written back to disk. If bit 1 is set in **zone_reclaim_mode** then writes will occur during zone reclaim. The process that is allocating memory will have to wait until writes have completed leading to an effective throttling until write out is complete for a sufficient number of pages. The write mode stops rapidly writing processes from utilizing off node memory. This may increase the speed of the application writing to memory but it will typically cause a process to appear to run slower since the throttling slows the process down and it can no longer terminate with most of memory dirty.

Zone reclaim can also be made to behave like *regular swap* by setting bit 2 in **zone_reclaim_mode**. Zone reclaim will then perform regular swap activities if memory is short and evict pages to swap until local allocations can continue. Zone reclaim with swap will limit default page allocation to the local node unless memory policies or cpuset settings are used to force off node allocation. Excessive memory use leads to swapping and not to allocations on other nodes.

6. Memory Migration

6.1. Why migrate pages

The need for page migration arises from the desire to move memory between nodes in order to reach better locality for the pages that are in use by processes. If multiple processes are running on a set of nodes then the termination of one process will lead to a memory imbalance that may require the moving of some memory from node to node to restore an optimal memory distribution and to minimize NUMA latencies. If the scheduler decides to move a process to another node then accesses to many of the pages of the moved process may become off node accesses. Migrating the pages of the process to the new node may restore the performance of the process to what it was before the scheduler move. However, it should be noted that page migration is resource intensive. It is most valuable for long running processes.

If a system contains asymmetric nodes—like for example nodes with processors of different speeds or different amounts of memory—then page migration allows the reconfiguration of a running job to a new set of nodes with different performance parameters. Memory of an application may be compacted by moving the pages of a process onto fewer nodes or expanded to multiple nodes by spreading them out.

Page migration interfaces were designed to address various needs. Some were developed for the use of the system administrator, another for the application programmer, others for the use of NUMA batch schedulers and another for tools that optimize the performance of individual applications by tracing their inter node allocation patterns.

6.2. How page migration works

For page migration to occur, all the references to a page have to be tracked down and then all references have to be changed to point to the new page. Migration occurs by removing one type of reference after another, then copying the page to the new location and then reestablishing the references. If we cannot remove all references then the migration attempt for a page is aborted and the references to the old page are restored. In that case we may have to retry later. Page migration may not be able to migrate a page at all if the removal process for references keeps failing. That may occur for example for pages frequently referenced by multiple processes. References will be reestablished through accesses before page migration can remove all of them.

Page migration removes references in the following sequence.[5]

1. The page to be migrated is removed from the LRU lists (active and inactive). The page is no longer subject to swapping or reclaim. The page usually stays off the LRU if page migration fails in order to retry page migration later.
2. All the page table entries pointing to the old page are removed and replaced with special entries that will cause the processes accessing the page to sleep until page migration is complete. If these cannot be successfully installed then the migration attempt is retried later.
3. We check the reference count of the page to make sure that no other kernel references exist to the page. No means exist to stop other kernel threads from using the page and no means exist to discover which kernel thread holds a reference to the page. If references exist that we cannot account for then the migration attempt has to be aborted. The migration can be retried later in the hope that other kernel threads will not be using the page anymore.
4. If the reference count shows that we have tracked down all references then new kernel references are established to the new page. The special page table entries are removed and the processes that were put to sleep are reawakened.

Page migration was introduced first as *swap migration* in Linux 2.6.16. A migration via swap means that the pages are pushed into swap. The process that brings the pages back into memory is then in effect giving these pages a new location. However, going via swap is a slow process since it requires that pages be written out into swap space and it only allows relocation of the pages according to the cpuset and memory allocation policies in effect at the time of the next touch.

The next step for page migration was to do *direct migration* (also 2.6.16). Direct migration still uses the swap entry to preserve the identification of a page (in case multiple processes have references to the same anonymous page) but no longer writes the page out to swap. Instead the page is directly copied to the target node. Direct page migration allows an accurate placement of the new page independent of memory policies and the cpuset configuration.

Finally *swapless migration* (that will first appear in 2.6.18) abandons the use of swap entries and no longer requires swap to be setup. Page table references are encoded in a special type of *migration entry* that takes the role of the swap entry. Swapless migration is able to preserve the write enable bit in the

5 Note that this is a very simplified version that avoids to mention a lot of the locking details.

page table and therefore able to avoid useless COW faults. Swapless migration also restores all references to their earlier state. Earlier migration approaches were not able to restore page table references to page cache pages and as a result migration appeared to shrink a process since the process lost its association with its page cache pages that then had to be reestablished through additional (minor) page faults.

A few of the supported page migration interfaces allow the specification of sets of source and target nodes. If so then the attempt is made to preserve the page placement relative to the set of nodes. Pages that are on the first node of the source node set will end up on the first node of the target node set. Pages that are on the second node of the source node set will end up on the second node of the target node set and so on. The intend here is to preserve the memory layout that a NUMA aware process may have generated through the use of memory policies. In this manner it is likely that the performance on the target nodes is comparable to the performance on the original nodes. In particular the memory spreading for shared memory structures is preserved.

6.3. Determining how a process uses memory

Memory migration introduces a new way to find out how the pages of a process are allocated over the nodes available. If one wants to migrate memory then it is first of all necessary to know where the system has allocated the pages. Linux already had a **maps** file in **/proc/<pid>/** to show the current memory segments of a process. However, the **maps** file does not display how many pages are in memory nor on which nodes these pages reside. So for page migration a new **/proc/<pid>/numa_maps** file was introduced.

The **numa_maps** file provides additional information for each memory segment of a process and shows how many pages are used on the available nodes in the system. The first field of **numa_maps** is the starting address of a memory segment. This address allows a correlation with other status information obtained from **maps** or **smaps.** The second field shows the memory policy currently in effect for a particular memory area (Chapter 7). The memory policy is important to understand how the system allocated memory in a particular segment of a process. The remainder of the lines contains additional information on pages in the memory area:

Information item	***Description***
N<node>=<pages>	The number of pages used on a particular node. The number of pages counts pages that are mapped into this particular memory area. Note that multiple processes may map the same page!
file=<filename>	The file backing this memory area. If the mapping is read-only then additional anonymous pages may have been generated for this area through COW (copy-on-write).
heap	This memory area is used for heap storage.
stack	This memory area is used for the stack.
huge	This memory area is used for huge pages. The number of pages displayed refers to page of huge page size and not to regular sized pages.
anon=<pages>	Number of anonymous pages.

Information item	***Description***
dirty=<pages>	Number of dirty pages.
mapped=<pages>	Number of mapped pages. Only shown if different from the count of anonymous or dirty pages.
mapmax=<count>	Maximum number of processes mapping any page in this memory area. The maximum mapcount may be useful as an indicator as to how much sharing is occurring for a particular memory area.
swapcache=<pages>	The number of anonymous pages that have an associated entry on the swap device but that have not been paged out.
active=<pages>	Number of pages on the active list. Only shown if different from the number of pages in the memory area. If this items shows then inactive pages exist that may be subject to removal by the swapper if memory pressure builds up.
writeback=<pages>	Number of pages queued for write out to disk.

The numbers of pages displayed are obtained by scanning over all pages in a specific memory area. Pages may be accounted for multiple times. For example a page may be an anonymous page, dirty and be placed on a certain node. Zero pages are not accounted for at all since they all use the same pages with only zeros. Multiple processes may share the same address space if the processes were started with the **CLONE_VM** parameter to the clone() system call (we often call them “threads”). Such a group of processes counts as one reference count for **mapmax** and all threads will show the same **numa_maps** output.

An example of **numa_maps** output for the **cron** process is given above. This was a system with 4 nodes. One can see that the file **/var/run/nscd/passwd** contains only a single page that is mapped by a large number of processes. The various pages in anonymous memory areas are mainly allocated on node 3.

Text 1: numa_maps output for cron running on a system with 4 nodes

```
00000000 default
2000000000000000 default anon=2 dirty=2 N3=2
2000000000200000 default file=/var/run/nscd/passwd mapped=1 mapmax=92 N3=1
2000000800000000 default file=/usr/sbin/cron mapped=5 N3=5
2000000800020000 default file=/usr/sbin/cron anon=1 dirty=1 N3=1
2000000800024000 default file=/lib/ld-2.4.so mapped=7 mapmax=147 N0=7
2000000800068000 default file=/lib/ld-2.4.so anon=2 dirty=2 N1=1 N3=1
2000000800080000 default file=/lib/libpam.so.0.81.2 mapped=1 mapmax=59 N0=1
20000008000a4000 default file=/lib/libpam.so.0.81.2 anon=1 dirty=1 N1=1
20000008000a8000 default anon=1 dirty=1 N3=1
20000008000ac000 default file=/lib/libpam_misc.so.0.81.2 mapped=1 mapmax=16 N0=1
20000008000bc000 default file=/lib/libpam_misc.so.0.81.2 anon=1 dirty=1 N1=1
20000008000c0000 default file=/lib/libc-2.4.so mapped=58 mapmax=147 N0=58
2000000800304000 default file=/lib/libc-2.4.so anon=2 dirty=2 N1=1 N3=1
200000080030c000 default anon=1 dirty=1 N3=1
2000000800310000 default file=/lib/libdl-2.4.so mapped=1 mapmax=112 N0=1
2000000800324000 default file=/lib/libdl-2.4.so anon=1 dirty=1 N1=1
2000000800328000 default anon=5 dirty=5 N1=5
603ffffffffc000 default anon=1 dirty=1 N1=1
607fffffff4d8000 default stack anon=1 dirty=1 N3=1
```

6.4. Cpuset controlled memory migration

The simplest way to migrate an applications pages is to put all processes for that application into the same cpuset. Cpusets have an additional control file in the directory of each cpuset called **memory_migrate.** If a 1 is written to **memory_migrate** then cpusets will take care of page migration whenever the memory nodes of a cpuset or a task in a cpuset are changed. Changing the nodes listed in **mems** will migrate all processes to the new set of nodes. Moving a single task by echoing the process id to the **tasks** file will move the pages of the process to conform to the node restrictions of that cpuset.

The container nature of cpusets makes it easy to control page migration for a group of processes or for a single processes that move between cpusets. Cpuset migration is particularly useful if the management of the nodes is external or manually done and if the processes do not manage their own NUMA locality information.

6.5. Policy based page migration via mbind()

Processes can set up memory allocation policies for all or part of their own memory. Memory policies for address ranges are specified using the **mbind()** system call which may then specify a memory allocation strategy or restrict allocations to a particular set of nodes. The **mbind()** call typically only returns an indication that pages already exist in a range that do not satisfy the allocation requirements.

A flag **MPOL_MF_MOVE** was added to **mbind()** to enable the moving of pages in order force the pages to conform to a restricted set of nodes. The additional flag enables an application to migrate all pages in a certain address range to a specific set of nodes increasing the control that an application has over its memory.

However, usage of the policy based page migration method requires the application to control its own allocations. This migration method is only useful for those writing or modifying programs to run in an optimal way on a NUMA system using **libnuma** in the **numactl** package.

6.6. The numa_migrate_pages() system call

numa_migrate_pages() is a new system call supported by **libnuma** in the **numactl** package. The prototype is:

int numa_migrate_pages(pid_t pid, const nodemask_t *from, const nodemask_t *to)

numa_migrate_pages() migrates all pages of the specified process that placed on the **from** nodes to the nodes listed in **to.** It returns the number of pages that were not moved or an error code on failure. The function call is mostly useful for batch schedulers that want to move the pages of a process.

If **numa_migrate_pages()** is called in the context of a regular user account then only the pages of processes that also belong to the same user are moved. Moreover pages are only moved if they are only mapped by a single process in order to avoid moving pages of other processes. If a user with administrative privileges calls **numa_migrate_pages()** then all pages of the process are moved regardless of other processes mappings. Typically this means that pages in shared libraries are included in memory migration.

Possible return values of **numa_migrate_pages*:***

Errno	***Reason***
0..n	Number of pages that were not migrated.
-EFAULT	Unable to read node lists.
-ESRCH	Target process does not exist
-EPERM	Access to the target nodes not allowed in this cpuset. Or a user trying to migrate a process belonging to another user.
-ENOMEM	Unable to allocate pages on the target node

6.7. The migratepages command line tool

The **migratepages** command line tool is based on the **numa_migrate_pages** system call and allows the migration of a processes pages from the command line. The tool is mostly useful for system administrators that want to move the pages of a certain process to other nodes. The syntax of the **migratepages** command line tool is:

```
migratepages <pid> <sourcenodes> <targetnodes>
```

Node lists are parsed in the same way as other node lists in the **numactl** package. One can specify ranges of nodes or lists of nodes in both positions..

If a regular user uses **migratepages** then only pages that are only mapped by the specified process are mapped. If a user with administrative privileges uses **migratepages** then all pages of a process are moved.

6.8. Migrating individual pages using the move_pages() system call

A new system call is expected to become available in Linux 2.6.19 called **move_pages().** The new function call is used to be able to determine the state of individual pages and may be used to migrate individual pages. The idea is that a profiling tool has somehow determined a list of pages that are not optimally placed and that we now want to move that list of pages to various target nodes. **move_pages()** takes parameters of arrays of various kinds and then operates on a large set of pages. The prototype is:

int move_pages(pid_t pid, int nr_pages, const void **pages, const int **nodes, int **status, int flags)

The **pages** array contains references to the individual pages in the process to be moved. The **nodes** array contains the nodes that these pages have to be moved to. If a **NULL** is passed instead of a **nodes** array then no pages are moved but the status of each page is determined and returned in **status**.

The **status** array is used to return the state of pages after the move and is only valid if **move_pages()** did not return an error code itself. The possible states are:

Page status	***Meaning***
0..MAXNUMNODES	The pages has been moved to the indicated node or is located on the node.
-ENOENT	The page is not present or a zero page.
-EACCES	The page is mapped by multiple processes and can only be moved by an administrative user by specifying the **MPOL_MF_MOVE_ALL** flag.
-EPERM	The page is under **mlock** and cannot be moved.
-EBUSY	Page is busy and cannot be moved. The page was taken off the LRU by the swapper, another process migrating pages or by some other kernel thread.
-EFAULT	Invalid address (no address range for the page or address is zero)
-ENOMEM	Unable to allocate memory on the target node
-EIO	Unable to write back page. The page must be written back in order to move it since the page is dirty and the file system does not provide a migration function that would allow the moving of dirty pages.
-EINVAL	A dirty page cannot be moved since the file system does neither provide a migration function nor a means to write back the page.

The **flags** parameter indicates if pages shared between processes should be moved:

Flag	***Meaning***
MPOL_MF_MOVE	Move pages that are only referenced by the specified process.
MPOL_MF_MOVE_ALL	Also move pages that are referenced by multiple processes. This requires administrative privileges.

Upon completion **move_pages()** may return the following error conditions:

Error condition	***Meaning***
-ENOENT	No pages were found that require moving. All pages are either already on the target node, not present, had an invalid address or could not be moved because they were mapped by multiple processes.
-EINVAL	Flags other than MPOL_MF_MOVE(_ALL) specified or an attempt to migrate the pages of a kernel thread.
-EPERM	MPOL_MF_MOVE_ALL specified without sufficient privileges or an attempt to move a process belonging to another user.
-EACCES	One of the target nodes is not allowed by the current cpuset.
-ENODEV	One of the target nodes is not online.
-ESRCH	Specified process does not exist.
-E2BIG	Too many pages to move.
-ENOMEM	Not enough memory to allocate control array.
-EFAULT	Unable to access parameters.

7. Memory Policies

Memory policies are a means to for an application to control its own memory allocation. Memory policies are set using the **mbind(2)** and **set_mempolicy(2)** system calls. Memory policy support allows the specification of one of the following allocation strategies for the whole process or for an address ranges of the process:

Policy	***Effect***
MPOL_DEFAULT	Default policy. Usually this means that an allocation should happen from the current node or from a node that has memory available and is as close as possible to the process we are allocating from.
MPOL_PREFERRED	Allocations are to be done from a particular node. However, if memory is not available on that node then any other node will be fine.
MPOL_INTERLEAVE	Memory allocations will be spread out over all available nodes. This is mostly used for shared memory areas. Spreading insures that no node is overloaded with requests and also leads to the use of equal amounts of memory from each node.
MPOL_BIND	The allocation is restricted to a set of nodes. Allocations will fail if these nodes cannot provide the needed memory.

Memory policies may be used in two different ways which often leads to confusion about their role for memory allocation. On the one hand memory policies may be set up statically for memory ranges and for the whole processes. The policies will then stay the same for the duration of the process and additional memory allocated will be allocated following the policy either for the memory range or the whole tasks. The cpusets subsystem is able to do a dynamic change of these policies to fit new node constraints in another cpuset if a process is migrated after it has established its policies.

Memory policies are inherited during fork and therefore one process may set memory policies for others that it then forks off. The **numactl** is a tool that can be used to set policies for another process following this method.

Another use of memory policies is to dynamically switch the allocation policy for the various data items that need to be allocated. The memory allocated may then no longer follow the memory policy that is currently in place. If an application manages its own sets of nodes then the cpuset subsystem cannot reliably translate memory policies into a new context.

8. Cpusets

The cpuset functionality in the Linux kernels allows a segmentation of a machine with a large number of nodes into sets of nodes. Processes can then be run within these node sets with a degree of autonomy from the rest of the system. Memory allocations are restricted to the cpuset and therefore cpusets are frequently used to partition available memory in a system in order to run several large scale applications that may then be managed separately.

Cpusets provide a container mechanism for processes and provide a degree of control of memory allocation. Cpusets support *memory spreading.* If memory spreading is enabled then node local is no longer the default policy for memory allocations. Instead memory is obtained round robin from every node in the cpuset. This results in most accesses to become off node and so in some ways is a performance reduction. However, this reduction in performance may be the price one is willing to pay if the default allocation methods lead to an overload on one node that results in a performance bottleneck. Some applications may not be NUMA aware and without access to the source the system administrator does not have the capability to change the allocation patterns that cause these overloads. The big switch in the cpuset may be one way to address the issue. Other applications exist where each thread may access any memory in use by the application. In that case there will be no advantage to node local allocations and memory spreading becomes an optimal memory allocation strategy.

The memory spreading within a cpuset can be controlled for the page cache (**memory_spread_page)** and separately for the slab allocations that the kernel uses to keep state for the task (**memory_spread_slab**). By far the most important setting is for the page cache. Note though that anonymous pages are not subject to **memory_spread_page**. If a large array is to be shared between multiple tasks then the array has to be equipped with an interleave memory policy in order to spread the pages of the array.

The most important aspect of slab allocation is that the spreading affects the allocation of the data needed to keep track of open files and the caching of the directory tree. If slab allocation follows the node local strategy then the file information will be allocated local to the process. That strategy is optimal if single processes open, read and close files. If allocations are spread then applications benefit that access files from all threads of an application.

As mentioned above cpuset containers may be used to control page migration. If all the threads of a large applications are placed within a cpuset then the whole application can be conveniently migrated to other nodes.

9. Potential future work

Local reclaim and memory migration are a step forward in Linux NUMA memory management. The implementation of memory migration capabilities will hopefully be complete once the functionality that is described in this document is reached in 2.6.19. However there are still a number of other issues in NUMA memory management that are under discussion and that will need a resolution in the future. Here is a brief summary of each of these issues:

9.1. Conceptual overlap between cpusets and memory policies

Currently the functionality of memory policies and cpusets overlaps to some extend. Both can be used to redirect allocations and both can be used to restrict allocations. It may be conceptually cleaner (and simplify the implementation) if each of those would just focus on one area.

Memory policies are a means to control memory allocation. However, they are also used to restrict allocations to sets of nodes via the **MPOL_BIND** policy. This overlaps with cpuset functionality. It would be better if we had some way of attaching a memory area to a cpuset and have the cpuset be the sole container facility for nodes.

The spreading feature of the cpusets can be use to set an interleave policy for most allocation. This is going into the domain of the memory policies. Maybe it would be possible if cpusets could set a default memory policy for allocations on their node sets for different types of allocation in order leverage memory policies instead of implementing their own spreading.

9.2. Cpuset interaction with memory subsystem managing their own locality (f.e. Slab)

Cpusets may currently restrict allocations made by other subsystems to specific nodes. Some subsystem manage their own locality and are designed to work with memory from specific nodes. For example the slab allocator builds up per node arrays of pages by using node specific allocation of those pages. Depending on the cpuset context of the executing task, cpusets may redirect these allocations so that pages from foreign nodes may show up in the per node arrays. This is not in harmony with the design of the NUMA slab allocator.

9.3. Memory policy support for the page cache

Memory policies are only partially implemented for page cache memory areas. Currently memory management uses the policy of the task that is allocating a page cache page to guide the allocation of the page. However, that page may be used by multiple processes that may have conflicting memory policies. The only tie in here is to the file and there have been proposals to implement memory policies for each file. However, that would require file system extensions to place memory policy information into those. Other proposals have set a memory policy for the pagecache in general. Cpusets to some degree implement a kind of page cache policy with the **memory_spread_page** setting. However, this is another case of cpusets reaching into the area of memory policies.

There is currently no clear way forward on this and maybe the existing situation is as good as it can be. Multiple proposals for page cache policy and file based policies have been rejected. We are waiting for the genius with the bright idea to move us forward.

9.4. Page dirtying in shared writable mapping not accounted for

Linux currently has no way of accounting for pages that have been dirtied in shared writable mappings. These mappings are rare but they are used frequently for streaming applications in NUMA systems. The lack of dirty accounting means that the kernel is not aware of how many pages are dirty and is unable to trigger filesystem activity that would write back the dirty pages. The dirty pages are only recognized when a process terminates. If the process is running for weeks then it could take weeks until changes to files are written back.

Not triggering writeback can have more severe consequences when the kernel gets into a situation where memory is needed to start writeback but all of memory is dirty. In that case the operating system deadlocks because the kernel has not been able to track dirty pages.

There is currently work in progress by Peter Zijlstra, Hugh Dickins, Nick Piggin and me to accurately track the dirtying and Linus has indicated that this patch will be accepted for 2.6.18.[6] So we expect a resolution to this issue shortly.

9.5. Lock contention when reading from a file in a multithreaded process

If multiple threads of a process read from the same file then there is heavy lock contention on the lock for the mapping. This is a problem particular to the page cache pages. The locking situation for anonymous pages was improved in 2.6.13 with the splitting of the page table lock.[7]

Nick Piggin currently has work in progress called the *Lockless Page Cache.* The approach is to reduce the use of locks for the page cache. His current proposal eliminates the need to take the lock for reading the mapping[8] and benchmarks show encouraging results.[9] We are looking at ways to further distribute the lock for the write case. One possible solution may be to put a spin lock into the lowest layer of the radix tree to implement an approach similar to what was done for the page table lock. However, the radix tree also contains tags that need to be inherited across different tree levels which creates new challenges for scalability.

9.6. Better memory reclaim for SLABs

Currently reclaim for the icache and dentries is implemented on the individual object level by both subsystem. This means that a lot of either cache has to be freed in order to allow the recovery of a significant portion of pages from either either cache. We would like to be able to target reclaim to a

6 http://lwn.net/Articles/185463/
7 http://lwn.net/Articles/156356/
8 http://lwn.net/Articles/178434/
9 http://kerneltrap.org/node/6530

particular node and be able to free remaining objects from a pages if only a few remain in a slab page. If we would be able to do so then we could leave large areas of either slab cache untouched during reclaim which increases the chance of future icache and dcache hits which in turn would increase performance.

9.7. Manual vs. Automatic page migration

There have been recent proposal to put automatic page migration into the kernel. However, our past experience has been that automatic page migration schemes must use heuristics to predict future page references. If this prediction is wrong then page migration is not improving performance but wasting processing resources. It seems that no scheme has found that consistently improves performance on an NUMA system. It may be that a user space logic is the best way to approach automatic page migration for now.

9.8. NUMA scheduler with per node statistics.

The logic needed to effectively manage processor time and memory resources seems to be quite complex. and it is likely that the complexity and overhead necessary to implement an effective NUMA scheduler which would also be able to handle memory migration effectively is very difficult.

Once solution would be to move the complexity of the NUMA scheduler into user space. For that purpose we would have to equip the schedule with an interface that allows a user space monitoring and control of the location of processes. The memory subsystem needs to provide exact information of memory use on each node and for each process. The NUMA scheduler in user space could then process all this data and maybe build a data base of application behavior that would allow effective scheduling. Profiles may exist for known application that can help with special scheduling needs.

10. Conclusion

NUMA technology is still evolving in the Linux kernel and a series of significant decisions will have to be made soon. The new processors that emerge explore a variety of ways to realize multiple execution thread on a single chip which may require additional changes to be able to utilize these special features. For example the cell processor has an asymmetric processor design, the Intel products are symmetric and use a shared frontside bus and the AMD products come with multiple hypertransport links that naturally lead to a NUMA type design. Each may present its own challenges if large scale systems are build based on those designs.

11. References

Catanzaro, Ben. *Multiprocessor System Architectures: A Technical Survey of Multiprocessor/Multithreaded Systems using SPARC, Multilevel Bus Architectures and Solaris (SunOS)*, Prentice Hall, 1994.

Center for Computational Sciences. *Evaluation of the SGI Altix 3700 at Oak Ridge National Laboratory.* (Oak Ridge National Laboratory, TN), http://www.csm.ornl.gov/evaluation/RAM/PDF/ORNL-SGI-Altix-Evaluation-Report.pdf. Accessed January 25th, 2006.

Love, Robert. *Linux Kernel Development.* Sams Publishing, Indianapolis: Indiana, 2004.

Schimmel, Kurt. *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers.* Addison-Wesley, 1994.

Silberschatz, Abraham, Peter Baer Galvin and Greg Gagne, *Operating System Concepts.* Wiley & Sons, 2004.

Tannenbaum, Andrew S. *Modern Operating Systems,* Prentice Hall, 1992.