Reprinted from the

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

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

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Linux memory management on larger machines

Martin J. Bligh
mbligh@aracnet.com

David Hansen
haveblue@us.ibm.com

Abstract

A large amount of work has gone into the memory management subsystem during the 2.5 series of Linux® kernels, and it is more stable under a wide variety of workloads than the 2.4 VM (virtual memory subsystem). Many scalability problems have been solved, making memory managment perform much better on larger machines (meaning either with more than 1GB of RAM, or more than one processor, or both). Some of these changes also benefit smaller machines.

During the 2.4 series of kernels, the main Linux distributions diverged massively from the mainline kernel, particularly in the area of VM. This causes ongoing maintainance problems, and wasted duplicated effort in problem solving and feature implementation. Many of the enhancements made by the distributions have been brought back into the mainline kernel during the 2.5 series, under the leadership of Andrew Morton, providing a solid base for future development, and a greater potential for co-operative work.

This paper discusses the changes made to the Linux VM system during 2.5 that will significantly impact larger machines. It also covers changes that are proposed for the future, most of which are currently available as separate patches. Larger machines also have to cope with a larger number of simultaneous tasks—I have focused on up to 5000.

For the sake of simplicity, clarity, and brevity, we assume an IA32 machine with PAE mode (3 level pagetables) and normal memory layout settings throughout the paper. Unless otherwise specified, measurements were taken on a 16-CPU NUMA-Q® system (PIII/700MHz/2MB L2 cache) with 16GB of RAM.

1 Introduction

Market economics dictate the prevalance of large 32 bit systems, despite the software complexity involved. Though cheap 64 bit chips are beginning to appear, they are still not available as large systems. However, the techniques and discoveries described in this paper are by no means only applicable to such machines.

2 The global kernel virtual area

The fundamental problem with 32 bit machines is the lack of virtual address space for both user processes and the kernel—32 bits limits us to 4GB total. Each user processes' address space is local to that process, but the kernel address space is global. In order to ensure efficient operation, the user address space is shared with the global kernel address space (see Figure 1).

The default address space split for Linux 2.4 and 2.5 is 3GB user: 1GB kernel. It is possible to change this split, but it is often not desirable—some applications (such as

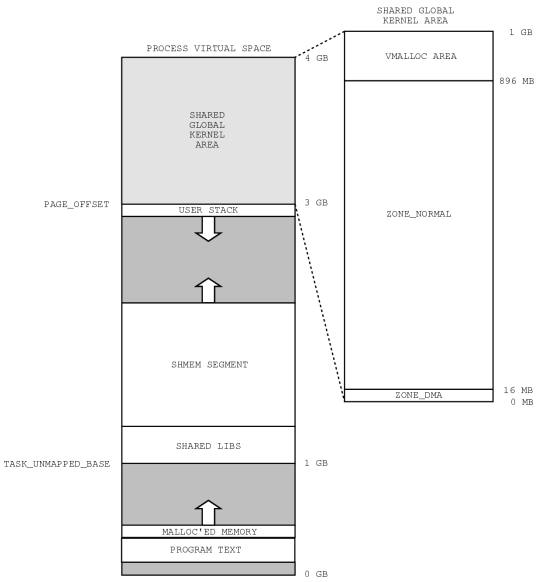

Figure 1: The process address space

databases) want as much address space for the process as possible for the application, whilst the kernel also wants as much space as possible for its data structures.

The first 896MB of physical memory is mapped 1:1 into the shared global kernel address space. This memory range is known as low memory (ZONE_NORMAL), and memory above the 896MB boundary is known as high memory (ZONE_HIGHMEM). The more physical memory we add to the machine, the bigger the kernel control structures need to be, but the control area is fixed size by the virtual space limitation.

Thus the more RAM we add to the machine, the more pressure there is on the global kernel area. The standard Linux 2.4 kernel copes very badly with large amounts of memory, perhaps limited to 4GB at best. The Linux 2.4 enterprise distributions will work with 16–32GB of memory, depending on the distribution. Linux 2.5 will cope with approximately 32GB of memory.

Unfortunately, most of the data that is put into the kernel address space is not swapable, and the Linux kernel often does not shrink the data gracefully under memory pressure. Thus, the failure condition is often difficult to diagnose; kswapd goes into a flat spin, all kernel memory allocations stop, and the system appears to have hung. Monitoring the “lowfree” field of /proc/meminfo in the runup to the system hang will often help to detect this condition.

The main space consumers for the kernel space are:

- mem_map (physical page control structures)
- Slab caches, particularly:
  - buffer_head
  - dentry_cache
  - inode_cache
- Pagetables

mem_map is an array of page control structures, one for each physical page of RAM on the system. On a 16GB machine, that takes 19% of the kernel’s address space. For 64GB, it takes 78% of all the space we have, leaving insufficient space for the normal kernel text and data. Whilst the machine may boot, it will not be usable.

William Irwin and Hugh Dickins are implementing a technology called “page clustering,” that makes one page control structure govern a group of pages, thus dramatically reducing the space taken (e.g. 8 page groups reduces us from 78% of space to 9%).

3 kmap

The kernel has permanent direct access to low memory, but needs to perform special opera-

tions to map high memory (however, note that user space can directly map high memory). High memory is usually mapped one 4K page at a time, via two main mechanisms: persistent kmap, and atomic kmap.

Persistent kmap uses a pool of 512 entries. All entries start out as clean; each entry is used in turn, and has a usage count associated with it. As entries are freed (usage count falls to 0) they are marked as dirty. When we reach the end of the pool, all dirty entries with 0 usage are marked as clean, a system-wide tlbflush is invoked, and now the buffers may be reused. All of these operations are global, and done under a global lock (kmap_lock).

Atomic kmap has a small number of entries per CPU—one for each of a few specific operations (that may need to be done in conjunction, so one entry is not sufficient). To reuse an atomic kmap slot, a single TLB entry needs to be flushed, and only on 1 CPU. This allows lockless operation, and CPU local data management (i.e., no cacheline bouncing). However, due to the CPU-local nature of the mapping, it is not possible to sleep, or reschedule onto another CPU whilst holding the mapping.

The problem comes in that persistent kmap turns out to be heavily used and rather slow. Not only is the data & locking global, but the global TLB flushes are very expensive (particular on machines without tlb_flush_range, such as IA32). Persistent kmap scales as $O(N^2)$ where N is the number of CPUs in the system (N times the frequency that the pool is exhausted * N times the impact from the tlb flushes). As CPU:memory speed ratios continue to grow, the caches become ever more important, and such algorithms are not suitable for heavy use.

The heaviest users were copy_to/from_user and related functions (copying data between kernel and userspace). It is possible that these operations would take a pagefault on the userspace page, and thus sleep (and thus cannot directly use atomic kmap). Other heavy users included one implementation of putting user pagetables into highmem—workloads with heavy pagetable manipulation (e.g. kernel compiles) were observed to spend more than half of their time just mapping and unmapping pte pages.

After much discussion on the subject during 2002, the following solution was agreed upon, and Andrew Morton implemented it. In essence, we now use atomic kmap for operations such as copy_to/from_user, but touch the page first to ensure it is faulted in, making it extremely unlikely that we will take a pagefault. In the unlikely event that a pagefault does occur, we handle the fault, then retry the copy operation using persistent kmap (in practice, this was never found to occur). Truly persistent global operations (typically for the lifetime of the OS instance) where performance it is not a concern can still use persistent kmap.

4 Pagetables

The pagetables map the process' virtual addresses to the physical addresses of the machine. For an IA32 machine with PAE, each PTE entry controlling a 4K page consumes 8 bytes of space, resulting in a fully populated 3GB process address space consuming 6MB of PTE entries. In other words, the overhead of PTEs is 0.2% of physical RAM if we have no sharing going on.

In most workloads, however, there is significant amounts of space shared between processes, either in shared libraries, or as shared memory segments. In particular, database workloads often use large shared segments (e.g. 2GB) shared between large numbers of processes. Whilst the memory itself is shared

between processes, the pagetables are duplicated; one copy for each process. Thus for 5000 processes sharing a 2GB shmem segment, the PTE overhead for that segment is now 20GB of RAM (i.e. the overhead is 1000% of the consumed space).

These levels of heavy sharing are for real workloads analysed, not a theoretical projection, and for a machine that might otherwise run happily with 8GB of RAM.There are two obvious ways to reduce the overhead: either we share the pagetables, or we reduce the size of each copy substantially.

Sharing the PTE level of the pagetables has been implemented by Dave McCracken. This enables us to share identical mappings over large areas between different processes, thereby reducing the mapping overhead for the case under consideration from 20GB to 4MB. It is totally transparent to applications, but is not currently in the 2.5 kernel as of 2.5.68.

The locking required for shared pagetables makes the patch slightly complex, and sharing can only occur on a pte-page sized basis (2MB of memory). Due to the mechanisms of shared libraries writing to certain areas of the shlib (and thus causing a copy-on-write split for the 2MB area) and the alignment requirements, it is not generally useful as a mechanism for reducing the overhead of shared libraries. It is, however, extremely effective on large shared memory segments, and reduces the overhead of fork+exec for large processes.

Support for large hardware pagesizes (aka hugetlbfs) can dramatically reduce the overhead of each process' pagetables. On IA32 PAE, there is only 1 large page size available (2MB), and this reduces the overhead by a factor of approximately 512. Memory consumption for the case under consideration is reduced for this case from 20GB to about 60MB. This is in the Linux 2.5 kernel, but requires small modifications to applications in order to use this facility.

A static pool of memory reserved for large pages is established at boot time, and handed out to applications that request it via a flag to shared memory create calls. Future work is planned to make a more flexible mechanism, whereby it is not necessary to reserve a static number of pages, and the kernel automatically uses large pages where appropriate.

In order to accomodate the large numbers of pagetables that are potentially needed on larger systems, it is possible to put the third level of the pagetables (PTEs) into the high memory area, rather than the main global kernel space. While this can greatly alleviate the space consumption problem, it comes at a price in terms of time.

Though modern implementations of highmem pagetables use atomic kmap, the cost of setting up the mappings for access, and the subsequent TLB flush is still expensive for such heavy usage, especially for workloads that create and destroy processes frequently. For kernel compiliation, the overhead of highpte was an increase of approximately 8% of system time.

5 UKVA

The shortage of virtual space on IA32 keeps the kernel from directly mapping everything that it might like to, especially things which are in high memory. We have mechanisms to do this temporarily with kmap() and kmap_atomic(), but both of these mechanisms impose significant overhead in data management and tlb flushing.

For workloads with large numbers of processes, one of the largest consumers of virtual space is page tables, specifically the bottom-level PTE pages. There is an option (high-

pte) in the kernel to put these pages in high memory and map them via kmap_atomic() as needed, but this incurs an 8% increase in system overhead. It would be much more efficient to permanently map the PTE pages, but into a per-process area instead of a global one, thus giving efficient operation without wasting large amounts of virtual space.

UKVA (User-kernel virtual addressing) provides a per-process kernel memory area. The same virtual space in each process is mapped to different physical pages, just like the userspace addresses (thus the "U"), but with the protections of kernel space (hence the "K"). A previous implementation actually located this area in the current user area. However, that implementation would have made it difficult to locate things other than user PTEs in the area. The implementation described here will locate the area inside the current kernel virtual area, and concentrate on locating PTEs in the area.

The first question is, "How big does it need to be?" An IA32 machine with PAE enabled has 4k pages, each controlled by a 8-byte pte entry. Each process will only need enough UKVA space to map in its own page tables.

4GB / (4K / page) = 1M pages

1M pages * 8 bytes/pte = 8MB of virtual space for ptes

For the purpose of this example, this space will be from 4G–8MB through 4GB; however, it does not really matter *where* this area is. It does not *need* to be aligned in a certain way, but this does make it easier to work with. First, the area should be aligned on a PTE page/PMD entry boundary (2MB with PAE). This will make it certain that the whole area can itself be mapped with only 4 PTE pages (more on this below). Secondly, the area should not straddle a PMD boundary, to avoid the initial setup being required to map more than 1 page, which makes it more expensive.

To map the required 8MB of virtual space, we require 2048 4K page table entries. We can fit 512 pte entries per 4K page, thus we require 4 UKVA PTE pages. Each time a pte page needs to be mapped in to the UKVA area, one of these 2048 pte entries contained in the 4 UKVA PTE pages will need to be set.

5.1 Initialization

Since the UKVA area will be the primary means for access to all PTE pages, it must be available for the entire life of the process's pagetables. For this reason, the initialization will occur in pgd_alloc(), at the same time as the top level pagetable entry is created.

On IA32, a pmd entry and a pte entry are the same size and PTRS_PER_PMD (the count of pmd entries per pmd page) is the same as PTRS_PER_PTE (the count of pte entries per pte page). Also, every time a pte page is allocated, a pmd entry is pointed to it. Each time you want to map an allocated PTE page, a pte entry is made, somewhere (highpte uses kmap() to do this). Instead of setting kmap() ptes, we will use UKVA ptes. This means there will be a 1:1 relationship between PMD pages/entries and UKVA PTE pages/entries.

During pgd_alloc(), the 4 UKVA PTE pages are allocated as soon as the PMD page which will point to them is allocated. The 4 pmd entries are made for the 4 pte pages, as are the corresponding 4 pte entries. However, making the PTE entries is slightly complex. One of the goals of UKVA is to replace HIGHPTE, which means that all of the PTE pages will be allocated in highmem, including the special 4 UKVA PTE pages. This means that the pte page that contains the 4 pte entries will need to be mapped via atomic kmap to make the entries. However, after this is done, they may

be accessed directly, without ever using kmap again.

This is the time where keeping the 8MB area from crossing a PMD boundary is important. Because of the 1:1 relationship, if the 4 pages are covered by more than 1 PMD page, they will also be covered by more than 1 PTE page, possibly doubling the amount of number of kmap calls which must occur.

5.2 Runtime

The bulk of the UKVA work is done in pte_alloc_map(). In keeping with the 1:1 relationship, each time a pmd_populate() is done, a UKVA PTE is also set. However, there are 2 possible ways to set a PTE with UKVA.

The first method is to simply index into the UKVA area, and set the PTE directly. Since the UKVA area is virtually contiguous, it can be accessed just like an array of every PTE in the system. The PTE contolling the first page of memory is at the start of the UKVA PTE space, just as the last PTE in the space controls the last page of RAM. It will always be known where the PTE for any given virtual address will be mapped. This also means the the 4 UKVA PTE pages themselves will be mapped to constant, known places.

The second method is used when pte_alloc_map() is asked to allocate a PTE for another process. Since the UKVA only contains the current process's pagetables, the pagetables of the other process must be walked, and appropriate entries made. During the walking process, the UKVE PTE pages must be mapped via kmap_atomic() so that they can be altered.

6 Hot & Cold pages

As the ratio of CPU speed to memory speed grows over time and CPU architectures change in ways such as pipelining, the efficient utilisation of processor caches becomes increasingly important. The hot and cold pages mechanism in Linux 2.5 (and its predecessors such as percpu pages) provide an important way to help increase the efficiency of the data cache. This is important for UP systems, but provides even greater benefit on SMP.

For each CPU in the system, for each zone of memory, we provide two queues for data pages: a hot queue, and a cold queue. The general precept is that pages in the hot queue are cache hot on that CPU, and pages on the cold queue are cache cold. Only 0-order pages (single page groups) are kept in these queues, the higher order allocations (multipage groups) are managed directly by the buddy allocator.

Both the hot and cold page lists allocate pages and free pages en masse from the buddy allocator for greater efficiency. This allows us to take multiple pages under one holding of the lock, whilst those codepaths and data management elements are cache hot. The lists have low watermarks, below which they will be refilled, and high watermarks, above which they will be emptied. Default batch size for allocations is 16 pages at a time; watermarks are 32–96 pages for the hotlists, and 0–32 pages for the cold lists.

The hot queue is managed as a LIFO stack—pages freed via the normal free_pages() route are pushed onto the hot stack (i.e. assumed to be cache warm). Further tuning in this area may be needed—the caller has better information about the cache warmth of the pages they are freeing than the generic routines. By default, page allocations come out of the hot list, unless __GFP_COLD is specified.

The cold list basically just functions as a batching mechanism for page allocations. It is used for pages that will not be first touched by the CPU in question (e.g. pagecache pages that

will be filled by DMA before they are read). This preserves the valuable cache hot pages for other uses, and saves cacheline invalidates for the CPU's cache. shrink_list(), shrink_cache(), and refill_inactive_zone() all free pages back into the cold list via the pagevec mechanism.

Below is a comparison of the kernel profiles of an equivalent workload (kernel compile) with and without the hot & cold pages mechanism, measuring how many ticks are spent in each routine, and which routines see the greatest change. Those labelled '+' get more expensive with hot & cold pages, those labelled '-' get cheaper. The 17.5% overall reduction in the total number of ticks spent is evident.

```
  Ticks  Percent Routine
   +243     0.0% buffered_rmqueue
   +197     6.6% page_remove_rmap
   +131     0.0% handle_mm_fault
   +116     0.0% fget
...
    -77   -15.7% release_pages
    -78   -34.4% atomic_dec_and_lock
    -89   -18.5% d_lookup
    -97   -16.2% __get_page_state
   -155   -35.6% link_path_walk
   -155   -23.8% copy_page_range
   -178   -27.8% shmem_getpage
   -193   -24.6% do_no_page
   -209  -100.0% pte_alloc_one
   -210   -26.0% zap_pte_range
   -303  -100.0% pgd_alloc
   -365  -100.0% __free_pages_ok
   -532   -28.0% do_anonymous_page
   -650   -37.0% do_wp_page
   -700  -100.0% rmqueue
  -4595   -17.5% total
```

The cost of rmqueue, pgd_alloc, pte_alloc_one, and __free_pages_ok has shifted into buffered_rmqueue, but it takes much less time to execute. The main cost for do_anonymous_page was in zeroing newly allocated pages, and the cost of do_wp_page is in copying pages from one to the other. Both obviously benefit greatly from the better cache warmth of the system. The profiles only show kernel time—userspace is actually the biggest beneficiary from this mechanism.

7 Page reclaim

In 2.5, the LRU lists were converted from global to per-zone. This makes it easier to free up one particular type of memory (e.g. ZONE_NORMAL) without affecting other types. It also breaks up the global locks and reduces cross-node cacheline traffic for NUMA machines. Following is the most significant elements from kernel profile data from a 2.4.18 kernel + NUMA patches doing a kernel compile on a 16-way NUMA-Q:

```
  2763 _text_lock_dcache
  2499 _text_lock_swap
  1199 do_anonymous_page
   763 d_lookup
   651 lru_cache_add
   646 __free_pages_ok
   612 do_generic_file_read
   573 lru_cache_del
...
```

The _text_lock_swap entry is the pagemap_lru_lock

The page-reclaim daemon (kswapd) must touch large amounts of data, both the user pages being manipulated, and their control structures (e.g. the LRU lists). However, on NUMA systems this is extremely problematic, as it causes a lot of cross-node memory traffic. Hence the global daemon was replaced with a per-node daemon, each of which only scans its own nodes pages, which is much more efficient.

One of the last major global locks in the VM was pagemap_lru_lock. Andrew Morton's pagevec implementation reduced contention on it by 98% by batching page operations together

into ‘pagevecs’—vectors of pages that could be manipulated together as groups more efficiently.

8 rmap

Whilst the pagetables of each process provide a mapping from that virtual address space to the physical addresses backing it, in Linux 2.4, there is no easy way to map back from a physical address to a virtual address. This is what rmap provides—a “reverse” mapping from the physical address back to the set of virtual addresses mapping it.

To reclaim memory, 2.4 used a “virtual scan”—walk each process, and see if we can unmap the physical pages it is using. 2.5 uses “physical scan”—walk each page of RAM, and see if it can be freed (this requires the rmap mechanism). This new mechanism has several advantages, perhaps the most important of which is stability. It has proven significantly more robust under pressure than the virtual scan in 2.4 code.

Whilst the usage of the rmap mechanism has remained fairly stable, the method of keeping the data for the reverse mapping has been a source of more contention and trouble. The current 2.5 code as of 2.5.68 uses a mechanism called “pte-chains” which keeps (for each physical page) a simple linked list of pointers back to the pte entries of the processes mapping each page.

These pte-chains have several problems:

- Locking
- Space consumption
- Time consumtion

The locking was at first implemented as a global lock, which was actually shared with the worst existing global VM lock (pagemap_lru_lock). This caused massive lock contention (data from a 12-way NUMA-Q), as seen in Table 1.

Therefore, the locking was changed to a per-chain lock, which was subsequently compacted into a 1 bit lock embedded in the flags field of the struct page to avoid more space consumption problems. This reduced kernel compile times by more than half on 16-way NUMA-Q (from 85s to 40s).

The next problem with pte-chains is the space consumption. A simple singly linked list will consume 4 bytes per entry for the pointer to the PTE and 4 bytes per entry for the pointer to the next entry. Two methods were used to alleviate this:

1. Pages with only a single mapping can use the “page-direct” optimisation—instead of storing the pointer to the linked list in the struct page, we use the same space to point directly to the only PTE by using the pte union introduced into struct page:

   ```
   union {
       struct pte_chain *chain;
       pte_addr_t direct;
   } pte;
   ```

   And this switch is governed by the PG_direct flag from the flags field in the struct page.

2. The lists are grouped by cacheline, allowing multiple PTE pointers per ‘list next’ pointer. Not only does this reduce the size of the linked list by almost half (assuming sufficient grouping), but it also greatly increase the data locality and cache efficiency for walking the chain.

Moreover, this space consumption all comes from low memory, an extremely precious resource on large 32-bit machines. To take

SPINLOCKS		HOLD		WAIT						
UTIL	CON	MEAN	MAX	MEAN	MAX	% CPU	TOTAL	NOWAIT	SPIN	NAME
45.5%	72.0%	8.5us	341us	138us	11ms	44.3%	3067414	28.0%	72.0%	pagemap_lru_lock
0.03%	31.7%	5.2us	30us	139us	3473us	0.02%	3204	68.3%	31.7%	deactivate_page+0xc
6.0%	78.3%	6.8us	87us	162us	9847us	9.4%	510349	21.7%	78.3%	lru_cache_add+0x2c
6.7%	73.2%	7.6us	180us	120us	8138us	6.4%	506534	26.8%	73.2%	lru_cache_del+0xc
12.7%	64.2%	7.1us	151us	140us	10ms	13.4%	1023578	35.8%	64.2%	page_add_rmap+0x2c
20.1%	76.0%	11us	341us	133us	11ms	15.0%	1023749	24.0%	76.0%	page_remove_rmap+0x3c

Table 1: Massive lock contention on 12-way NUMA-Q

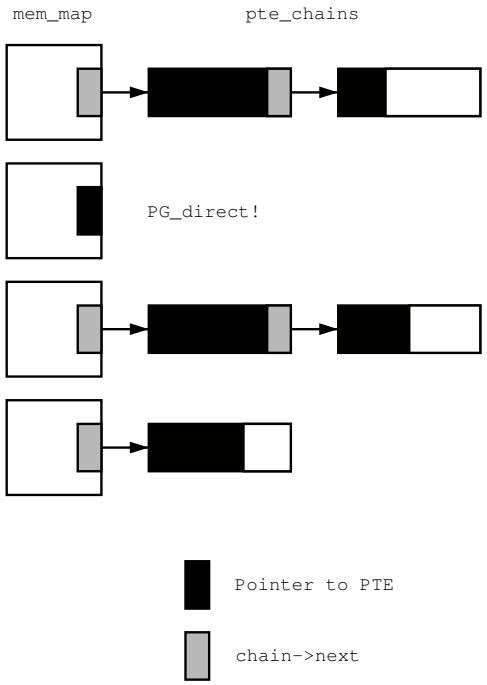

Figure 2: pte-chain based rmap

our example of 5000 processes sharing a 2GB memory segment again, not only do we now have 20GB of pagetables, but 10GB of pte_chains. Whilst the 20GB of pagetables can at least be moved off into high memory, this is not easy to do for pte chains. In order to move the chains into high memory, the "next element" pointers would need to become physical addresses, instead of virtual ones. Not only are these larger (36 bits instead of 32), they also need to be mapped into virtual addresses before use, an incredibly expensive procedure for walking the linked lists.

Last, but not least, of the problems is the time consumption. For every page used, and for every process that uses it, we must take a lock (using an expensive atomic operation) and create a page entry. Worse still, when we tear down the mapping, we must take that same lock, and then walk the pte-chain looking for the element to free. This takes approximately a linear amount of time, depending on the number of elements sharing that page. Even for just a load of 128 on SDET, the kernel profile shows the rmap functions massively dominating:

```
86159 page_remove_rmap
38690 page_add_rmap
17976 zap_pte_range
14431 copy_page_range
10953 __d_lookup
 9978 release_pages
 9369 find_get_page
 7483 atomic_dec_and_lock
 6924 __copy_to_user_ll
 6830 kmem_cache_free
```

The problem is especially acute under a workload such as SDET that does significant amounts of fork/exec/exit traffic, where mappings must be continually built up and then torn down again. I see this problem as fundamental with a page based approach; though it may be alleviated somewhat by tuning, it is still a per-page operation, and thus too expensive. Even for a simple kernel compile, the page_remove_rmap is still the most expensive function in the whole kernel:

```
23222 page_remove_rmap
14034 do_anonymous_page
 7638 __d_lookup
 6406 page_add_rmap
 5188 __copy_to_user_ll
 3656 find_get_page
 3429 __copy_from_user_ll
 3126 zap_pte_range
 2108 do_page_fault
 1925 atomic_dec_and_lock
 1852 path_lookup
```

8.1 rmap shadow pages

Ingo Molnar has proposed a new rmap method which I shall call "rmap shadow pages," which alleviates some of the problems with pte-chains, but is still page based. At the time of writing, there is no implementation available, but some of its properties may be determined by analysis.

Instead of the chains being allocated as needed on a cacheline sized block, it is proposed to allocate two "shadow pages" for each pte-page (page filled with PTE entries). These would form a doubly-linked list with other shadow pages. To retrieve the list of PTE entries for a particular page, one would consult the pte-chain pointer in the relevant struct page, and walk the list (similarly to PTE-chains).

The PTE entry itself need not be stored inside the rmap shadow page, but the rmap pages are implicitly "linked" to the pte page in some fashion (e.g. being placed together in some contiguous group of two pages). However, the cacheline locality characteristics seem to be against this method for scanning, as it involves touching a separate cacheline for every element in the list.

To add a page to the linked list, we would take the pte_chain lock, and add ourselves to the head of the list (one modification to the shadow page, plus one to the struct page). To remove a page from the list would not require walking

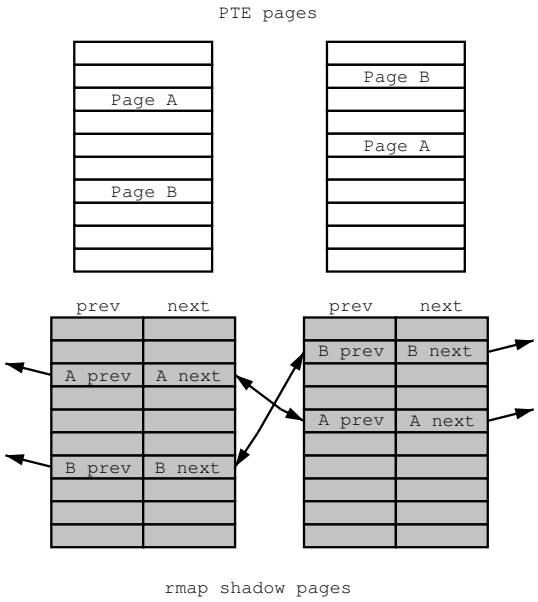

Figure 3: rmap shadow pages

the list (as for pte_chains), but we would traverse from the PTE page to the corresponding shadow page, map both its prev and next element pages, and perform a regular unlink for a doubly-linked list.

One of the major advantages of the rmap shadow pages method is that the rmap data can be more easily moved into the highmem area. However, this is not without cost—each page accessed must be mapped via kmap, which has proven expensive for PTE pages in the highpte implementation.

Whilst rmap shadow pages may fix some of the problems of pte-chains, it is still page-based, and thus requires a large amount of data manipulation. It is therefore unlikely to solve the fundmantal time and space problem, though moving the chains into high memory may be worthwhile.

8.2 Object based rmap

Other operating systems have taken a different approach to the physical to virtual address mapping problem. K42 has taken an approach based on file objects (akin to the Linux address_space structure), which seemed to be promising after initial discussions with Orran Krieger and other K42 engineers.

Instead of keeping a reverse mapping for each page in the system, we can discover the list of virtual addresses by going from the struct page to the address_space object. From the address_space object, we can walk a list of vmas—areas of process virtual memory which map that object. By adding the offset within the file (stored in struct page as "index") to the base virtual address of each vma, we can derive the virtual address within each process. From there, we can walk the pagetables to find the appropriate PTE.

The key advantage of this method is that there is no overhead at all for the setup and teardown of each page. This comes at the cost of higher overhead to find the PTEs at scanning time (the list of VMAs and the pagetables for each process must be walked). However, many workloads will run without memory pressure, or only have pressure on the caches, which are easily freed, so the approach seems promising.

However, there are a few fundamental problems with an object-based approach within Linux. For one, there is not a backing file object for every page in the system, some pages (e.g. private process data allocated via malloc/sbrk) are anonymous, i.e. not associated with any file. Whilst it would be possible to create a file object for anonymous pages, this would not be a simple change.

Another problem is that the calculation of adding the offset to the base virtual address within the process assumes that the vma is linear. Whilst this used to be true, the 2.5 kernel contains a new mechanism called "sys_remap_file_pages" that allows for non-linear VMAs.

Bearing in mind that pte_chains are most expensive under heavy sharing (the linked list must be walked for page_remove_rmap), some analysis was taken of the length of the pte-chains for both file-backed and anonymous objects. This showed that the anonymous objects were only mapped once for nearly all pages, and the shared mappings were nearly all file-backed.

Based on these observations, and the simplicity of implementation, a "partially object-based" scheme was proposed. This used the object-based mappings for file-backed pages, and the pte-chains method for anonymous memory (and for nonlinear mappings). Dave McCracken implemented this scheme, and it was very effective in reducing both the space and time taken by pte-chains.

Kernbench-16: (make -j 256 vmlinux)				
	Elapsed	User	System	CPU
pte-chains	47.21	569.17	139.55	1500.67
partial objrmap	46.09	568.19	121.83	1496.67

Note the 12% drop in total system time.

SDET 64 (see disclaimer)		
	Throughput	Std. Dev
2.5.68	100.0%	0.2%
2.5.68-objrmap	121.2%	0.3%

Again, kernel profiles clearly show the reduction:

-30518	-78.9%	page_add_rmap
-72197	-83.8%	page_remove_rmap

Partial objrmap also drastically reduced the number of pte_chain objects in the slab cache, graphically demonstrating that file_backed chains are predomininant. For a `make -j256 vmlinux`:

pte-chains	24116	pte_chain objects in slab cache
objrmap	716	pte_chain objects in slab cache

(a 97% reduction).

However, at the time of writing, there are still a couple of remaining objections to partial objrmap:

1. The interaction with sys_remap_file_page's nonlinear vmas is complex and convoluted due to the conversion to and from pte_chains which may be necessary. However, it is generally recongnised that sys_remap_file_pages is a special case. If those vmas are pre-declared as nonlinear, most of the problems disappear. Furthermore, for the intended use of sys_remap_file_pages (windowing onto large database shared segments), it is acceptable to lock those pages into memory (this is normally done in any case), in which case none of this information will ever be needed, so it is not necessary to keep it.

2. If 100 processes each map 100 vmas onto the same address space, then objrmap would have to scan 10,000 regions, not simply the 100 mappings that the page-based methods might have had for their chains. Whether this corner case is important or not is a matter of debate, as it was exactly the case that sys_remap_file_pages was designed to fix, and callers should be using that method for such an unusual situation. However, a simple optimisation is proposed that should alleviate the problem in any case:

For each distinct range of addresses mapped by a vma inside the address_space, we define an address_range. This takes advantage of the fact that we are likely to remap the same range repeatedly (e.g. for shared libraries). From each shared range, we attatch a list of vmas that map that range. Furthermore, we sort the list of address ranges by start address.

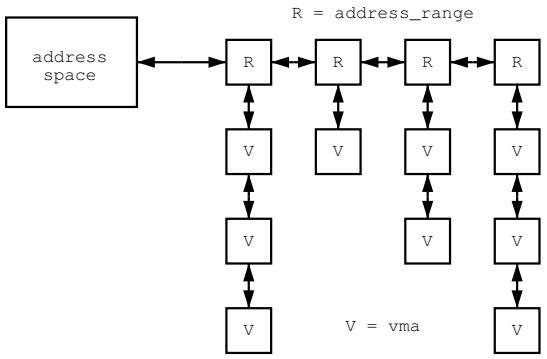

Figure 4: list of lists

```
struct address_range {
    unsigned long           start;
    unsigned long           end;
    struct list_head        ranges;
    struct list_head        vmas;
};
```

9 NUMA support

On NUMA systems, it is more efficient to access node-local memory than remote memory. Thus Linux tries to allocate memory to a process from the node it is currently running on, providing for more efficient performance. This also allows for locality of memory and control structures, reducing cross-node cacheline traffic.

By default, we allocate memory from the local node (if some is free), then round robin amongst the remaining nodes by node number, starting at the local node, and progressing upwards. Kernel code can also request memory on specific nodes via alloc_pages_node().

Matt Dobson has created a simple NUMA binding interface for userspace, allowing processes to request memory from a specific node,

or group of nodes. This is very useful for large database applications, which wish to bind "partitions" of the database to certain nodes.

Several critical control structures (e.g. the mem_map array—the control structures for the physical RAM pages, and the pgdat—the node control structure) are now allocated on the nodes own memory, providing for better performance on NUMA systems. More items (e.g. the scheduler data, and per-cpu data) should be migrated into node-local memory in the future.

Of the three main memory allocators (alloc_pages, vmalloc, slab cache), only the slab cache is not NUMA aware. Manfred Spraul has created patches to do this, but we have not seen observeable performance benefits from this method yet. Part of the problem is the inherently global nature of many of the caches (eg the directory cache), which will need to be attacked first.

Some of the kernel architectures (ones with hardware assistance from the CPU) have kernel text replication functioning. This makes a copy of the kernel data to each node, and processes will use their own node's local copy of the data, reducing backplane traffic, and interconnect cache pollution. Replicating the read-only portion of shared libraries also seems promising, though this has not yet been implemented in Linux. Replicating any data that is not read-only is likely to be too complex to be beneficial.

10 Legal