Improving Kernel Performance by Unmapping the Page Cache

James Bottomley
SteelEye Technology, Inc.
James.Bottomley@SteelEye.com

Abstract

The current DMA API is written on the founding assumption that the coherency is being done between the device and kernel virtual addresses. We have a different API for coherency between the kernel and userspace. The upshot is that every Process I/O must be flushed twice: Once to make the user coherent with the kernel and once to make the kernel coherent with the device. Additionally, having to map all pages for I/O places considerable resource pressure on x86 (where any highmem page must be separately mapped).

We present a different paradigm: Assume that by and large, read/write data is only required by a single entity (the major consumers of large multiply shared mappings are libraries, which are read only) and optimise the I/O path for this case. This means that any other shared consumers of the data (including the kernel) must separately map it themselves. The DMA API would be changed to perform coherence to the preferred address space (which could be the kernel). This is a slight paradigm shift, because now devices that need to peek at the data may have to map it first. Further, to free up more space for this mapping, we would break the assumption that any page in ZONE_NORMAL is automatically mapped into kernel space.

The benefits are that I/O goes straight from the device into the user space (for processors that have virtually indexed caches) and the kernel has quite a large unmapped area for use in kmapping highmem pages (for x86).

1 Introduction

In the Linux kernel[1] there are two addressing spaces: memory physical which is the location in the actual memory subsystem and CPU virtual, which is an address the CPU's Memory Management Unit (MMU) translates to a memory physical address internally. The Linux kernel operates completely in CPU virtual space, keeping separate virtual spaces for the kernel and each of the current user processes. However, the kernel also has to manage the mappings between physical and virtual spaces, and to do that it keeps track of where the physical pages of memory currently are.

In the Linux kernel, memory is split into zones in memory physical space:

- `ZONE_DMA`: A historical region where ISA DMAable memory is allocated from. On x86 this is all memory under 16MB.

[1] This is not quite true, there are kernels for processors without memory management units, but these are very specialised and won't be considered further

- ZONE_NORMAL: This is where normally allocated kernel memory goes. Where this zone ends depends on the architecture. However, all memory in this zone is mapped in kernel space (visible to the kernel).
- ZONE_HIGHMEM: This is where the rest of the memory goes. It's characteristic is that it is not mapped in kernel space (thus the kernel cannot access it without first mapping it).

1.1 The x86 and Highmem

The main reason for the existence of ZONE_HIGHMEM is a peculiar quirk on the x86 processor which makes it rather expensive to have different page table mappings between the kernel and user space. The root of the problem is that the x86 can only keep one set of physical to virtual mappings on-hand at once. Since the kernel and the processes occupy different virtual mappings, the TLB context would have to be switched not only when the processor changes current user tasks, but also when the current user task calls on the kernel to perform an operation on its behalf. The time taken to change mappings, called the TLB flushing penalty, contributes to a degradation in process performance and has been measured at around 30%[1]. To avoid this penalty, the Kernel and user spaces share a partitioned virtual address space so that the kernel is actually mapped into user space (although protected from user access) and vice versa.

The upshot of this is that the x86 userspace is divided 3GB/1GB with the virtual address range 0x00000000-0xbfffffff being available for the user process and 0xc0000000-0xffffffff being reserved for the kernel.

The problem, for the kernel, is that it now only has 1GB of virtual address to play with *including* all memory mapped I/O regions. The result being that ZONE_NORMAL actually ends at around 850kb on most x86 boxes. Since the kernel must also manage the mappings for every user process (and these mappings must be memory resident), the larger the physical memory of the kernel becomes, the less of ZONE_NORMAL becomes available to the kernel. On a 64GB x86 box, the usable memory becomes minuscule and has lead to the proposal[2] to use a 4G/4G split and just accept the TLB flushing penalty.

1.2 Non-x86 and Virtual Indexing

Most other architectures are rather better implemented and are able to cope easily with separate virtual spaces for the user and the kernel without imposing a performance penalty transitioning from one virtual address space to another. However, there are other problems the kernel's penchant for keeping all memory mapped causes, notably with Virtual Indexing.

Virtual Indexing[3] (VI) means that the CPU cache keeps its data indexed by virtual address (rather than by physical address like the x86 does). The problem this causes is that if multiple virtual address spaces have the same physical address mapped, but at different virtual addresses then the cache may contain duplicate entries, called aliases. Managing these aliases becomes impossible if there are multiple ones that become dirty.

Most VI architectures find a solution to the multiple cache line problem by having a "congruence modulus" meaning that if two virtual addresses are equal modulo this congruence (usually a value around 4MB) then the cache will detect the aliasing and keep only a single copy of the data that will be seen by all the virtual addresses.

The problems arise because, although architectures go to great lengths to make sure all user mappings are congruent, because the kernel memory is always mapped, it is highly unlikely that any given kernel page would be congruent to a user page.

1.3 The solution: Unmapping ZONE_NORMAL

It has already been pointed out[4] that x86 could recover some of its precious ZONE_NORMAL space simply by moving page table entries into unmapped highmem space. However, the penalty of having to map and unmap the page table entries to modify them turned out to be unacceptable.

The solution, though, remains valid. There are many pages of data currently in ZONE_NORMAL that the kernel doesn't ordinarily use. If these could be unmapped and their virtual address space given up then the x86 kernel wouldn't be facing quite such a memory crunch.

For VI architectures, the problems stem from having unallocated kernel memory already mapped. If we could keep the majority of kernel memory unmapped, and map it only when we really need to use it, then we would stand a very good chance of being able to map the memory congruently even in kernel space.

The solution this paper will explore is that of keeping the majority of kernel memory unmapped, mapping it only when it is used.

2 A closer look at Virtual Indexing

As well as the aliasing problem, VI architectures also have issues with I/O coherency on DMA. The essence of the problem stems from the fact that in order to make a device access to physical memory coherent, any cache lines that the processor is holding need to be flushed/invalidates as part of the DMA transaction. In order to do DMA, a device simply presents a physical address to the system with a request to read or write. However, if the processor indexes the caches virtually, it will have no idea whether it is caching this physical address or not. Therefore, in order to give the processor an idea of where in the cache the data might be, the DMA engines on VI architectures also present a virtual index (called the "coherence index") along with the physical address.

2.1 Coherence Indices and DMA

The Coherence Index is computed by the processor on a per page basis, and is used to identify the line in the cache belonging to the physical address the DMA is using.

One will notice that this means the coherence index must be computed on *every* DMA transaction for a *particular* address space (although, if all the addresses are congruent, one may simply pick any one). Since, at the time the dma mapping is done, the only virtual address the kernel knows about is the kernel virtual address, it means that DMA is always done coherently with the kernel.

In turn, since the kernel address is pretty much not congruent with any user address, before the DMA is signalled as being completed to the user process, the kernel mapping and the user mappings must likewise be made coherent (using the `flush_dcache_page()` function). However, since the majority of DMA transactions occur on *user* data in which the kernel has no interest, the extra flush is simply an unnecessary performance penalty.

This performance penalty would be eliminated if either we knew that the designated kernel ad-

dress was congruent to all the user addresses or we didn't bother to map the DMA region into kernel space and simply computed the coherence index from a given user process. The latter would be preferable from a performance point of view since it eliminates an unnecessary map and unmap.

2.2 Other Issues with Non-Congruence

On the parisc architecture, there is an architectural requirement that we don't simultaneously enable multiple read and write translations of a non-congruent address. We can either enable a single write translation or multiple read (but no write) translations. With the current manner of kernel operation, this is almost impossible to satisfy without going to enormous lengths in our page translation and fault routines to work around the issues.

Previously, we were able to get away with ignoring this restriction because the machine would only detect it if we allowed multiple aliases to become dirty (something Linux never does). However, in the next generation systems, this condition will be detected when it occurs. Thus, addressing it has become critical to providing a bootable kernel on these new machines.

Thus, as well as being a simple performance enhancement, removing non-congruence becomes vital to keeping the kernel booting on next generation machines.

2.3 VIPT vs VIVT

This topic is covered comprehensively in [3]. However, there is a problem in VIPT caches, namely that if we are reusing the virtual address in kernel space, we must flush the processor's cache for that page on this re-use otherwise it may fall victim to stale cache references that were left over from a prior use.

Flushing a VIPT cache is easier said than done, since in order to flush, a valid translation must exist for the virtual address in order for the flush to be effective. This causes particular problems for pages that were mapped to a user space process, since the address translations are destroyed *before* the page is finally freed.

3 Kernel Virtual Space

Although the kernel is nominally mapped in the same way the user process is (and can theoretically be fragmented in physical space), in fact it is usually offset mapped. This means there is a simple mathematical relation between the physical and virtual addresses:

$$virtual = physical + \texttt{\_\_PAGE\_OFFSET}$$

where `__PAGE_OFFSET` is an architecture defined quantity. This type of mapping makes it very easy to calculate virtual addresses from physical ones and vice versa without having to go to all the bother (and CPU time) of having to look them up in the kernel page tables.

3.1 Moving away from Offset Mapping

There's another wrinkle on some architectures in that if an interruption occurs, the CPU turns off virtual addressing to begin processing it. This means that the kernel needs to save the various registers and turn virtual addressing back on, all in physical space. If it's no longer a simple matter of subtracting `__PAGE_OFFSET` to get the kernel stack for the process, then extra time will be consumed in the critical path doing potentially cache cold page table lookups.

3.2 Keeping track of Mapped pages

In general, when mapping a page we will either require that it goes in the first available slot (for x86), or that it goes at the first available slot congruent with a given address (for VI architectures). All we really require is a simple mechanism for finding the first free page virtual address given some specific constraints. However, since the constraints are architecture specific, the specifics of this tracking are also implemented in architectures (see section 5.2 for details on parisc).

3.3 Determining Physical address from Virtual and Vice-Versa

In the Linux kernel, the simple macros `__pa()` and `__va()` are used to do physical to virtual translation. Since we are now filling the mappings in randomly, this is no longer a simple offset calculation.

The kernel does have help for finding the virtual address of a given page. There is an optional `virtual` entry which is turned on and populated with the page's current virtual address when the architecture defines `WANT_PAGE_VIRTUAL`. The `__va()` macro can be programmed simply to do this lookup.

To find the physical address, the best method is probably to look the page up in the kernel page table mappings. This is obviously less efficient than a simple subtraction.

4 Implementing the unmapping of `ZONE_NORMAL`

It is not surprising, given that the entire kernel is designed to operate with `ZONE_NORMAL` mapped it is surprising that unmapping it turns out to be fairly easy. The primary reason for this is the existence of highmem. Since pages in `ZONE_HIGHMEM` are always unmapped and since they are usually assigned to user processes, the kernel must proceed on the assumption that it potentially has to map into its address space any page from a user process that it wishes to touch.

4.1 Booting

The kernel has an entire bootmem API whose sole job is to cope with memory allocations while the system is booting and before paging has been initialised to the point where normal memory allocations may proceed. On parisc, we simply get the available page ranges from the firmware, map them all and turn them over lock stock and barrel to bootmem.

Then, when we're ready to begin paging, we simply release all the unallocated bootmem pages for the kernel to use from its `mem_map`[2] array of pages.

We can implement the unmapping idea simply by covering all our page ranges with an offset map for bootmem, but then unmapping all the unreserved pages that bootmem releases to the `mem_map` array.

This leaves us with the kernel text and data sections contiguously offset mapped, and all other boot time

4.2 Pages Coming From User Space

The standard mechanisms for mapping potential highmem pages from user space for the kernel to see are

[2] This global array would be a set of per zone arrays on NUMA

`kmap`, `kunmap`, `kmap_atomic` and `kmap_atomic_to_page`. Simply hijacking them and divorcing their implementation from `CONFIG_HIGHMEM` is sufficient to solve all user to kernel problems that arise because of the unmapping of `ZONE_NORMAL`.

4.3 In Kernel Problems: Memory Allocation

Since now every free page in the system will be unmapped, they will have to be mapped before the *kernel* can use them (pages allocated for use in user space have no need to be mapped additionally in kernel space at allocation time). The engine for doing this is a single point in `__alloc_pages()` which is the central routine for allocating every page in the system. In the single successful page return, the page is mapped for the kernel to use it if `__GFP_HIGH` is not set—this simple test is sufficient to ensure that kernel pages only are mapped here.

The unmapping is done in two separate routines: `__free_pages_ok()` for freeing bulk pages (accumulations of contiguous pages) and `free_hot_cold_page()` for freeing single pages. Here, since we don't know the gfp mask the page was allocated with, we simply check to see if the page is currently mapped, and unmap it if it is before freeing it. There is another side benefit to this: the routine that transfers all the unreserved bootmem to the `mem_map` array does this via `__free_pages()`. Thus, we additionally achieve the unmapping of all the free pages in the system after booting with virtually no additional effort.

4.4 Other Benefits: Variable size pages

Although it wasn't the design of this structure to provide variable size pages, one of the benefits of this approach is now that the pages that are mapped as they are allocated. Since pages in the kernel are allocated with a specified order (the power of two of the number of contiguous pages), it becomes possible to cover them with a TLB entry that is larger than the usual page size (as long as the architecture supports this). Thus, we can take the `order` argument to `__alloc_pages()` and work out the smallest number of TLB entries that we need to allocate to cover it.

Implementation of variable size pages is actually transparent to the system; as far as Linux is concerned, the page table entries it deal with describe 4k pages. However, we add additional flags to the pte to tell the software TLB routine that actually we'd like to use a larger size TLB to access this region.

As a further optimisation, in the architecture specific routines that free the boot mem, we can remap the kernel text and data sections with the smallest number of TLB entries that will entirely cover each of them.

5 Achieving The VI architecture Goal: Fully Congruent Aliasing

The system possesses every attribute it now needs to implement this. We no-longer map any user pages into kernel space unless the kernel actually needs to touch them. Thus, the pages will have congruent user addresses allocated to them in user space *before* we try to map them in kernel space. Thus, all we have to do is track up the free address list in increments of the congruence modulus until we find an empty place to map the page congruently.

5.1 Wrinkles in the I/O Subsystem

The I/O subsystem is designed to operate without mapping pages into the kernel *at all*. This

becomes problematic for VI architectures because we have to know the user virtual address to compute the coherence index for the I/O. If the page is unmapped in kernel space, we can no longer make it coherent with the kernel mapping and, unfortunately, the information in the BIO is insufficient to tell us the user virtual address.

The proposal for solving this is to add an architecture defined set of elements to `struct bio_vec` and an architecture specific function for populating this (possibly empty) set of elements as the biovec is created. In parisc, we need to add an extra unsigned long for the coherence index, which we compute from a pointer to the mm and the user virtual address. The architecture defined components are pulled into `struct scatterlist` by yet another callout when the request is mapped for DMA.

5.2 Tracking the Mappings in `ZONE_DMA`

Since the tracking requirements vary depending on architectures: x86 will merely wish to find the first free pte to place a page into; however VI architectures will need to find the first free pte satisfying the congruence requirements (which vary by architecture), the actual mechanism for finding a free pte for the mapping needs to be architecture specific.

On parisc, all of this can be done in `kmap_kernel()` which merely uses rmap[5] to determine if the page is mapped in user space and find the congruent address if it is. We use a simple hash table based bitmap with one bucket representing the set of available congruent pages. Thus, finding a page congruent to any given virtual address is the simple computation of finding the first set bit in the congruence bucket. To find an arbitrary page, we keep a global bucket counter, allocating a page from that bucket and then incrementing the counter[3].

6 Implementation Details on PA-RISC

Since the whole thrust of this project was to improve the kernel on PA-RISC (and bring it back into architectural compliance), it is appropriate to investigate some of the other problems that turned up during the implementation

6.1 Equivalent Mapping

The PA architecture has a software TLB meaning that in Virtual mode, if the CPU accesses an address that isn't in the CPU's TLB cache, it will take a TLB fault so the software routine can locate the TLB entry (by walking the page tables) and insert it into the CPU's TLB. Obviously, this type of interruption must be handled purely by referencing physical addresses. In fact, the PA CPU is designed to have fast and slow paths for faults and interruptions. The fast paths (since they cannot take another interruption, i.e. not a TLB miss fault) must all operate on physical addresses. To assist with this, the PA CPU even turns off virtual addressing when it takes an interruption.

When the CPU turns off virtual address translation, it is said to be operating in absolute mode. All address accesses in this mode are physical. However, all accesses in this mode also go through the CPU cache (which means that for this particular mode the cache is actually Physically Indexed). Unfortunately, this can also set up unwanted aliasing between the

[3]This can all be done locklessly with atomic increments, since it doesn't really matter if we get two allocations from the same bucket because of race conditions

physical address and its virtual translation. The fix for this is to obey the architectural definition for "equivalent mapping". Equivalent mapping is defined as virtual and physical addresses being equal; however, we benefit from the obvious loophole in that the physical and virtual addresses don't have to be exactly equal, merely equal modulo the congruent modulus.

All of this means that when a page is allocated for use by the kernel, we must determine if it will ever be used in absolute mode, and make it equivalently mapped if it will be. At the time of writing, this was simply implemented by making all kernel allocated pages equivalent. However, really all that needs to be equivalently mapped is

1. the page tables (pgd, pmd and pte),
2. the task structure and
3. the kernel stacks.

6.2 Physical to Virtual address Translation

In the interruption slow path, where we save all the registers and transition to virtual mode, there is a point where execution must be switched (and hence pointers moved from physical to virtual). Currently, with offset mapping, this is simply done by and addition of `__PAGE_OFFSET`. However, in the new scheme we cannot do this, nor can we call the address translation functions when in absolute mode. Therefore, we had to reorganise the interruption paths in the PA code so that both the physical and virtual address was available. Currently parisc uses a control register (`%cr30`) to store the virtual address of the `struct thread_info`. We altered all paths to change `%cr30` to contain the physical address of `struct thread_info` and also added a physical address pointer to the `struct task_struct` to the thread info. This is sufficient to perform all the necessary register saves in absolute addressing mode.

6.3 Flushing on Page Freeing

as was documented in section **??**, we need to find a way of flushing a user virtual address *after* it's translation is gone. Actually, this turns out to be quite easy on PARISC. We already have an area of memory (called the tmpalias space) that we use to copy to priming the user cache (it is simply a 4MB memory area we dynamically program to map to the page). Therefore, as long as we know the user virtual address, we can simply flush the page through the tmpalias space. In order to confound any attempted kernel use of this page, we reserve a separate 4MB virtual area that produces a page fault if referenced, and point the page's `virtual` address into this when it is *removed* from process mappings (so that any kernel attempt to use the page produces an immediate fault). Then, when the page is freed, if it's `virtual` pointer is within this range, we convert it to a tmpalias address and flush it using the tmpalias mechanism.

7 Results and Conclusion

The best result is that on a parisc machine, the total amount of memory the operational kernel keeps mapped is around 10MB (although this alters depending on conditions).

The current implementation makes all pages congruent or equivalent, but the allocation routine contains `BUG_ON()` asserts to detect if we run out of equivalent addresses. So far, under fairly heavy stress, none of these has tripped.

Although the primary reason for the unmapping was to move parisc back within its architectural requirements, it also produces a knock on effect of speeding up I/O by eliminating the cache flushing from kernel to user space. At the time of writing, the effects of this were still unmeasured, but expected to be around 6% or so.

As a final side effect, the flush on free necessity releases the parisc from a very stringent "flush the entire cache on process death or exec" requirement that was producing horrible latencies in the parisc fork/exec. With this code in place, we see a vast (50%) improvement in the fork/exec figures.

References

[1] Andrea Arcangeli *3:1 4:4 100HZ 1000HZ comparison with the HINT benchmark* 7 April 2004 http://www.kernel.org/pub/linux/kernel/people/andrea/misc/31-44-100-1000/31-44-100-1000.html

[2] Ingo Molnar *[announce, patch] 4G/4G split on x86, 64 GB RAM (and more) support* 8 July 2003 http://marc.theaimsgroup.com/?t=105770467300001

[3] James E.J. Bottomley *Understanding Caching* Linux Journal January 2004, Issue 117 p58

[4] Ingo Molnar *[patch] simpler 'high-pte' design* 18 February 2002 http://marc.theaimsgroup.com/?l=linux-kernel&m=101406121032371

[5] Rick Van Riel *Re: Rmap code?* 22 August 2001 http://marc.theaimsgroup.com/?l=linux-mm&m=99849912207578