CS152/252A Fall 2026

VIPT Caches & Intro to OoO

Discussion 6

1 Virtual Memory and Caches

1.1 In the caches we have studied so far, the virtual address is translated to a physical address before the cache is accessed, and the physical address supplies the tag, index, and offset. This is a physically indexed, physically tagged (PIPT) cache. Why might this be slow? How could we do better?

Every access must finish translation before the cache lookup can begin, which puts the TLB on the critical path of every hit. A TLB miss is even worse, since it requires a page table walk.

We could have at least the L1 cache operate on virtual addresses, so that a hit does not need translation at all. Translation is only needed on a miss.

1.2 Suppose we introduce a *virtual cache* that works entirely in the virtual address space and only translates on a miss. Hits are now fast compared to PIPT. What are some possible problems with this sort of cache? *Hint: what if we have multiple programs running on the same CPU?*

We no longer have isolation between programs: if `P2`'s data is in the virtual cache, `P1` can hit on it using the same virtual address.

This sort of cache is called a virtually indexed, virtually tagged (VIVT) cache, and it also causes two naming problems:

- **Homonyms**: the same virtual address in different programs can map to different physical addresses, so programs clobber each other's data.
- **Synonyms (aliasing)**: different virtual addresses can map to the same physical address, so the cache can hold multiple copies of one block and return stale values.

1.3 What are some possible fixes for these problems?

Add some sort of tag or ID to each cache line (for example, an address space ID) so that one program cannot hit on another program's lines. Alternatively, flush the cache when switching between programs, which is expensive.

2 Virtually Indexed, Physically Tagged Caches

A virtually indexed, physically tagged (VIPT) cache uses the virtual address for the index and part of the physical address as the tag, as in PIPT. We still need the physical address from the TLB or a page walk to compare the tag, but the cache lookup and translation now happen in parallel. Physical tags eliminate homonyms. Can we still have aliasing?

2.1 A direct-mapped, write-back VIPT cache uses the following address bits:

- Block offset: bits 5–0
- Index: bits **12–6** (from the VA)
- Tag: bits **31–12** (from the PA)

The page size is 4 KiB. The page table maps VPN `0x1` ⟶ PPN `0x7` and VPN `0x2` ⟶ PPN `0x7`. Memory contains `Mem[0x7040] = 5`, and the cache starts empty. Fill in the table for each access below. What value does access 4 return?

(a) Load VA `0x1040`
(b) Load VA `0x2040`
(c) Store `9` to VA `0x1040`
(d) Load VA `0x2040`

Access	VA	PA	Index	Tag	Hit/Miss	Value
1	`0x1040`	`0x7040`	65	`0x7`	Miss	5
2	`0x2040`	`0x7040`	1	`0x7`	Miss	5
3	`0x1040`	`0x7040`	65	`0x7`	Hit	9
4	`0x2040`	`0x7040`	1	`0x7`	Hit	5 (stale!)

Access 4 returns **5**, even though the program just stored 9 to the same physical address. The two virtual addresses placed the same physical block in two different sets, and the write-back store only updated the copy in set 65.

2.2 Now the cache uses these **new** bits, with the same page table, memory contents, and accesses as before:

- Block offset: bits 5–0
- Index: bits **11–6** (from the VA)
- Tag: bits **31–12** (from the PA)

Fill in the table. What value does access 4 return?

Access	VA	PA	Index	Tag	Hit/Miss	Value
1	`0x1040`	`0x7040`	1	`0x7`	Miss	5
2	`0x2040`	`0x7040`	1	`0x7`	Hit	5
3	`0x1040`	`0x7040`	1	`0x7`	Hit	9
4	`0x2040`	`0x7040`	1	`0x7`	Hit	9

Access 4 returns **9**. Both virtual addresses map to the same set, so there is only one copy of the block.

2.3 Why did the first VIPT cache design cause aliasing but not the second one? Can you generalize the rule of thumb for VIPT cache designs?

The index comes from the VA, and the tag comes from the PA. In the first design, bit 12 of the index came from the VPN. The VPN bits can differ for the same physical address, so the two virtual addresses mapped the same block to different sets, and each tag comparison happened in a different set.

The page offset bits are copied directly from the VA to the PA, so they are always the same for every virtual address that maps to a given physical address. Index bits should therefore come only from the page offset portion of the VA.

With 2^L sets, 2^b-byte blocks, and a k-bit page offset:

- $L + b \leq k$: no aliasing, since the index and offset bits are always the same.
- $L + b > k$: aliasing is possible.

2.4 A VIPT cache has the following parameters:

- Page size: 4 KB
- Block size: 64 B
- Cache size: 32 KB
- Associativity: 2-way

(a) Can this cache have aliasing?
(b) If yes, when one physical block is accessed through different virtual addresses, how many different sets could it end up in?
(c) Keeping the associativity and block size the same, what is the largest this cache can be without aliasing?
(d) Keeping the cache size at 32 KB, what associativity would remove aliasing?

The page offset is 12 bits and the block offset is 6 bits. The cache has $32\ \text{KB}\ /(64\ \text{B} \times 2) = 256$ sets, so the index is 8 bits.

(a) Yes. The block offset and index use $6 + 8 = 14$ bits, which is more than the 12-bit page offset.
(b) 2 index bits come from the VPN, so the block could end up in $2^2 = 4$ different sets.
(c) We need block offset + index ≤ 12, so the index can be at most 6 bits (64 sets). The cache can be at most $64\ \text{sets} \times 2\ \text{ways} \times 64\ \text{B} = 8\ \text{KB}$.
(d) We still need 64 sets: $32\ \text{KB}\ /(64\ \text{sets} \times 64\ \text{B}) = 8$ ways.

Because the index bits are limited to the page offset, VIPT caches cannot be very big without aliasing. This is why modern processors typically use VIPT only for the small L1 caches. Larger pages allow larger VIPT caches.

3 Intro to Out of Order Execution

Consider a small upgrade to the pipeline. We keep IF and ID, but add separate execution paths. Instructions issue in program order, and execution can overlap. Count the issue cycle as the first execution cycle.

Execution path	Operations	Latency
Path 1	Integer ALU / memory	ALU: 1 + MEM: 2
Path 2	FP add	2 cycles
Path 3	FP multiply	4 cycles

3.1 Both instructions below have ready operands. Issue `A` in cycle 1, then `B` in cycle 2. When does each instruction complete? Which instruction finishes first? Does issuing in program order guarantee completion in order?

```
A: fmul f4, f0, f2
B: fadd f8, f6, f10
```

Instruction	Issue cycle	Execution cycles	Completes
`A: fmul`	1	1, 2, 3, 4	End of 4
`B: fadd`	2	2, 3	End of 3

`B` finishes first. The program order and issue order are both `A`, `B`, but the completion order is `B`, `A`. Different latencies change the completion order, even with in order issue.

Now consider a simple pipeline with a waiting queue. Loads use the ALU and then MEM. FP operations go directly to WB.

Timing rules:

- F and D take 1 cycle each. Fetch `A` in cycle 1.
- Dispatch into the queue whenever it has room. Issue whenever operands and an execution path are ready.
- Issue at most 1 instruction per cycle.
- Forward results starting in the following cycle.
- WB accepts 1 result per cycle.
- In order: issue the next instruction, or wait.
- Out of order: issue the oldest ready instruction.

```
A: fmul f4, f0, f2
B: fadd f6, f4, f8
C: lw   x5, 0(x6)
```

3.2 Why must `B` wait? Can `C` execute during that wait?

`B` needs `f4` from `A`. `C` only uses `x6` to load from memory, so it is independent of both `A` and `B` and could execute while `B` waits.

3.3 Trace the pipeline with **in order** issue. Use `f4?` for a cycle spent waiting for `A`'s result, and `Q` for a cycle spent waiting to issue. In which cycle does `C` write back?

Cycle	1	2	3	4	5	6	7	8	9	10	11
A	F	D	Mul1	Mul2	Mul3	Mul4	WB				
B		F	D	f4?	f4?	f4?	Add1	Add2	WB		
C			F	D	Q	Q	Q	EX	Mem1	Mem2	WB

`B` waits for `f4` from `A`, and `C` waits behind `B` even though its operands are ready. `C` writes back in cycle 11.

3.4 Trace the same code with **out of order** issue. In which cycle does each instruction write back?

Cycle	1	2	3	4	5	6	7	8	9	10	11
A	F	D	Mul1	Mul2	Mul3	Mul4	WB				
B		F	D	f4?	f4?	f4?	Add1	Add2	WB		
C			F	D	EX	Mem1	Mem2	WB			

`B` still waits for `f4`, but `C` uses the ALU and MEM while `A` runs. Writeback: `A` in cycle 7, `C` in cycle 8, `B` in cycle 9.

4 Hazards and Register Renaming

For each pair below, the earlier instruction is listed first and the later instruction second.

Hazard	Earlier / later access	What must stay correct?
RAW	Write / read	Read the producer's value
WAR	Read / write	Read before a later overwrite
WAW	Write / write	Keep the later write as the final value

4.1 Find the RAW, WAR, and WAW relationships in the code below. `B` waits for `A`. Could `C` safely run before `B` reads `f3`? What would go wrong if `C` and `D` wrote `f3` in reverse order?

```
A: fmul f1, f0, f2
B: fadd f0, f3, f1
C: fmul f3, f2, f3
D: fadd f3, f3, f1
```

Type	Instruction pairs	Register
RAW	`A` to `B`, `A` to `D`	`f1`
RAW	`C` to `D`	`f3`
WAR	`A` to `B`	`f0`
WAR	`B` to `C`, `B` to `D`, `C` to `D`	`f3`
WAW	`C` to `D`	`f3`

One register name holds several values:

- `B` needs the initial value of `f3`, so `C` cannot overwrite `f3` before `B` reads it.
- `D` needs the value of `f3` produced by `C`.
- The final value of `f3` must come from `D`. If `C` and `D` wrote `f3` in reverse order, `f3` would end up holding `C`'s value.

4.2 Rewrite using free temporaries f4, f5, f6, f7. Give each destination a fresh name and update its later reads. Where do the final values of original f0, f1, and f3 now live? Which dependencies remain?

Old code snippet:

```
A: fmul f1, f0, f2
B: fadd f0, f3, f1
C: fmul f3, f2, f3
D: fadd f3, f3, f1
```

New code snippet:

```
A: fmul f4, f0, f2
B: fadd f5, f3, f4
C: fmul f6, f2, f3
D: fadd f7, f6, f4
```

The final values of the original `f1`, `f0`, and `f3` are in `f4`, `f5`, and `f7`, respectively.

The RAW dependencies remain: `A` to `B`, `A` to `D`, and `C` to `D`. All of the register WAR and WAW relationships disappear, so `C` can now execute while `B` waits for `A`.

In hardware, the processor assigns fresh physical registers and uses a rename table to track where each architectural register's current value lives. Instructions issue and complete possibly out of order, but commit their architectural effects in program order.