CSCI 564 Advanced Computer Architecture

Lecture 06: Cache Optimizations

Dr. Bo Wu (with modifications by Sumner Evans)

March 3, 2021

Colorado School of Mines

Performance Enemy Number One: Cache Misses

Know the Enemy

We can categorize all cache misses into three types.

- **Compulsory:** the program has never requested this data before. A miss is mostly unavoidable.
- **Conflict:** the program has seen this data, but it was evicted by another piece of data that mapped to the same "set".
- **Capacity:** the program is actively using more data than the cache can hold.

These are called *the three C's*.

A Simple Example

Consider a direct-mapped cache with 16 blocks, a block size of 16 bytes.

We have an application which repeats the following memory access sequence:

- 0x80000000
- 0x80000008
- 0x80000010
- 0x80000018
- 0x30000010

Cache Geometry Calculations

Index bits $= \log_2(16/1) = 4$

Offset bits $= \log_2(16) = 4$

Tag bits $= 32 - (4 + 4) = 24$

Example:

$$\texttt{0x}\underbrace{\texttt{800000}}_{\text{tag}}\underbrace{\texttt{1}}_{\text{index}}\underbrace{\texttt{0}}_{\text{offset}}$$

A Simple Example

	valid	tag	data
0	1	800000	
1	1	300000	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

`0x80000000` miss: compulsory
`0x80000008` **hit!**
`0x80000010` miss: compulsory
`0x80000018` **hit!**
`0x30000010` miss: compulsory
`0x80000000` **hit!**
`0x80000008` **hit!**
`0x80000010` miss: conflict
`0x80000018` **hit!**
`0x30000010` miss: conflict

A Simple Example: Increased Cache Line Size

Consider a direct-mapped cache with **8 blocks**, a block size of **32 bytes**.

We have an application which repeats the following memory access sequence:

- 0x80000000
- 0x80000008
- 0x80000010
- 0x80000018
- 0x30000010

Cache Geometry Calculations

Index bits $= \log_2(\mathbf{8}/1) = 3$

Offset bits $= \log_2(\mathbf{32}) = 5$

Tag bits $= 32 - (3 + 5) = 24$

Example:

$$\texttt{0x80000010} = \underbrace{\texttt{011100000000000000000000}}_{\text{tag}}\ \underbrace{\texttt{000}}_{\text{index}}\ \underbrace{\texttt{10000}}_{\text{offset}}$$

A Simple Example: Increased Cache Line Size

	valid	tag	data
0	1	300000	
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

0x80000000	miss: compulsory
0x80000008	**hit!**
0x80000010	**hit!**
0x80000018	**hit!**
0x30000010	miss: compulsory
0x80000000	miss: conflict
0x80000008	**hit!**
0x80000010	**hit!**
0x80000018	**hit!**
0x30000010	miss: conflict

A Simple Example: Increased Associativity

Consider a **2-way set-associative** cache with **8 blocks**, a block size of **32 bytes**.

We have an application which repeats the following memory access sequence:

- `0x80000000`
- `0x80000008`
- `0x80000010`
- `0x80000018`
- `0x30000010`

Cache Geometry Calculations

Index bits $= \log_2(8/\mathbf{2}) = 2$

Offset bits $= \log_2(32) = 5$

Tag bits $= 32 - (2 + 5) = 25$

Example:

$$\texttt{0x80000010} = \underbrace{\texttt{0111000000000000000000000}}_{\text{tag}}\ \underbrace{\texttt{00}}_{\text{index}}\ \underbrace{\texttt{10000}}_{\text{offset}}$$

A Simple Example: Increased Associativity

set	valid	tag	data
0	1	1000000	
	1	600000	
1			
2			
3			

0x80000000	miss: compulsory
0x80000008	**hit!**
0x80000010	**hit!**
0x80000018	**hit!**
0x30000010	miss: compulsory
0x80000000	**hit!**
0x80000008	**hit!**
0x80000010	**hit!**
0x80000018	**hit!**
0x30000010	**hit!**

Reducing Each Type of Cache Miss

Reducing Compulsory Misses: Increase Cache Line Size

The processor will request larger chunks of memory at a time.

This only works if there is good *spacial locality*, otherwise, you are bringing in data you don't need.

- If you are reading bytes effectively at random (a few bytes here, a few bytes there), this will hurt performance.
- In cases where you have sequential accesses, it will help:

```
for (int i = 0; i < 1000000; i++) {
    sum += data[i];
}
```

Reducing Compulsory Misses: Prefetching

The idea is to *speculate* on future instruction and data accesses and fetch them into cache.

- Instruction accesses are easier to predict than data accesses.

Varieties of prefetching:

- Hardware prefetching
- Software prefetching
- Mixed schemes

Reducing Compulsory Misses: Hardware Prefetching

Consider the following code:

```
for (int i = 0; i < 1000000; i++) {
    sum += data[i];
}
```

In this case, the processor could identify the pattern and proactively prefetch data the program will ask for.

What's the pattern? $\text{nextAddr} = \text{curAddr} + 4$.

Reducing Compulsory Misses: Hardware Prefetching

There are many variants of hardware prefetching

- **Prefetch-on-miss:** prefetch $b + 1$ upon miss on b.
- **One block lookahead scheme:**
  - Initiate prefetch for block $b + 1$ when block b is accessed
  - Can extend to N-block lookahead.
- **Strided prefetch:** If the sequence of accesses observed is $b, b + N, b + 2N$, then prefetch $b + 3N$, etc.

https://software.intel.com/content/www/us/en/develop/articles/

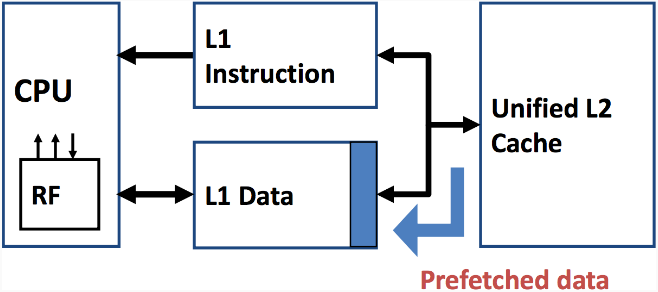
disclosure-of-hw-prefetcher-control-on-some-intel-processors.html

Reducing Compulsory Misses: Hardware Prefetching: Issues

We want our prefetching to be

- **Useful**: should produce cache hits
- **Timely**: should be not too late and not too early

We also have to be wary of cache and bandwidth pollution.



Reducing Compulsory Misses: Software Prefetching

We can attempt to have the hardware prefetch for us by accessing data before we need it.

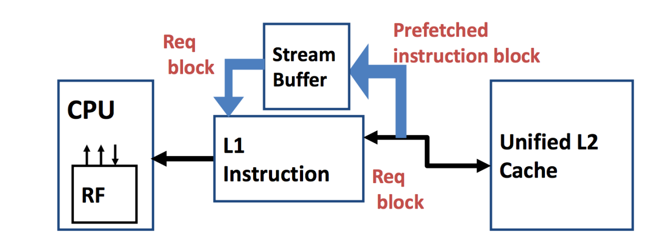
```
for (int i = 0; i < N; i++) {
    prefetch( &a[i + P] );
    prefetch( &b[i + P] );
    SUM = SUM + a[i] * b[i];
}
```

Although accesses are *predictable*, we will run into issues with getting the prefetch *timing* right.

- If you prefetch very close to when the data is requested, you may be too late.
- If you prefetch too early, you will cause pollution.
- You can estimate how long it will take for the data to come into L1 cache and set `P` accordingly.
- Why is this hard to do?

Reducing Compulsory Misses: Hardware Instruction Prefetching

- Fetch two blocks on miss: the requested block (`i`) and the next consecutive block (`i + 1`).
- Place the requested block in cache, and the next block in the instruction stream buffer.
- If miss in the cache but a hit in the stream buffer, move the stream buffer block into cache and prefetch the next block (`i + 2`).



Reducing Compulsory Misses: Restructuring

We can restructure the code to take advantage of the memory access pattern.

```
struct Atom {
    double v;
    double f;
    double3 p;
};

struct Atom atoms[N];

for (i = 0; i < N; i++)
    ... = atoms[i].v + ...

for (i = 0; i < N; i++)
    ... = atoms[i].f + ...

for (i = 0; i < N; i++)
    ... = atoms[i].p + ...
```

→

```
double vs[N];
double fs[N];
double3 ps[N];

for (i = 0; i < N; i++)
    ... = atoms[i].v + ...

for (i = 0; i < N; i++)
    ... = atoms[i].f + ...

for (i = 0; i < N; i++)
    ... = atoms[i].p + ...
```

Reducing Conflict Misses: Why They Happen

Conflict misses occur when the data we need was in the cache previously, but got evicted.

When do evictions occur?

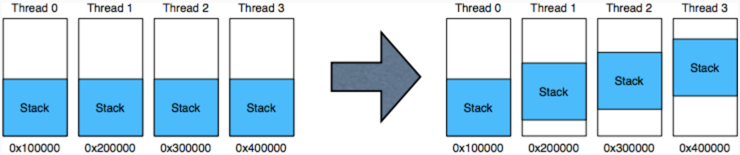
- Direct-mapped: another request mapped to the same cache line
- Associative: too many requests mapped to the same set

Example: assume a 4 KiB cache

```
while (1) {
    for (i = 0; i < 1024 * 1024; i += 4096) {
        sum += data[i];
    }
}
```

Reducing Conflict Misses: Colliding Threads and Data

- The stack and the heap tend to be aligned to large chunks of memory (maybe 128 MiB).
  - Threads often run the same code in the same way.
  - This means that thread stacks will end up occupying the same parts of cache.
  - Randomize the base of each thread's stack.



- Large data structures (for example, arrays) are also often aligned. Randomizing `malloc` can help.

Reducing Capacity Misses: Why They Happen

Capacity misses occur because the processor is trying to access too much data.

- *Working set*: the data that is currently important to the program
- If the working set is bigger than the cache, you are going to miss frequently.

Capacity misses are a bit hard to measure

- Easiest definition: non-compulsory miss rate in an equivalently-sized fully-associative cache
- Intuition: take away the compulsory misses and the conflict misses, and what you have left are the capacity misses

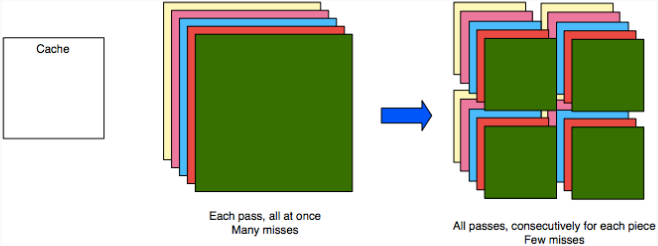
Reducing Capacity Misses: Basic Mitigations

- Increase capacity
- More associativity or more associative "sets"
  - Costs area and makes the cache slower
- Cache hierarchy does this implicitly already
  - If the working set "falls out" of the L1 cache, L2 cache can still be utilized
- In practice, you make L1 as big as you can within your cycle time and the L2 and L3 as big as you can while keeping it on chip.

Reducing Capacity Misses: Tiling

Say we have an application that needs to make several passes over a large array. Doing each pass in turn will “blow out” our cache.

“Blocking” or “tiling” the loops will prevent the blowout, but whether or not it’s possible to do so depends on the structure of the loop.



You can tile hierarchically to fit into each level of the memory hierarchy.

Practice

Practice: Affects of Prefetching: Worksheet Problem 1

What affects may prefetching have on

1. compulsory misses?
2. capacity misses?
3. conflict misses?

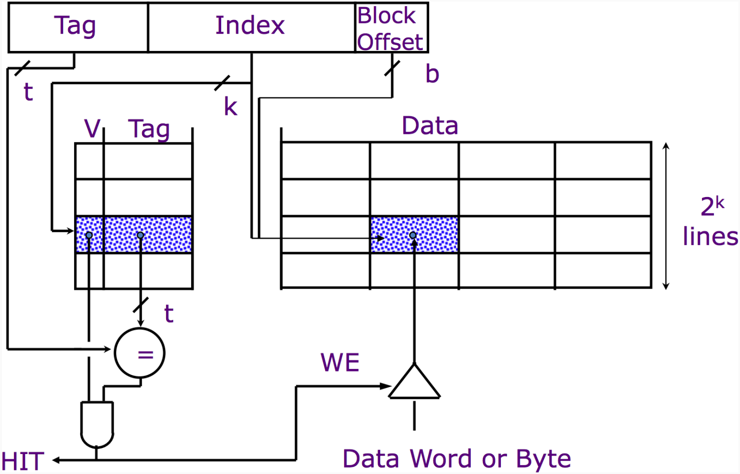
Practice: Prefetching Strategies: Worksheet Problem 2

Assume you have a cache where cache lines are 32 bytes. Also assume that integers take 4 bytes.

Write a loop in C that performs significantly better when using a *strided prefetcher* than when using a *one block lookahead scheme*.

A Few More Considerations

Write Performance: Process



Write Performance: Reducing Write Time

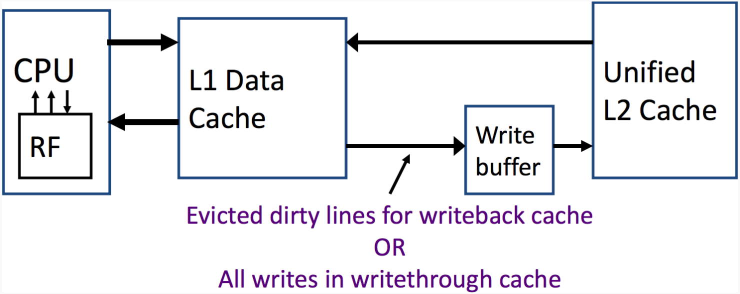
Problem: Writes take two cycles. One for tag check and another for writing the data.

Solution 1:

1. Check tag, put old data and write data into a buffer
2. If tag check fails, write old data back, otherwise, write new data.

Solution 2: pipeline the writes

Reducing Miss Penalty



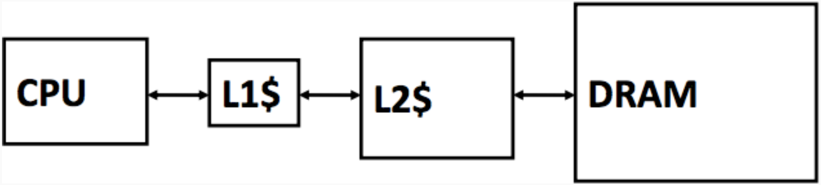
Problem: Write buffer may hold updated value of location needed by write miss

- Simple solution: on read miss, wait for write buffer to drain
- Faster solution: check write buffer addresses against the read miss address. If it matches, return the value in the write buffer. Otherwise, service the read before the writes in the buffer.

Multi-Level Caches

Problem: a memory level cannot be both large and fast

Solution: increasing sizes of cache at each level



- Local miss rate = misses in cache / accesses to cache
- Global miss rate = misses in cache / CPU memory accesses
- Misses per instruction = misses in cache / number of instructions

Presence of L2 Influences L1 Design

We can have a smaller L1 if there's also L2

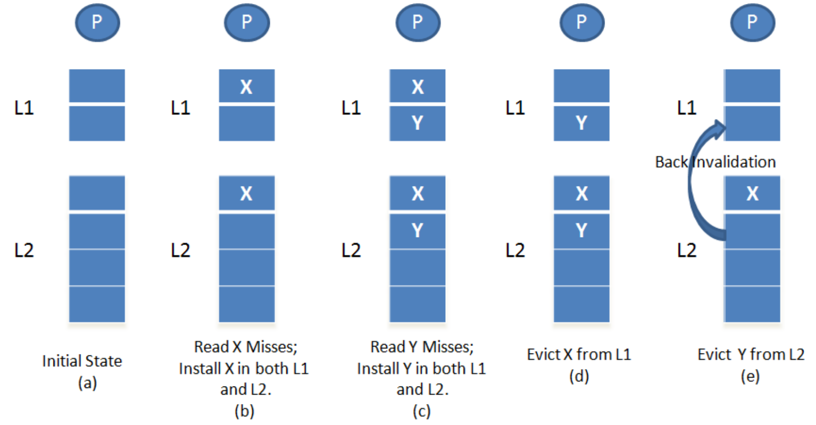
- Trade increased L1 miss rate for reduced L1 hit time and reduced L1 miss penalty
- Reduces average access energy

We can also use simpler write-through L1 with on-chip L2

- Write-back L2 absorbs write traffic, so writes don't go off-chip unless an eviction from L2 occurs.

Inclusion Policy: Inclusive

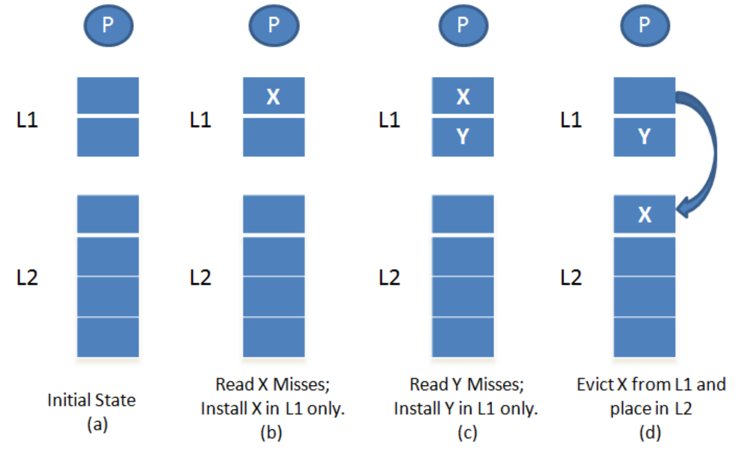
Inclusive multi-level cache: L2 cache holds copies of data in L1 cache



https://en.wikipedia.org/wiki/Cache_Inclusion_Policy

Inclusion Policy: Exclusive

Exclusive multi-level cache: L1 cache may hold data not in L2 cache



https://en.wikipedia.org/wiki/Cache_Inclusion_Policy

Victim Cache

What happens if the cache is W-way associative, but there are $W + 2$ lines mapped to each set?