Page Colouring on ARMv6 (and a bit on ARMv7)

Page colouring is a technique for allocating pages for an MMU such that the pages exist in the cache in a particular order. The technique is sometimes used as an optimization (and is not specific to ARM), but as a result of the cache architecture some ARMv6 processors actually *require* that the allocator uses some page colouring. Some ARMv7 processors also have related (though much less severe) restrictions. This article will explain why the cache architecture imposes this restriction, and what it means in practice.

Note that this restriction only very rarely needs to be considered outside of the physical memory allocator in the kernel (or other privileged code). Typical user-space code probably won't have to deal with this directly, though understanding page colouring can help to explain why some `mmap` calls work on ARMv7 but fail on ARMv6, for example.

The restriction stems from the fact that many ARMv6 processors use VIPT caches. VIPT means "virtually indexed, physically tagged". If you're not familiar with cache terminology, that probably won't mean a lot, but I will try to explain by way of example.

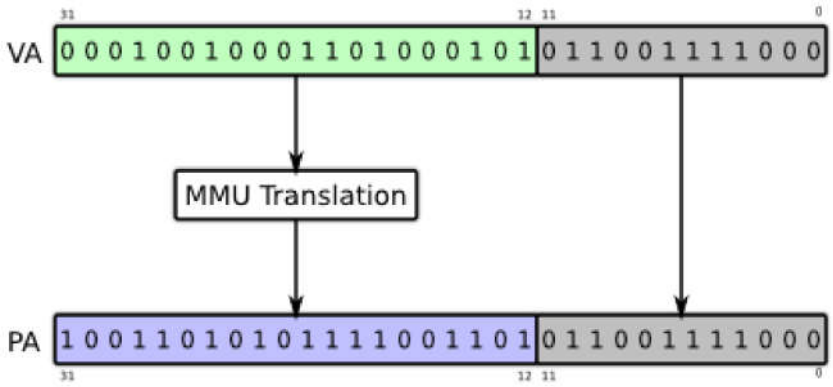
In general, ARMv7 is not affected by ARMv6's page colouring restrictions. However, ARMv7 can have VIPT instruction caches, so some restrictions can still apply, even in user space.

Virtual and Physical Addressing

Consider a case where the processor needs to make a simple memory access. The reason for the access is irrelevant; it could be the processor loading the next instruction to execute, or it could be a data transfer as a result of a load or store instruction, for example. Regardless, let's assume that the address requested is `0x12345678`. This is a *virtual address*, in that the real memory behind it might have a different *physical address*, and that the MMU (memory management unit) will need to use page tables to perform the translation.

Many operating systems use this mechanism to run multiple applications at the same time, even if the applications expect to run from the same address: each application can believe that its entry point is at `0x8000`, but the physical memory will most likely be different. This process is effectively transparent to user-space applications.

The finest granularity of memory mapping on ARMv6 MMUs is the 4KB page [1]. The bottom 12 bits of a virtual address will exactly match the bottom 12 bits of the underlying physical address. This is because 12 bits can address any offset into a 4KB page (since $2^{12} = 4096$), and pages are naturally-aligned [2] in both virtual and physical memory. As a result a virtual address translation may be viewed as shown below [3]:



Virtual to Physical address translation for 32-bit addressing with 4KB pages.

For clarity, I have shown virtual address parts in green and physical address parts in blue. The lower 12 bits (in grey) are usually not illustrated separately from the virtual and physical parts, but I have tried to make it clear that they do not change in translation as this property is relevant to page colouring. In this example, the MMU has been configured to map the virtual address `0x12345678` to the physical address `0x9abcd678`.

VIPT Cache Lookups

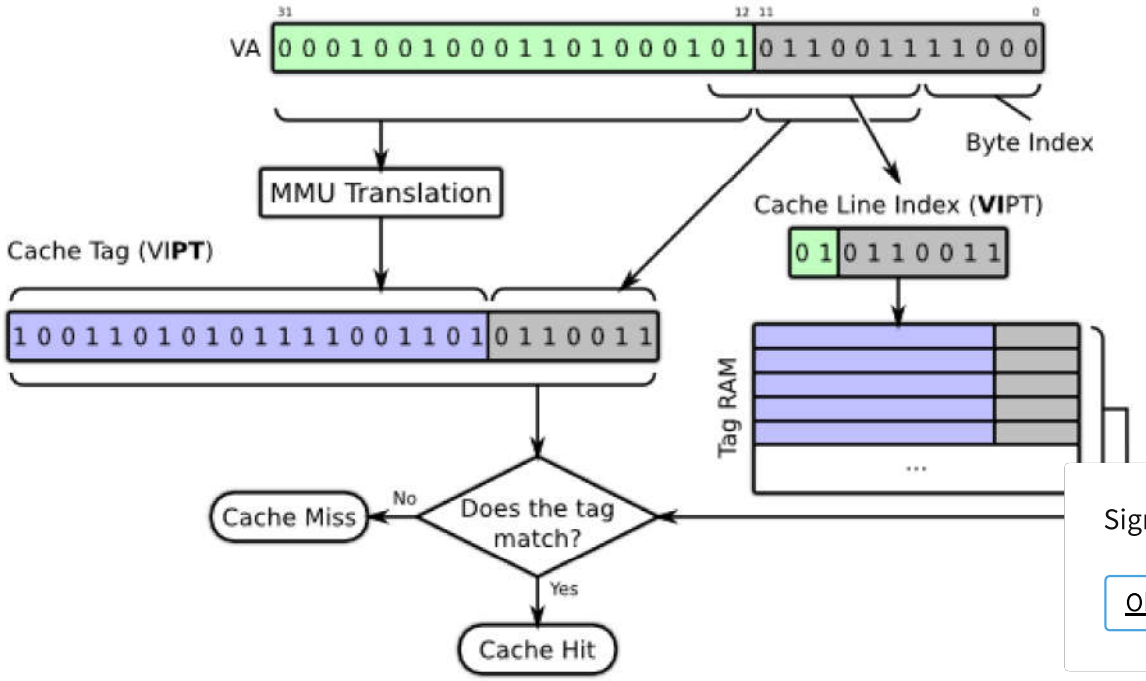
Ideally, memory accesses will result in a cache lookup, since cache accesses are much faster than memory accesses. The "VIPT" property gives a hint as to how the cache lookup works:

1. The virtual address is used to construct a cache index. This is what the "VI" from "VIPT" refers to. The cache index is an offset into a list of tags (or "tag RAM"), where each tag is associated with one cache line and indicates what memory (if any) is cached in that line. In this example, each cache line holds 32 bytes.
2. The virtual address is translated into a physical address.
3. The physical address is used to construct a cache tag. This is what the "PT" from "VIPT" refers to.

4. The cache index is used to fetch the tag from the cache, and this is compared with the cache tag computed from the physical address. If they match, the lookup is "hit" in cache, and the relevant data will be fetched from cache. Otherwise, it will be fetched from the next level of cache (or from memory).

The process as described is simplified somewhat, but it should explain enough of the process to understand page colouring.

The diagram below may make the above steps somewhat clearer:



Simplified cache lookups in a VIPT cache.

There are a few interesting points to note:

- In this example, the byte index is 5 bits long, since the cache lines in this example are 32 bytes long and 5 bits are required to address every byte in a given cache line. This is the case on ARM1136 and ARM1176, for example.
- The MMU translation can be done in parallel with the cache line index lookup.
- Two bits of the cache index are from the virtual address, but the tags that the cache index can refer to are all derived from physical addresses.

Cache Ways

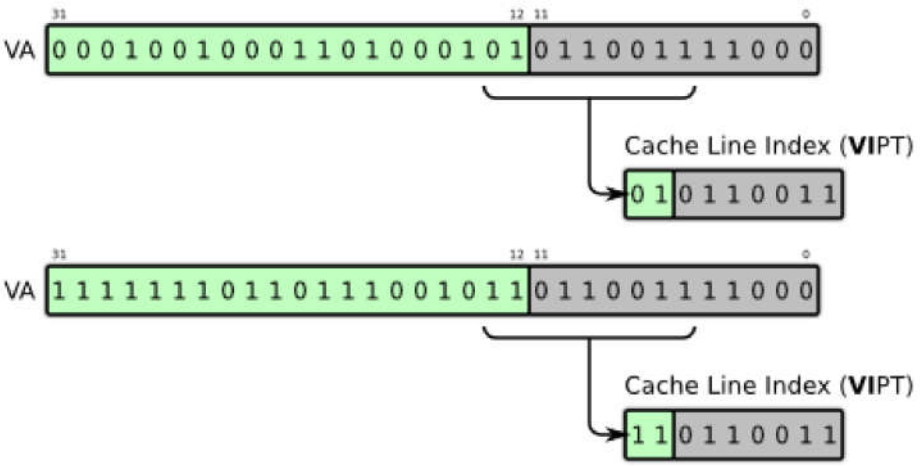
ARM1136 and ARM1176 use 4-way set-associative caches. When a memory location is stored in the cache, the index generated from its address permits it to be stored in any of the four *ways*. This mechanism is used to improve cache efficiency. I'm not going to

describe cache ways in any more depth, but Wikipedia has a good description (at the time of writing) if you're interested.

The effect that ways have on page colouring is simply that they affect the number of bits in the cache line index. In a 4-way, 64KB cache (which is the limit for ARM1136 and ARM1176), each way will have 64/4 = *16KB* of addressable space. The cache pictured above, therefore, shows a single 16KB cache way.

VIPT Cache Lookups: The Problem of Duplicate Mappings

The VIPT cache lookup given above works perfectly well if there is a one-to-one mapping between each virtual and physical address. However, this often isn't the case. Even if a specific application doesn't request a duplicate mapping, they can come about due to usage of shared memory, as well as administrative reasons in the operating system. Consider a case where the physical address `0x9abcd678` is mapped to both `0x12345678` and `0xfedcb678`:

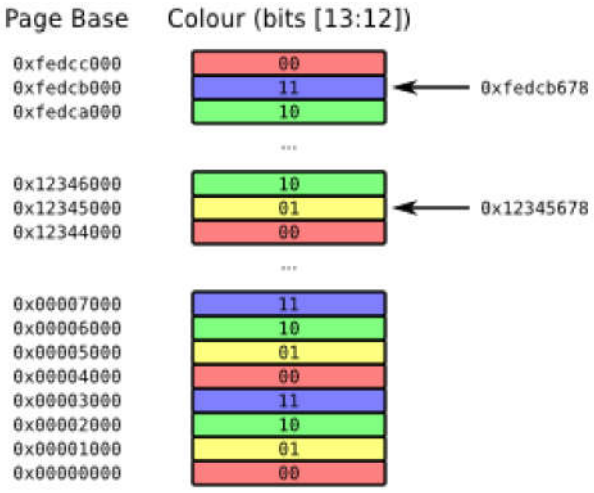


Cache line indices for `0x12345678` and `0xfedcb678`.

Both of the virtual addresses in the diagram above are configured to map onto the same physical address in our example. However, note that the cache line indices are different. It's possible, therefore, that the memory already exists in a cache line with a different virtual index. If a cache line is *dirty* (meaning that it has changes that haven't yet been written to memory), then changes made to another alias might be lost, or might mask the original. The duplicate mappings don't even need to active at the same time for this problem to occur, since some data from a previously-active process may still exist in cache after the page tables are updated. Whatever happens, it won't be good, so operating systems must enforce *page colouring restrictions* for memory allocations.

The Solution: Page Colouring Restrictions

Page colouring assigns a colour to each memory page. Colours can be assigned to both virtual and physical addresses, but it's virtual addresses that are the problem here. ("Colour" in this sense is used as an explanatory tool.) Each page in virtual memory is assigned a colour in sequence, as follows:



Page colours, showing how colours map onto bits 12 and 13 of the address. example virtual addresses are also shown to illustrate that they have differer

The actual colours chosen to represent each page are irrelevant. However, notic two example addresses have different colours. In the example system, with 4KE 16KB cache ways, bits 12 and 13 of the virtual address determine the page colour. These were the two bits used in the cache index in the VIPT lookup diagram. Note that there are four page colours because there were two bits of the virtual address in the cache index, and $2^2 = 4$.

The *page colouring restrictions* therefore require that a page allocator restricts any given physical page to a single virtual colour.

There are a couple of obvious ways to comply with ARMv6's page colouring restriction:

- Restrict any given physical page to a single virtual colour. Since ARM supports MMU granularities larger than 4KB -- all of which must be naturally aligned [2] -- the easiest way to do this is to ensure that bits 12 and 13 of a given virtual address match the same bits in the corresponding physical address.

  If only 4KB pages are used, it is possible to construct mappings which preserve the colour restrictions but where bits 12 and 13 differ between virtual and physical addresses, but it is unlikely to be worth the extra complexity.

-

> Systems with ways of 4KB or smaller do not have any page colouring restrictions, as no bits of the virtual address are present in the cache line index at that scale. Indeed, at that point, a VIPT cache requires no more software management than a PIPT cache.

Note that ARM1136 and ARM1176 both have caches that are fixed at 4 ways, so one such system with 4KB ways will have 16KB caches.

ARMv7 and Page Colouring

ARMv7 processors have PIPT data caches (or at least are required to behave as if they do). That is, they have physically indexed, physically tagged data caches, and no page colouring restrictions apply. However, they can have VIPT *instruction* caches and even VIVT instruction caches are allowed to an extent. They can also have PIPT instruction caches. (Refer to the Architecture Reference Manual for details.) Cortex-A8 and Cortex-A9 both have VIPT instruction caches.

For most use-cases, a VIPT instruction cache behaves very much like a PIPT instruction cache. Instruction caches are read-only, so the most obvious coherence issues cannot occur. However, VIPT instruction caches may still need special handling when invalidating lines by virtual address: aliased entries with different colours will not be invalidated. Actually, the Architecture Reference Manual gives the following statement:

> *The only architecturally-guaranteed way to invalidate all aliases of a physical address from a VIPT instruction cache is to invalidate the entire instruction cache.*

In practice, you should be able to simply invalidate all the possible colour variants of a given virtual address. Of course, if you don't care about *all* virtual aliases, then it doesn't matter that others might exist. Authors of self-modifying code (such as JIT compilers) can safely and efficiently work within this system by ensuring that any code they intend to execute (after modification) is invalidated using the address which is used to run it, rather than with an alias.

Note that writes to instruction data through the data cache -- for example, in self-modifying code -- always require explicit cache maintenance operations, as I described in a previous article. This is because there is no coherency between instruction and data caches in the ARM architecture.

ARMv6 User-Space Example: `mmap`

The following `mmap` sequence will typically work (under GNU/Linux) as expected on ARMv7, but fail on ARMv6:

```
// Preconditions:
//  - fd is a file descriptor as returned from open() (or a related functio
//  - The file is at least 4096 bytes long (and so can fill the mapping).
//  - The file's permissions are compatible with those in the mmap calls.
//  - MAP_FIXED is supported.
//  - The addresses provided can be mapped by the process, and are not
//    allocated or otherwise reserved or invalid.

void * buffer0 = mmap((void*)(0x12340000), 4096, PROT_READ | PROT_WRITE, MA
void * buffer1 = mmap((void*)(0xdebc1000), 4096, PROT_READ | PROT_WRITE, MA
```

Note that the first call requests a mapping to a virtual address with colour `00`, whilst the second call requests a mapping to a virtual address with colour `01`.

On ARMv6, the second `mmap` will typically fail (and return `MAP_FAILED`), but the first will pass. In some cases, the first call may also fail, for example when physical pages are already allocated to a different colour. The user process has no control over this, though it seems (experimentally, on a GNU/Linux platform) that files are mapped at 16KB alignment, so file mappings with colour `00` are reasonably reliable on ARMv6.

On ARMv7, both mappings will (in general) work, and any user-allocatable virtua can be specified provided that it is aligned at a 4KB page boundary.

Note that these `mmap` calls can't guarantee success on any platform. As an exam platform may limit the number of regions that can be mapped, and once this limit is reached `mmap` will return `MAP_FAILED` and set `errno` to `EMFILE`.

Properties of Each Caching Strategy

Why use VIPT caches if they have this inherent page colouring restriction?

The simplest (and perhaps fastest) cache lookups can be done on VIVT caches. That is, virtually indexed, virtually tagged caches. If a cache line is loaded, no MMU lookup will be needed at all. However, VIVT caches have coherency problems with duplicate mappings, and cache maintenance operations must be performed when the page tables are updated (such as on context switch). There are hardware workarounds for some of these problems, but the most efficient solution can be to simply use VIPT or PIPT caches.

VIPT caches are a natural evolution of VIVT caches. They require an MMU translation for any access, but the physical tag means that physical addresses can be uniquely identified in the cache, and so duplicate mappings are possible (within the page colouring restrictions) and no cache maintenance operations are required on context switch. Also, although an MMU

translation is still required, it can be done in parallel with the indexing operation, as shown earlier in the article.

ARMv7-A specifies that data caches behave like PIPT caches, so a given tag will never be duplicated in the tag RAM because the caches are indexed by the physical address. The disadvantage of this is that the cache lookup cannot be done in parallel with the MMU translation, since the physical address is needed for every stage of the cache lookup. With good TLB hit rates, however, this disadvantage is somewhat mitigated as a successful TLB lookup can be very fast.

Note, however, that PIPT caches could (in some situations) still benefit from page colouring as it can improve cache line eviction behaviour. The details of this would form a blog article alone, so I won't go into further detail. However, the Wikipedia article has a good description (at the time of writing).

---

[1] Versions of the ARM architecture older than ARMv6 also supported 1KB "tiny pages", but these are not available in ARMv6 or more recent architecture versions.

[2] By "natural alignment", I mean that objects are aligned to an address that is a multiple of their size. That is, 4KB pages are aligned to 4KB boundaries, and 1MB sections are aligned to 1MB boundaries (for example).

[3] In reality, the translation is likely to be performed by a TLB (translation lookaside buffer) rather than direc but the principle is the same.