Cache and TLB Flushing
Under Linux

David S. Miller <davem@redhat.com>

This document describes the cache/tlb flushing interfaces called by the Linux VM subsystem. It enumerates over each interface, describes its intended purpose, and what side effect is expected after the interface is invoked.

The side effects described below are stated for a uniprocessor implementation, and what is to happen on that single processor. The SMP cases are a simple extension, in that you just extend the definition such that the side effect for a particular interface occurs on all processors in the system. Don't let this scare you into thinking SMP cache/tlb flushing must be so inefficient, this is in fact an area where many optimizations are possible. For example, if it can be proven that a user address space has never executed on a cpu (see vma->cpu_vm_mask), one need not perform a flush for this address space on that cpu.

First, the TLB flushing interfaces, since they are the simplest. The "TLB" is abstracted under Linux as something the cpu uses to cache virtual-->physical address translations obtained from the software page tables. Meaning that if the software page tables change, it is possible for stale translations to exist in this "TLB" cache. Therefore when software page table changes occur, the kernel will invoke one of the following flush methods _after_ the page table changes occur:

1) void flush_tlb_all(void)

The most severe flush of all. After this interface runs, any previous page table modification whatsoever will be visible to the cpu.

This is usually invoked when the kernel page tables are changed, since such translations are "global" in nature.

2) void flush_tlb_mm(struct mm_struct *mm)

This interface flushes an entire user address space from the TLB. After running, this interface must make sure that any previous page table modifications for the address space 'mm' will be visible to the cpu. That is, after running, there will be no entries in the TLB for 'mm'.

This interface is used to handle whole address space page table operations such as what happens during fork, and exec.

3) void flush_tlb_range(struct vm_area_struct *vma,
unsigned long start, unsigned long end)

Here we are flushing a specific range of (user) virtual address translations from the TLB. After running, this interface must make sure that any previous page table modifications for the address space 'vma->vm_mm' in the range 'start' to 'end-1' will be visible to the cpu. That is, after running, here will be no entries in the TLB for 'mm' for virtual addresses in the range 'start' to 'end-1'.

The "vma" is the backing store being used for the region. Primarily, this is used for munmap() type operations.

The interface is provided in hopes that the port can find a suitably efficient method for removing multiple page sized translations from the TLB, instead of having the kernel call flush_tlb_page (see below) for each entry which may be modified.

4) void flush_tlb_page(struct vm_area_struct *vma, unsigned long addr)

This time we need to remove the PAGE_SIZE sized translation from the TLB. The 'vma' is the backing structure used by Linux to keep track of mmap'd regions for a process, the address space is available via vma->vm_mm. Also, one may test (vma->vm_flags & VM_EXEC) to see if this region is executable (and thus could be in the 'instruction TLB' in split-tlb type setups).

After running, this interface must make sure that any previous page table modification for address space 'vma->vm_mm' for user virtual address 'addr' will be visible to the cpu. That is, after running, there will be no entries in the TLB for 'vma->vm_mm' for virtual address 'addr'.

This is used primarily during fault processing.

```
5) void update_mmu_cache(struct vm_area_struct *vma,
                         unsigned long address, pte_t *ptep)
```

At the end of every page fault, this routine is invoked to tell the architecture specific code that a translation now exists at virtual address "address" for address space "vma->vm_mm", in the software page tables.

A port may use this information in any way it so chooses. For example, it could use this event to pre-load TLB translations for software managed TLB configurations. The sparc64 port currently does this.

6) void tlb_migrate_finish(struct mm_struct *mm)

This interface is called at the end of an explicit process migration. This interface provides a hook to allow a platform to update TLB or context-specific information for the address space.

The ia64 sn2 platform is one example of a platform that uses this interface.

Next, we have the cache flushing interfaces. In general, when Linux is changing an existing virtual-->physical mapping to a new value, the sequence will be in one of the following forms:

```
	1) flush_cache_mm(mm);
	   change_all_page_tables_of(mm);
	   flush_tlb_mm(mm);

	2) flush_cache_range(vma, start, end);
	   change_range_of_page_tables(mm, start, end);
	   flush_tlb_range(vma, start, end);

	3) flush_cache_page(vma, addr, pfn);
	   set_pte(pte_pointer, new_pte_val);
	   flush_tlb_page(vma, addr);
```

The cache level flush will always be first, because this allows us to properly handle systems whose caches are strict and require a virtual-->physical translation to exist for a virtual address when that virtual address is flushed from the cache. The HyperSparc cpu is one such cpu with this attribute.

The cache flushing routines below need only deal with cache flushing to the extent that it is necessary for a particular cpu. Mostly, these routines must be implemented for cpus which have virtually indexed caches which must be flushed when virtual-->physical translations are changed or removed. So, for example, the physically indexed physically tagged caches of IA32 processors have no need to implement these interfaces since the caches are fully synchronized and have no dependency on translation information.

Here are the routines, one by one:

1) void flush_cache_mm(struct mm_struct *mm)

	This interface flushes an entire user address space from the caches. That is, after running, there will be no cache lines associated with 'mm'.

	This interface is used to handle whole address space page table operations such as what happens during exit and exec.

2) void flush_cache_dup_mm(struct mm_struct *mm)

	This interface flushes an entire user address space from the caches. That is, after running, there will be no cache lines associated with 'mm'.

	This interface is used to handle whole address space page table operations such as what happens during fork.

	This option is separate from flush_cache_mm to allow some optimizations for VIPT caches.

3) void flush_cache_range(struct vm_area_struct *vma,
                          unsigned long start, unsigned long end)

Here we are flushing a specific range of (user) virtual addresses from the cache. After running, there will be no entries in the cache for 'vma->vm_mm' for virtual addresses in the range 'start' to 'end-1'.

The "vma" is the backing store being used for the region. Primarily, this is used for munmap() type operations.

The interface is provided in hopes that the port can find a suitably efficient method for removing multiple page sized regions from the cache, instead of having the kernel call flush_cache_page (see below) for each entry which may be modified.

4) void flush_cache_page(struct vm_area_struct *vma, unsigned long addr, unsigned long pfn)

This time we need to remove a PAGE_SIZE sized range from the cache. The 'vma' is the backing structure used by Linux to keep track of mmap'd regions for a process, the address space is available via vma->vm_mm. Also, one may test (vma->vm_flags & VM_EXEC) to see if this region is executable (and thus could be in the 'instruction cache' in "Harvard" type cache layouts).

The 'pfn' indicates the physical page frame (shift this value left by PAGE_SHIFT to get the physical address) that 'addr' translates to. It is this mapping which should be removed from the cache.

After running, there will be no entries in the cache for 'vma->vm_mm' for virtual address 'addr' which translates to 'pfn'.

This is used primarily during fault processing.

5) void flush_cache_kmaps(void)

This routine need only be implemented if the platform utilizes highmem. It will be called right before all of the kmaps are invalidated.

After running, there will be no entries in the cache for the kernel virtual address range PKMAP_ADDR(0) to PKMAP_ADDR(LAST_PKMAP).

This routing should be implemented in asm/highmem.h

6) void flush_cache_vmap(unsigned long start, unsigned long end)
   void flush_cache_vunmap(unsigned long start, unsigned long end)

Here in these two interfaces we are flushing a specific range of (kernel) virtual addresses from the cache. After running, there will be no entries in the cache for the kernel address space for virtual addresses in the range 'start' to 'end-1'.

The first of these two routines is invoked after map_vm_area() has installed the page table entries. The second is invoked before unmap_kernel_range() deletes the page table entries.

There exists another whole class of cpu cache issues which currently require a whole different set of interfaces to handle properly. The biggest problem is that of virtual aliasing in the data cache of a processor.

Is your port susceptible to virtual aliasing in its D-cache? Well, if your D-cache is virtually indexed, is larger in size than PAGE_SIZE, and does not prevent multiple cache lines for the same physical address from existing at once, you have this problem.

If your D-cache has this problem, first define asm/shmparam.h SHMLBA properly, it should essentially be the size of your virtually addressed D-cache (or if the size is variable, the largest possible size). This setting will force the SYSv IPC layer to only allow user processes to mmap shared memory at address which are a multiple of this value.

NOTE: This does not fix shared mmaps, check out the sparc64 port for one way to solve this (in particular SPARC_FLAG_MMAPSHARED).

Next, you have to solve the D-cache aliasing issue for all other cases. Please keep in mind that fact that, for a given page mapped into some user address space, there is always at least one more mapping, that of the kernel in its linear mapping starting at PAGE_OFFSET. So immediately, once the first user maps a given physical page into its address space, by implication the D-cache aliasing problem has the potential to exist since the kernel already maps this page at its virtual address.

```
  void copy_user_page(void *to, void *from, unsigned long addr, struct page *page)
  void clear_user_page(void *to, unsigned long addr, struct page *page)
```

These two routines store data in user anonymous or COW pages. It allows a port to efficiently avoid D-cache alias issues between userspace and the kernel.

For example, a port may temporarily map 'from' and 'to' to kernel virtual addresses during the copy. The virtual address for these two pages is chosen in such a way that the kernel load/store instructions happen to virtual addresses which are of the same "color" as the user mapping of the page. Sparc64 for example, uses this technique.

The 'addr' parameter tells the virtual address where the user will ultimately have this page mapped, and the 'page' parameter gives a pointer to the struct page of the target.

If D-cache aliasing is not an issue, these two routines may simply call memcpy/memset directly and do nothing more.

```
void flush_dcache_page(struct page *page)
```

Any time the kernel writes to a page cache page, _OR_ the kernel is about to read from a page cache page and user space shared/writable mappings of this page potentially exist, this routine is called.

NOTE: This routine need only be called for page cache pages which can potentially ever be mapped into the address space of a user process. So for example, VFS layer code handling vfs symlinks in the page cache need not call this interface at all.

The phrase "kernel writes to a page cache page" means, specifically, that the kernel executes store instructions that dirty data in that page at the page->virtual mapping of that page. It is important to flush here to handle D-cache aliasing, to make sure these kernel stores are visible to user space mappings of that page.

The corollary case is just as important, if there are users which have shared+writable mappings of this file, we must make sure that kernel reads of these pages will see the most recent stores done by the user.

If D-cache aliasing is not an issue, this routine may simply be defined as a nop on that architecture.

There is a bit set aside in page->flags (PG_arch_1) as "architecture private". The kernel guarantees that, for pagecache pages, it will clear this bit when such a page first enters the pagecache.

This allows these interfaces to be implemented much more efficiently. It allows one to "defer" (perhaps indefinitely) the actual flush if there are currently no user processes mapping this page. See sparc64's flush_dcache_page and update_mmu_cache implementations for an example of how to go about doing this.

The idea is, first at flush_dcache_page() time, if page->mapping->i_mmap is an empty tree and ->i_mmap_nonlinear an empty list, just mark the architecture private page flag bit. Later, in update_mmu_cache(), a check is made of this flag bit, and if set the flush is done and the flag bit is cleared.

IMPORTANT NOTE: It is often important, if you defer the flush, that the actual flush occurs on the same CPU as did the cpu stores into the page to make it dirty. Again, see sparc64 for examples of how to deal with this.

```
void copy_to_user_page(struct vm_area_struct *vma, struct page *page,
                       unsigned long user_vaddr,
                       void *dst, void *src, int len)
void copy_from_user_page(struct vm_area_struct *vma, struct page *page,
                          unsigned long user_vaddr,
                          void *dst, void *src, int len)
```

When the kernel needs to copy arbitrary data in and out of arbitrary user pages (f.e. for ptrace()) it will use these two routines.

Any necessary cache flushing or other coherency operations that need to occur should happen here. If the processor's instruction cache does not snoop cpu stores, it is very likely that you will need to flush the instruction cache for copy_to_user_page().

```
  void flush_anon_page(struct vm_area_struct *vma, struct page *page,
                       unsigned long vmaddr)
```

When the kernel needs to access the contents of an anonymous page, it calls this function (currently only get_user_pages()). Note: flush_dcache_page() deliberately doesn't work for an anonymous page. The default implementation is a nop (and should remain so for all coherent architectures). For incoherent architectures, it should flush the cache of the page at vmaddr.

```
  void flush_kernel_dcache_page(struct page *page)
```

When the kernel needs to modify a user page is has obtained with kmap, it calls this function after all modifications are complete (but before kunmapping it) to bring the underlying page up to date. It is assumed here that the user has no incoherent cached copies (i.e. the original page was obtained from a mechanism like get_user_pages()). The default implementation is a nop and should remain so on all coherent architectures. On incoherent architectures, this should flush the kernel cache for page (using page_address(page)).

```
  void flush_icache_range(unsigned long start, unsigned long end)
```

When the kernel stores into addresses that it will execute out of (eg when loading modules), this function is called.

If the icache does not snoop stores then this routine will need to flush it.

```
  void flush_icache_page(struct vm_area_struct *vma, struct page *page)
```

All the functionality of flush_icache_page can be implemented in flush_dcache_page and update_mmu_cache. In 2.7 the hope is to remove this interface completely.

The final category of APIs is for I/O to deliberately aliased address ranges inside the kernel. Such aliases are set up by use of the vmap/vmalloc API. Since kernel I/O goes via physical pages, the I/O subsystem assumes that the user mapping and kernel offset mapping are the only aliases. This isn't true for vmap aliases, so anything in the kernel trying to do I/O to vmap areas must manually manage coherency. It must do this by flushing the vmap range before doing I/O and invalidating it after the I/O returns.

```
  void flush_kernel_vmap_range(void *vaddr, int size)
```

flushes the kernel cache for a given virtual address range in the vmap area. This is to make sure that any data the kernel modified in the vmap range is made visible to the physical page. The design is to make this area safe to perform I/O on. Note that this API does *not* also flush the offset map alias of the area.

void invalidate_kernel_vmap_range(void *vaddr, int size) invalidates the cache for a given virtual address range in the vmap area which prevents the processor from making the cache stale by speculatively reading data while the I/O was occurring to the physical pages. This is only necessary for data reads into the vmap area.