Improving Kernel Performance by Unmapping the Page Cache

James Bottomley

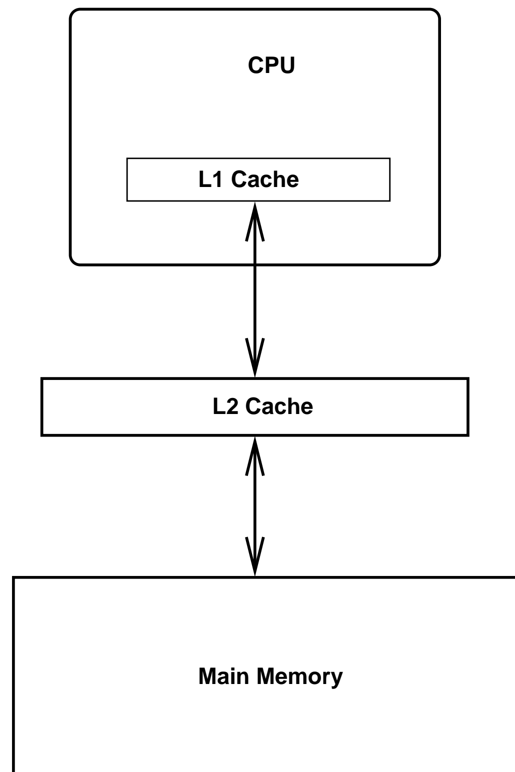
SteelEye Technology

21 July 2004

Caches and Their Problems

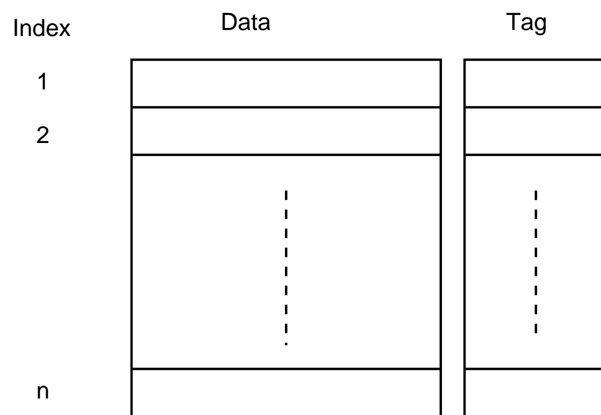
- In any computer system, speed of execution depends on how fast data and instructions can be fetched into the CPU.
- These days, the CPU clock speed is much faster than the main memory can retrieve data, so caching is essential.
- A cache (in this sense) is a fixed size area of fast memory either within or close to the CPU where known values in main memory can be stored for later fast retrieval.
- A writeback cache also takes care of flushing data out to main memory after a CPU write.

Typical Cache Layout



- Diagram shows two levels of cache.
- L1 is internal to the CPU and L2 is external.
- Caches may be either exclusive (data appears in one cache only)
- or inclusive.

Cache Layout



- Cache is composed of three elements
  - Index (corresponding to location in the cache)
  - Actual Data cached.
  - Tag (additional information that ensures what's in the cache is the data you were looking for)

Cache Types

- Physically Indexed, Physically Tagged (PIPT)

  phyical address: | tag | index | |

- Virtually Indexed, Physically Tagged (VIPT)

  Physical Address: | tag | index_1 | |

  Virtual Address: | | index_0 | index_1 | |

- Virtually Indexed, Virtually Tagged (VIVT)

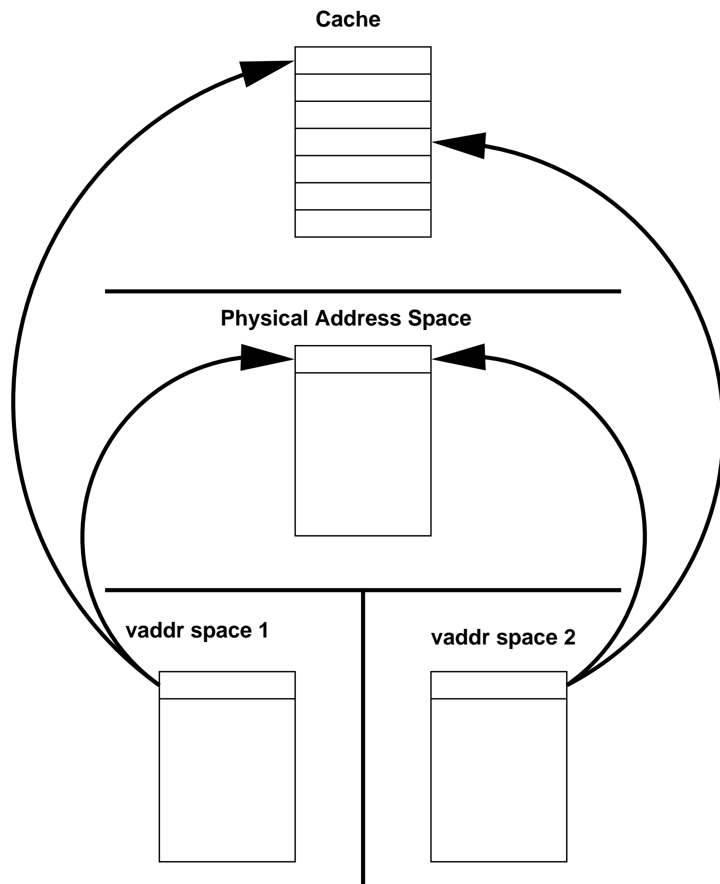
  Process id: | tag_1 |

  Virtual Address: | tag_0 | index | |

Cache Line Aliasing

- Any time the same main memory data appears more than once in the cache, aliasing is said to have occurred.
- This happens because the same physical page is mapped into more than one virtual address space.
- for VIVT caches, aliasing is impossible to prevent
- for VIPT caches, aliasing may be prevented if the virtual index of the page in the different address spaces is the same
  - The stride by which two addresses in any virtual space have the same index is called the congruence modulus.

Aliasing Illustration



- Same page is mapped into two different locations in two different process address spaces
- The virtual indices of the two virtual addresses are different.
- Every byte in the page thus appears in two separate cache lines.

The Problem of Aliasing

- Whenever the cache contains aliases, it basically means there are *incoherent* copies of identical data.
- The incoherency is a property of the caching architecture.
  - PIPT—No aliasing.
  - VIPT—Avoidable aliasing
  - VIVT—Unavoidable aliasing
- If the cache is write back, can get into a situation where two lines representing the *same* data are both dirty
  - This is absolutely fatal
- Managing the incoherency caused by aliasing is the responsibility of the Operating System.

DMA and Virtual Indexing

- DMA is Direct Memory Access.
- This means direct to *physical* memory address
- In PIPT, DMA can participate directly in the caching process by simply ejecting lines that DMA is done to.
- In VI architectures, can't do this because you don't know what the virtual index is for a given physical address.
- Have to program a *Coherence Index* as part of DMA.
- This coherence index can only name *one* address space (hence only one of the aliases).

Physical Addressing

- In a Virtually Indexed cache any access via a physical (also called absolute) needs to be coherent
- Most CPUs have non-virtual address lookup requirements (usually in the paging subsystem).
- Most caches work around this by treating the physical address identically to the way it treats virtual addresses for caching purposes.
- however, now means that you can get aliasing between physical and virtual addresses.

Aliasing in Linux

- The kernel *expects* operate in the presence of aliasing.
- There is a complete kernel API for reconciling the aliases between the various address spaces.
- However look at the operation of this API:
  - Device does DMA which is made coherent in kernel space
  - Kernel flushes the aliases to make the DMA coherent to a user process.
  - *every* DMA must be flushed this way, which is extremely inefficient

Aliasing in Linux

- The double flush in Virtually Indexed architectures is expensive and shows up as degredataion of I/O throughput.
- Eliminating this would provide a significant speed up.
- There's another problem: Some VIPT architectures *require* the elimination of aliasing.
  - We have a few parisc chipsets that require this
  - the most current example is pa8800 which currently can boot linux but not run for any length of time without crashing.

Kernel Virtual Addressing

- In almost every architecture today, the kernel is *offset mapped*
- That means that virtual and physical addresses are related by simple addition:

  virtual = physical + `__PAGE_OFFSET`.
- gives automatic resolution of virtual to physical aliases in VIPT systems.
- Makes it very easy to do address conversion
  - this is required to move from absolute to virtual addressing.
  - which is necessary for interrupt paths on parisc.

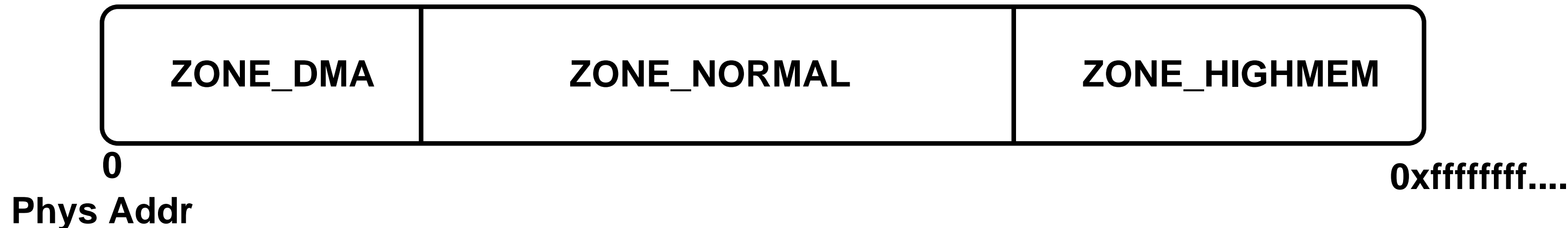
Achieving the Elimination of Aliasing

- To eliminate aliasing completely, all of the addresses must be equal modulo the congruence modulus.
- Cannot do this without breaking offset mapping in the kernel.
- This is not a simple exercise!
- So, let's explore this.

Users and `mmap()`

- Linux currently contains a hook to allow architecture code to intercept and rearrange user process vm areas.
- We use this in parisc to ensure all of our user vm areas begin on the congruence offset boundary
- This means that we can ensure that every *user* mapped area *never* has aliasing problems with other user mapped areas.
- Thus, our only problem is the kernel. If we can make kernel addresses congruent (non-aliased) with any user address, the whole system will have completely eliminated aliases.

Zones and the Kernel



- `ZONE_DMA` is historical
- `ZONE_NORMAL` is ordinarily mapped into kernel space, and is where all usual kernel allocations come from
- `ZONE_HIGHMEM` is not ordinarily mapped into kernel space (must use `kmap()` to access it) and is where all user process allocations come from.

Kernel Virtual Map

- Most of this is taken up with offset mapping of physical memory
- However, there is also a region set aside for `vmalloc()` and `kmap()`.
- This region is *fixed* size; thus the kernel can easily run out of kmap space (can be a problem on x86).
- Obviously, if you are a Virtually Indexed architecture *all* your memory (including `ZONE_HIGHMEM`) must be mapped into the kernel.

Unmapping Zone Normal

- Since zone normal (and zone dma) is the only permanently mapped zone into the kernel, unmapping it will immediately make the kernel fully congruent...
- However, this would also mean the kernel had no currently mapped memory.
- So the trick is to map the memory on allocation
- and unmap it again on release.
- doing this, a standard parisc linux has around 10MB all told mapped in zone normal.

Problems with Virtual to Physical translation

- Once we no longer use offset mapping, virtual to physical address translation becomes difficult.
- to find the virtual address from the physical address, we can use the `virtual` field of the page structure (not present unless `WANT_PAGE_VIRTUAL` is defined).
- to go from virtual to physical, we have to do page table lookups (expensive).

Begin with Bootmem

- In current Linux (on parisc), we populate the bootmem with every piece of system memory we can find.
- After the system has come up far enough to initialise memory management, we pass all the unused memory to the usual memory management system via `free_pages()`
- Thus, we can begin life as a completely offset mapped system
- and then we release the offset mappings as memory comes back in via `free_pages()`.

Kernel Allocations

- Any kernel allocations come via `kmalloc()`.
- They can never be `__GFP_HIGHMEM`
- Can be used in structures (like task struct or pmd/pgd/pte) that may be accessed via physical address.
- Thus, need to be congruent to *physical address*.

User Allocations

- Always allocated from `__GFP_HIGHMEM`
- will automatically be placed congruently to other user processes.
- will not be accessed at all by the kernel without using `kmap()`.
- Thus, no need to map into the kernel until `kmap()` at which point the mapping must be congruent to the user address.
- Unfortunately, I/O and memory freeing disrupts this.

Allocations

- All allocations (both user and kernel) come in through a single entry point:
  - `__alloc_pages()`
- Thus we only need a single hook in this routine to do the map on allocation for the kernel
- can tell exactly from the gfp flags whether this is an allocation for userspace or the kernel.
- for userspace, do not know user address when page is allocated, only when it is put into the user vma, so need an additional hook.

Determining User Addresses

- Sounds easy, but if a given user page has never been `kmap()`'d it will never have a value placed into `page->virtual`.
- Problem, because if we free it or do I/O to it, we don't know what the associated virtual address should be.
- This usually isn't known when the page is allocated, so how do we find it?
- Turns out we need to hijack a NUMA hook: `alloc_page_vma()` which allocates a page specifially to be placed at a user address.

Freeing Memory

- Should be simple ...
- But ... In VIPT, if you free a page but do not flush it from the cache, if we map the same page at a congruent address at a later time, it may still have stale cache lines.
- Thus, must flush the page from the cache when it is freed.
- Problem: If this page was a user page, and never mapped into the kernel, we may not have a valid mapping when we come to flush it.
- Thus, flush it through a temporarily aliased mapping.

Hooks for Freeing Memory

- It turns out that the memory free paths are more complex than the allocation ones.
- There are two separate possible free paths
  - `__free_pages_ok()` (for bulk page freeing) and
  - `__free_hot_cold_page()` (for single page freeing).
- have to hook into both of these.

Doing I/O

- The final problem is that the linux bio/request system contains no mechanism for identifying the user process that requested the I/O.
- Indeed, each page in a bio may have come from a *different* user space.
- True solution is probably to add process tagging to each page in the bio.
- However, quick and dirty fix is to `kmap()` each page (at a congruent address) as part of the setup of the DMA operation.

Other wins: fork()/exec()

- On parisc, fork/exec has a huge cost, primarily because we have to flush the entire cache when a process dies
- We do this to ensure we don't get stale cache lines on page reuse.
- However, if we are now flushing the pages when they are freed, we no-longer need to flush the cache at all for process death.
- This is a *huge* win. Benchmarks show the fork/exec time to improve by 50%

Conserving TLB space

- Every CPU only has a limited amount of TLB space for mappings (on parisc this is 150–280 slots)
- On most CPUs (but not x86) you can conserve this space by making TLB entries cover larger spaces
  - PA has page sizes that go up in powers of 4
- since the allocation routine `__alloc_pages()` knows the order of the allocation, you can hook into this to cover the region with the largest sized TLB entries you can manage.
- Thus, we can begin using *variable sized pages*.

Conclusions

- Ease of implementation
  - Four hooks in generic code.
  - All the rest hidden inside the Architecture Specific part.
- Unintended Consequences.
  - Began by trying to speed up I/O throughput by eliminating an unnecessary page flush.
  - Provided x86 a way to enlarge kmap space
  - Also got vastly improved fork/exec speed (hence faster kernel builds)
  - And finally turns out to be an architectural requirement for HP's latest chip (pa8800).
- Current State of the Code.