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.*

A 2.5 Page Clustering Implementation

William Lee Irwin III
IBM Linux Technology Center
wli@holomorphy.com | wlirwin@us.ibm.com

Abstract

Page clustering is a form of "large pages" that increases the kernel's minimum allocation unit for physical memory (base page size). There are several good reasons to do this. One is a form of prefaulting accomplished by instantiating groups of PTE's mapping a given base page. Another is a constant factor reduction of the number of objects the kernel must traverse in order to manipulate a given collection of pages. The increase in PAGE_SIZE also implies an increase in PAGE_CACHE_SIZE, which enables the use of filesystems with larger blocksizes. Last, but not least, the constant factor reduction of lowmem consumed by mem_map is crucial for the performance of 64GB i386 machines.

Page clustering has a number of technical challenges involved in a 2.5 counterpart of the 2.4.7 implementation. First, highpte poses unusual difficulties, as neither sub-PAGE_SIZE highmem allocations nor sub-PAGE_SIZE kmapping were supported in the original implementation. Rmap also poses challenges, as it makes direct assumptions about PTE's being of size PAGE_SIZE. Finally, arch code above all makes many assumptions about PAGE_SIZE's relationship to the area mapped by PTE's, particularly in arch support and VM initialization code.

In summary, the author will describe the problems that arose during his implementation of page clustering for 2.5 along with their solutions, for an audience of kernel programmers.

1 What is page clustering?

1.1 Background

Memory present in a system is described by physical addresses. Most (if not all) modern machines are byte addressable, but the MMU usually operates at a lower "resolution," and its finest resolution is what page clustering refers to as MMUPAGE_SIZE. When the MMU's translations are set up, be they in hardware-interpreted data structures or in software-programmable TLB's, they refer to "page frames" of that size or larger, which effectively are a unit of measurement for memory. Similarly one may refer to virtual memory in those units, and arbitrary relationships between virtual page frames and physical page frames are constructible with a combination of hardware and software translation tables.

Demand-paged virtual memory systems, when a task takes a TLB miss not resolvable via the kernel's software translation tables (which may be interpreted directly by hardware) are then faced with the task of finding a physical page frame to back a virtual page frame with.

Without page clustering, the kernel maintains a data structure, the struct page, representing each physical page frame, and another, the page table entry, to represent each virtual page frame. When not constrained by hardware, the kernel is free to make ridiculous choices of structures for the page tables. For instance, each virtual page frame could (in principle) be represented by a node in a binary search tree or a linked list. Linux® uses radix trees as mandated by hardware on i386 on all architectures, which are somewhat more efficient than various choices, though some architectures natively use other structures such as inverted page tables for them.

The page tables are assisted by a binary search tree of virtual extents representing either extents of files or zero-filled regions, whose nodes are called `vma`'s. The physical page is chosen so as to arrange virtual contiguity of file pages in tandem with file offset contiguity, or otherwise to fetch largely arbitrary pages and zero them out before mapping them. When the relationship of a pagetable entry to a physical page frame and its corresponding `struct page` data structure, is restricted to within a single `vma` it is a 1:1 relationship.

A system for tracking memory in use and not in use is built around this relationship, and so the `MMUPAGE_SIZE` became the allocation unit for memory. `PAGE_SIZE` is used to simultaneously refer to the notion of the MMU's finest granularity and the memory allocator's finest granularity in preexisting Linux ® code.

1.2 How page clustering differs

First, one should observe that if the MMU's finest granularity is `MMUPAGE_SIZE`, one may simulate an MMU with a granularity of any power of two multiple (`PAGE_MMUCOUNT`) by simply instantiating `PAGE_MMUCOUNT` PTE's at a time, and making each `struct page` refer to `PAGE_MMUCOUNT` contiguous and aligned native page frames. Also, if the pagetables are not constrained by hardware, one can easily alter their structure to only have one PTE for each `PAGE_SIZE` instead of `MMUPAGE_SIZE` and by so doing reduce their space consumption. Additionally, if the MMU supports translations of size `PAGE_SIZE` one can simply perform one TLB insertion (or PTE instantiation if hardware-interpreted) for each `PAGE_SIZE` area mapped by the pagetables.

One could say this is a "weak form" of page clustering. It has the undesirable side-effect of breaking binary compatibility and hence not being transparent, but has several advantages. The port of Linux ® to the IA64 processor already uses this low code impact technique for performance reasons, as it reduces TLB misses and the overhead of manipulating large collections of pages by a factor of `PAGE_MMUCOUNT`. Some performance benefits for I/O are possible, as physical contiguity is better preserved so larger scatter/gather lists are possible, though this is offset by a larger cost of preparing buffers for small I/O transactions. BSD's VAX port did it this way.

The binary incompatibility inherent in the above approach makes it unsuitable for practical deployment on systems with significant preexisting userbases. For instance, ELF executables are linked in ways mandating differing protections within what could potentially be a single `PAGE_SIZE` virtual region, and mmap() is often performed at offsets that are not `PAGE_SIZE`-aligned or in lengths divisible by `PAGE_SIZE`. To address the `mmap()` granularity issue, the 1:1 relationship between virtual page frames and accounting structures for physical memory must be extended to `PAGE_MMUCOUNT`:1. There is also a very invasive audit required to enforce the newly introduced distinction between `MMUPAGE_SIZE` and `PAGE_SIZE` by programming dimensional analysis into various address and index calculations. This could be called the "strong form" of page clustering.

The solution, in high-level terms, essentially has two cases for userspace. The first, which is easier, is file-backed memory. The unit of memory cacheing file contents is `PAGE_CACHE_SIZE`, which (for 2.5) is identical to `PAGE_SIZE`. An index in units of `PAGE_SIZE` is usable for recovering the struct page representing the area of the file that would need to be faulted in. However, to preserve mapping semantics one must also recover an offset into the area represented by the `struct page` in units of `MMUPAGE_SIZE`. The second case is anonymous memory, which is not forced to be simultaneously virtually and physically contiguous by virtue of its contents. Userspace demands one `MMUPAGE_SIZE` unit of memory but receives `PAGE_SIZE` unit of memory, and so to prevent very noticeable amounts of waste, one scans nearby PTE's for other virtual pages anonymizing faults, that is, write faults on COW file pages or on the zero page, could be taken on. These are candidate pages for copying (the zero page is special cased to use faster zeroing algorithms on most architectures). The anonymizing case results in a complex relationship between the virtual pages in a process and the anonymous page.

In summary, page clustering divorces the kernel's internal allocation unit, or the size of an area represented by a struct page, from the notion of the MMU's mapping granularity with the constraint that the allocation unit be larger.

1.3 Why page clustering?

Page clustering introduces several advantages. The first is that by using a larger unit for cacheing file pages, one can support filesystems with larger block sizes. The second is that the additional physical contiguity introduced by the larger allocation unit allows one to construct larger scatter gather lists for I/O (again with the proviso about preparing write buffers). The third is that the number of objects in various collections of pages is reduced for a linear speedup of the algorithms. The fourth is that the page faults may be batched, reducing the page fault rate.

The fifth, which is the primary reason why this project to resurrect the 2.4.7 page clustering patch was carried out, is largely specific to i386 PAE, though possibly also applicable to 32-bit kernels running on large memory 64-bit machines. `sizeof(struct page)/PAGE_SIZE` is the constant of proportionality for the fraction of memory consumed by the `struct page`'s required to account for all the physical memory in the system. On 32-bit systems with extended addressing or when the kernel runs in 32-bit mode, this is irrespective of virtualspace and the total memory consumed may be larger than kernel virtualspace. For instance, with a fully-populated 40-bit physical address space, a 32-bit virtualspace, a 4KB `PAGE_SIZE`, and a 64B `sizeof(struct page)`, the coremap is 16GB in size, which is infeasible to simultaneously map. Page clustering reduces this space overhead by a factor of `PAGE_MMUCOUNT`, which is arbitrary (within the constraints of the quality of implementation), and so renders the coremap's space overhead $O(1)$ with respect to physical memory.

2 Implementation

2.1 Early boot

The issues encountered in early boot were largely simple, but widespread. Early boot debugging was done on a 16 processor NUMA-Q ® with 16GB RAM. First, pagetables and various fragments of memory that were formerly assumed to be 4KB but described with `PAGE_SIZE` needed to be updated, including mappings for the IO-APIC, numerous pagetables, and structures like the idle threads' stacks. Then memory detection required various kinds of dimensional analysis to properly calculate coremap indices from page frame numbers and vice-versa.

Numerous index calculations and indexing operations into the coremap were broken. They had no counterpart in 2.4.x, but didn't require much thought to correct:

```
#define pfn_to_page(pfn)  (mem_map + (pfn))
#define page_to_pfn(page) \
    ((unsigned long)((page) − mem_map))
#define pfn_valid(pfn)    ((pfn) < max_mapnr)
```

became

```
#define pfn_to_page(pfn)    \
        (&mem_map[(pfn)/PAGE_MMUCOUNT])
#define page_to_mapnr(page) \
    ((unsigned long)((page) − mem_map))
#define page_to_pfn(page)   \
    (PAGE_MMUCOUNT∗page_to_mapnr(page))
#define pfn_valid(pfn)      \
    ((pfn) < max_mapnr∗PAGE_MMUCOUNT)
```

and so on.

Of the issues, `kmap_pte` and `pkmap_page_table` were particularly troublesome; to get booting, they were removed in favor of walking kernel pagetables, but are to be reinstated in the near future. The issue was that they were allocated 4KB at a time using the bootmem allocator, but were assumed to point at contiguous pagetables capable of mapping the entire permanent kmap and atomic kmap arenas, which had grown to where they required multiple pagetables each.

An unusual issue arose from maintaining an 8KB stack size while raising `PAGE_SIZE` to arbitrary sizes. `fork_init()` first received a divide by zero from the following code fragment:

```
/∗ create a slab on which task_structs can be allocated ∗/
task_struct_cachep =
    kmem_cache_create("task_struct",
        sizeof(struct task_struct),0,
        SLAB_MUST_HWCACHE_ALIGN, NULL, NULL);
if (!task_struct_cachep)
    panic("fork_init(): cannot create
task_struct SLAB cache");

/∗
 ∗ The default maximum number of threads is set to a safe
 ∗ value: the thread structures can take up at most half
 ∗ of memory.
 ∗/
max_threads = mempages /
(THREAD_SIZE/PAGE_SIZE) / 8;
```

This was clearly due to `THREAD_SIZE/PAGE_SIZE` vanishing. But then unusual errors arose for unclear reasons. As it turned out, I'd changed kernel stacks to be slab allocated, but, the kernel stacks were so small compared to `PAGE_SIZE` they used on-slab slab management and so failed to be 8KB-aligned. Changing the threshold to use off-slab slab management for objects larger than `MMUPAGE_SIZE` in addition to the other criteria sufficed, with zero runtime impact on the `PAGE_SIZE == MMUPAGE_SIZE` case.

Next, the placement of vmallocspace relative to fixmapspace and the overrunning of vmallocspace by fixmapspace for unusually large values of `PAGE_SIZE` became issues. This was resolved by some painful compile-time mechanics to shove the kmap and permanent kmap windows into vmallocspace, dynamically size them with respect to `PAGE_SIZE`, and make poor guesses in assembly as to the boundaries between vmallocspace and `ZONE_NORMAL`. The boundaries out to be safe because the assumptions were not truly used apart from an indirect reference to them via `MAXMEM`. Specifically:

```
#define VMALLOC_END
(FIXADDR_START-2*MMUPAGE_SIZE)

#define __VMALLOC_START \
    (VMALLOC_END - VMALLOC_RESERVE  \
         - 2*MMUPAGE_SIZE)
#define VMALLOC_START \
(high_memory                            \
     ?  max(__VMALLOC_START,            \
            (unsigned long)high_memory) \
     :  __VMALLOC_START                 \
)
#define __MAXMEM \
((VMALLOC_START - 2*MMUPAGE_SIZE  \
        - __PAGE_OFFSET)          \
        & LARGE_PAGE_MASK)
#define MAXMEM \
__pa((VMALLOC_START-2*MMUPAGE_SIZE) \
    & LARGE_PAGE_MASK)
```

This is actually a side effect of a design decision which makes the virtualspace layout change dynamically with PAGE_SIZE. That is, virtual mapping windows raise an issue now that the area callers want to map is usually PAGE_SIZE in size, and to make it the size they expect, fixmapspace must grow. There is an alternative design possible, which is to keep fixmapspace a fixed size or at most some fixed size, and to have "sliding windows into a partial page." Using such an API would proceed something like the following:

```
int k;
void *old, *new;
old = kmap_atomic_start(old_page, KM_USER0);
new = kmap_atomic_start(new_page, KM_USER1);
for (k = 0; k < PAGE_KMAP_COUNT; ++k) {
    memcpy(new, old, KMAP_SIZE);
    old = kmap_atomic(old_page, KM_USER0, k);
    new = kmap_atomic(new_page, KM_USER1, k);
}
kmap_atomic_end(old_page, old, KM_USER0);
kmap_atomic_end(new_page, new, KM_USER1);
```

This is somewhat more invasive, but more efficient with respect to virtualspace. As PAGE_SIZE grows in a 32-bit environment with progressively more extended physical memory, such measures become progressively more prudent. However, the need to take such measures may be significantly mitigated by eliminating the permanent kmap pool in combination with using per-cpu pagetables for the kmap windows, as the typical targets needing the largest PAGE_SIZE values have a maximum of 32 or 64 cpus. Eliminating the permanent kmap pool has the additional advantage of preventing deadlocks caused by a number of tasks each attempting to acquire multiple permanent kmaps but acquiring fewer than desired by the time the pool is exhausted.

2.2 coremap initialization

The coremap initialization is worthy of its own discussion. First, in order to satisfy the early boot setup code, the coremap must be laid out so it maps PAGE_SIZE units of memory to properly aligned positions in the coremap. Simultaneously, most (if not all) of the calculations are done with pfn's so various bits of dimensional analysis must be programmed in.

First, zone->zone_start_pfn and zone->spanned_pages need to be treated consistently. Then, zones_sizes[] needs to be converted to pass PAGE_SIZE units and free_area_init_core() fixed up to increment its pfn counter by PAGE_MMUCOUNT. bad_range() must then be adjusted for unit conversion before doing its bounds checks. Finally, the page allocator need not keep so many orders around to satisfy allocations of a given size, so use MAX_ORDER - PAGE_MMUSHIFT instead of MAX_ORDER.

Bootmem also needed large adjustments; they were largely done to make its own internal accounting based on MMUPAGE_SIZE and then interface it with the page allocator which has a PAGE_SIZE granularity. This ended up being rather invasive.

2.3 Kernel pagetables

vmalloc() usage was too widespread to undergo a full audit for space conservation. The choice was between using PAGE_SIZE or MMUPAGE_SIZE as the unit of vmalloc() mapping and allocation, and I chose MMUPAGE_SIZE. This is transparent to userspace, so either would be legitimate, but on i386 PAE vmallocspace is too constrained to take internal fragmentation hits. This meant that instead of a full audit for space conservation, a full audit for misuse of PAGE_SIZE is needed. This turned out not to be very problematic at all, as few drivers needed the conversion, and those that did had mild failure modes, failing only to probe.

page_table_range_init() and relatives greatly disliked the change of units and the ambiguous location of kmap_pte and pkmap_page_table. It proved infeasible rapidly bring up the system while preserving them intact, so they were removed and the code greatly simplified at the expense of a very large diff.

2.4 Process pagetables

User pagetable manipulations consisted largely of straightforward substitutions in pagetable code. Of course, something was missed. It appeared that an unusual binary compatibility bug arose with respect to shared libraries that was very difficult to trigger. The cause of this was that there was only one caller of pte_modify() in the core VM in a corner case of mprotect(). This passed a first pass of inspection because _PAGE_CHG_MASK didn't trip grep, but it turned out to rely on PTE_MASK, which by virtue of the macro indirection, also slipped past grep. After over a week of chasing it, the substitution that slipped through my fingers was finally carried out.

The next "interesting" binary compatibility bug was that core dumps were corrupted. The get_user_pages() calling convention had become lossy. It was returning struct page's to refer to the areas mapped by PTE's, and it along with follow_page() was the only area of the kernel exibiting this particular kind of confusion. The solution was to return pfn's and not struct page's, and was highly successful. Badari Pulavarty assisted in implementing the portion relevant to direct I/O.

The most interesting bug of all was actually the first, which prevented userspace from running at all. /sbin/init would be stuck in a loop somewhere in userspace, and it could only very rarely be caught in the kernel. What eventually had to be done to track down the issue was to log all page faults. What was eventually discovered was that pid 1 has a special status in the kernel, and loops when taking invalid faults instead of being delivered SIGSEGV. After some poking around, it became evident they were always anonymous pagefaults.

So at first, the workaround was to fragment anonymous pages. But this could only be temporary in order to meet the performance goals. The issue was resurrected when it came time to attempt to fully utilize anonymous pages for performance reasons. It took some time to come around to examining the contents of the purportedly zeroed memory, but eventually divining the page pointed to by the PTE taking the fault, which pointed to the zero page. And the fact a nonzero address was being fetched from the zero page prompted the examination of its contents. By an unusual coincidence the author had been implementing the GDT setup for an i386 executive earlier that day, and noticed a very clear resemblance to the contents of the supposed zero page. Very shortly thereafter it was discovered that the `empty_zero_page[]` used on i386 as backing memory for the zero page was a 4KB array followed immediately by the kernel's GDT. The bug was resolved by using a custom-allocated and zeroed page instead of the `struct page` tracking `empty_zero_page[]`.

Finally, userspace pagetables required fixups in order to prevent extremely wasteful fragmentation. The code turned out to be somewhat hairy, as it required reference counting pagetable pages and some scanning of PMD entries in an aligned `PAGE_MMUCOUNT`-sized group. Furthermore, in order to interoperate with highpte, significantly more complex definitions of `pte_offset_map()`, `pmd_populate()` and relatives were required.

```
#define pte_offset_map(dir, address)    \
((pte_t *)                              \
  kmap_atomic(pmd_page(*(dir)),KM_PTE0) \
 + (PTRS_PER_PTE                        \
    * ((pmd_val(*(dir))/MMUPAGE_SIZE)   \
       % PAGE_MMUCOUNT)                 \
 + pte_index(address))                  \
)
```

`pmd_populate()` became too large to paste here because it had to deal with several issues to recover from partial unmappings of the `PAGE_MMUCOUNT` PMD group and PTE page refcounting. For the wary, it collapses to its prior size when `PAGE_MMUCOUNT == 1`.

2.5 file-backed memory

Handling userspace faulting semantics for file-backed memory was actually trivial. The most unsophisticated fault handling scheme imaginable suffices.

There was a small issue with `sys_remap_file_pages()` where the populate methods used the `install_page()` API internally to perform the dirty work of walking the pagetables down to the PTE to edit, and as it referred to the location to map by the `struct page`, lost the offset into the page to map. This was trivially corrected with an additional argument with the offset.

2.6 Swap-backed memory

Swap faults are not truly worth optimizing with pagetable scanning; they don't fragment like freshly zeroed anonymous pages because the swapcache is an effective lookup structure and userspace can fetch things just fine. Instead they are faulted in one by one, and that simplified things at least temporarily while the scanning code wasn't in place.

The organization of the swap map differs from the 2.4.7 patch, which created a swap map entry for each `MMUPAGE_SIZE` piece of a page, and so had to account for reference counts on the page held by multiple swap entries. The 2.5 page clustering implementation instead uses a single swap map entry for every `PAGE_SIZE`-sized page, and so simplified swap reference count semantics, reduces the vmallocspace consumption of the swap map by a factor of `PAGE_MMUCOUNT`, and reduces the search space for swapoff. Some differences there are also visible with the encoding of `swp_entry_t`'s, which directly play with swap map indices and offsets into pages in various points throughout the core VM where beforehand they didn't need to..

2.7 anonymizing faults

There is a problem to solve caused by the fact that a process faulting on anonymous memory requests MMUPAGE_SIZE bytes of memory but is granted PAGE_SIZE bytes of memory. Again, there is more than one way to deal with this.

The first, not used here, is to maintain a one PAGE_SIZE area as a "ready list" and service anonymous faults until it's exhausted.

The second is to speculatively prefault neighboring anonymous pages in order to utilize the entire anonymous page. Scanning neighboring PTE's for zero-mapped or COW pages (i.e. to be anonymized). This has the potential to reduce the fault rate for some loads at the cost of not guaranteeing full utilization. Initial indications appear to be that even heuristics that appear relatively weak in comparison to those of the 2.4 patch suffice.

The logic is relatively complex, and some additional complexity as compared to the 2.4.x code was added by simultaneously scanning PTE's both upward and downward. Some additional code is required to cross vma boundaries and detect whether a given page is anonymous or COW. Crossing pagetable page boundaries was not implemented, for the basic reason that PAGE_MMUCOUNT*PMD_SIZE is enough virtualspace to scan to mitigate most of the fragmentation, and also to remain future-compatible with pagetable sharing, which is somewhat adverse to crossing pagetable pages. Additionally, totally unbounded scanning could result in some overhead.

When the scanning code is done, what it has done is assembled a vector of pfn's for all the mmupages it has to copy, and they are by no means contiguous. In order not to be grossly TLB-inefficient, an interface is provided to map vectors of pfn's, kmap_atomic_sg(). The use of it is obvious, as it maps each component of the pfn vector to a virtually contiguous PAGE_SIZE virtual area in its corresponding piece of the virtual page, and the only non-straight-line code in copying is checking for the zero page.

2.8 I/O

I/O by and large had relatively simple issues. The i386 PCI DMA API had some address calculations in need of minor substitutions, and the block layer was largely immune to the whole affair apart from direct I/O and SCSI ioctl's using get_user_pages(). An unfortunate limitation exists in that the block layer is incapable of dealing with 512*q->max_sectors < PAGE_SIZE. I didn't produce a fix for this, as it's a somewhat obscure condition that can only occur when particularly crippled devices meet particularly large value of PAGE_SIZE. I feel that it should eventually be handled as part of the implementation.

IDE had a small issue in that its PRD tables were sized in terms of PAGE_SIZE, which it appears to expect not to vary from 4KB. AGP also had an unusual issue involving mapping its aperture. But most drivers that failed simply performed a vmalloc() or ioremap() of an area sized proportionally to PAGE_SIZE during initialization and failed to probe, which was harmless apart from failing to provide functionality (i.e. no data corruption) and very easy to correct. The starfire ethernet adapter fell in this category.

3 Trademarks

This work represents the view of the author and does not necessarily represent the view of IBM.

IBM is a trademark of International Business Machines Corporation.

Linux® is a trademark of Linus Torvalds.

Other company, product or service names may be trademarks or service marks of others.