Memory Efficient Code

Holger Pirk

Slides as of 18/01/21 13:31:21

Recall

- Caches have a fixed cache line size (say 64 byte)
- Caches have a fixed capacity (say 32 Kbyte)
  - Most caches evict the Least Recently Used cache line to make space for incoming ones

Definitions

Cache Miss A piece of data accessed by an instruction but not in cache

Data Hazards Pipeline stall cycles due to cache misses

Bandwidth vs. Latency Boundness

Classifying cache misses

- What access cause the cache miss?
  - If the value has been accessed before, we call the respective cache miss a **capacity miss**
  - If not, it is a **compulsory miss**

Classifying code

- We call code fully memory bound if all stall cycles are due to data hazards
- If the memory Bus is fully utilized we call it **memory bandwidth bound**
- If the memory Bus is **not** fully utilized we call it **memory latency bound**

Bandwidth vs. Latency Boundness

Measures

- If it is bandwidth bound, increase cache-line utilization:
- If it is latency bound, prefetch
- If it is capacity bound, reduce the footprint/hot dataset

Thank you for coming!

Compulsory Read Cache Misses

Consider this code

```
#include <stdlib.h>
struct tuple { int x; int y; int z;};
int sumIt(tuple* input, long size) {
  int sum = 0;
  for(size_t i = 0; i < size; i++)
    sum += input[i].x;
  return sum;
}
```

- How many data-hazards would this run into (assume an in-order processor)?
- $\frac{\text{datasize}}{\text{cache line size}}$

What can we do about it? What concepts could help?

Asynchonous processing!

Hardware Prefetching

- Caches speculatively load the next cache line
- How do they speculate?
  - Recognizing patterns: adjacent cache lines, strides, ...
  - Modern CPUs even recognize multiple interleaved patterns
  - The *generic cost model* distinguishes sequential and random

misses for that reason

- Works well for regular memory accesses
  - Breaks for irregular accesses (like data-dependent accesses)

Hardware Prefetching

Consider this code

```
#include <stdlib.h>
struct tuple { int x; int y; int z;};
int sumIt(tuple* input, long size, tuple* input2) {
  int sum = 0;
  for(size_t i = 0; i < size; i++)
    sum += input2[input[i].x].y;
  return sum;
}
```

- The CPU needs help...

Software Prefetching

- You can provide prefetching hints
  - `void __builtin_prefetch (const void *addr, ...)`

This is how you use them:

```
#include <stdlib.h>
struct tuple { int x; int y; int z;};
int sumIt(tuple* input, long size, tuple* input2) {
  int sum = 0;
  for(size_t i = 0; i < size; i++){
    sum += input2[input[i].x].y;
    __builtin_prefetch(&input2[input[i + 16].x]);
  }
  return sum;
}
```

https://godbolt.org/z/c2qBoU

But what if you are still memory bound?

Increase cache-line utilization!

Cache-line utilization

Definition

$$\text{Cache-line utilization} = \frac{\text{Data requested by instructions}}{\text{Data loaded into cache}}$$

How to increase cache-line utilization?

- Change the data layout in memory!

Poor cache-line utilization code

```
#include <stdlib.h>
struct tuple { int x; int y; int z;};
int sumIt(tuple* input, long size, tuple* input2) {
  int sum = 0;
  for(size_t i = 0; i < size; i++)
    sum += input2[input[i].x].y;
  return sum;
}
```

Cache-line utilization

Better cache-line utilization code

```
#include <stdlib.h>
struct tuple { int* x; int* y; int* z;};
int sumIt(tuple input, long size, tuple input2) {
  int sum = 0;
  for(size_t i = 0; i < size; i++)
    sum += input2.y[input.x[i]];
  return sum;
}
```

Cache-line utilization

Bad

```
#include <stdlib.h>
struct tuple { int x; int y; int z;};
int sumIt(tuple* input, long size, tuple* input2) {
  int sum = 0;
  for(size_t i = 0; i < size; i++)
    sum += input2[input[i].x].y;
  return sum;
}
```

Cache-line utilization

Good

```
#include <stdlib.h>
struct tuple { int* x; int* y; int* z;};
int sumIt(tuple input, long size, tuple input2) {
  int sum = 0;
  for(size_t i = 0; i < size; i++)
    sum += input2.y[input.x[i]];
  return sum;
}
```

But what about writing to memory?

Compulsory Write Cache Misses

Consider this code

```
int* output = getOutput();
for(size_t i = 0; i < size; i++)
        output[i] = 0;
```

- How many data-hazards would this run into (assume an in-order processor)?
- $\frac{2 \times \text{datasize}}{\text{cache line size}}$

What can we do about it? What concepts could help?

Lazy processing!

Memory allocation: Eager

```
static void selection(benchmark::State& state) {
        auto argument = state.range(0);
        auto input = (int*)malloc(sizeof(int) * argument);
        for(size_t i = 0; i < argument; i++)
                input[i] = (i * 12343) % 499;

        for(auto _ : state) {
                auto output = (int*)malloc(sizeof(int) * argument);
                auto outI = 0ul;
                for(size_t i = 0; i < argument; i++) {
                        if(input[i] > 5)
                                output[outI++] = i;
                }
                benchmark::DoNotOptimize(output);
                free(output);
        }
        free(input);
}
BENCHMARK(selection)->Arg(1024 * 1024 * 32);
```

Memory allocation: Lazy

```
for(size_t i = 0; i < argument; i++) {
  if(capacity <= outI)
    output = (int*)realloc(output, (capacity += reallocSize) * sizeof(int));
  if(input[i] > 5)
    output[outI++] = i;
}
```

Memory allocation: None

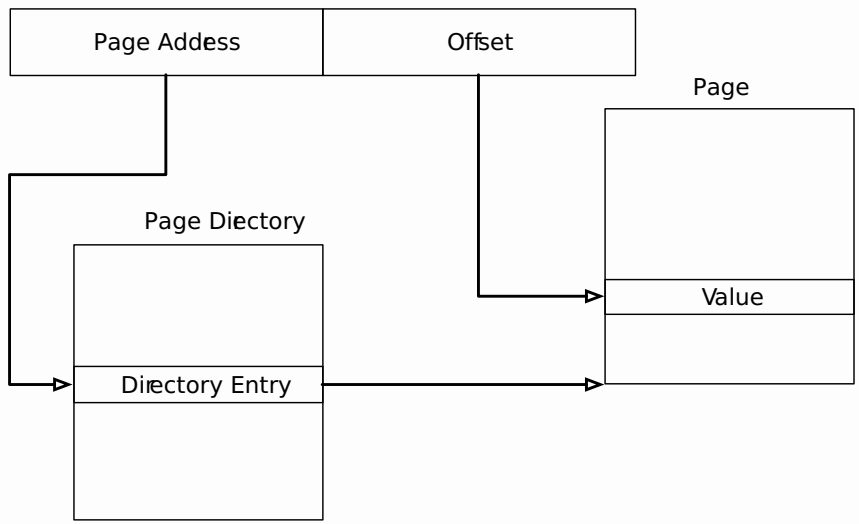
```
state.PauseTiming();
auto output = (int*)malloc(sizeof(int) * argument);
state.ResumeTiming();
auto outI = 0ul;
for(size_t i = 0; i < argument; i++) {
  if(input[i] > 5)
    output[outI++] = i;
}
```

Why?

Address Translation

- Pointers are virtual addresses, i.e., local to a process
- Multiple processes can have the same virtual addresses point to different values
  - (this is important to isolate programs)
- But memory has physical addresses

- These need to be translated using a directory
  - This is an expensive process called page-table walking
- Address translation is performed by the operating system and hardware



Exploiting virtual memory

- Virtual memory pages support copy on write
- Trick: have a single, zero-initialized page marked as **Copy-on-Write**
- Modifying it causes an interrupt the kernel needs to handle
- While handling that interrupt, the kernel copies the page

The Translation Lookaside Buffer (TLB)

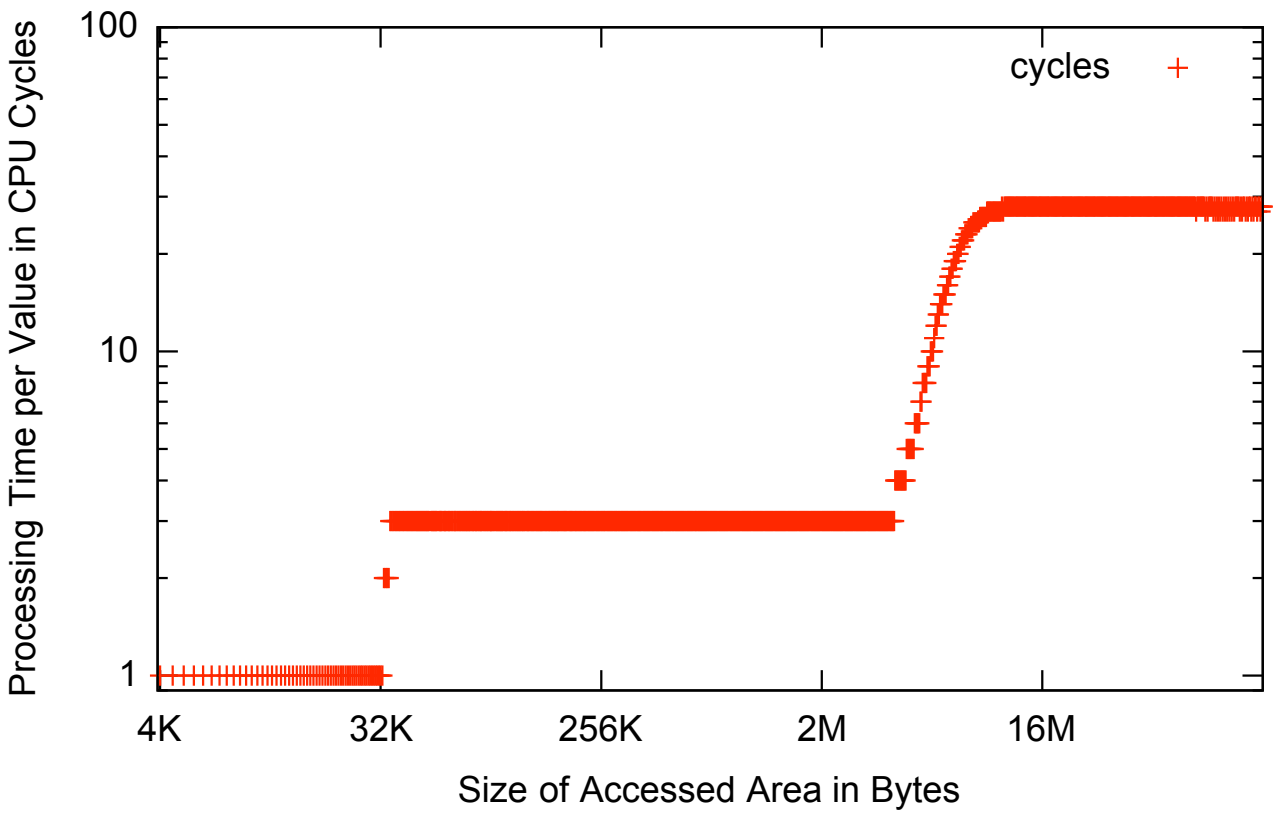
- The TLB is a cache dedicated to entries of this mapping table
- A TLB miss means the address has to be re-calculated (causing a page-table walk)
- This can be done either in hardware (using a pre-defined address table scheme)

Discussion

What happens when a prefetch causes a TLB miss?

The Problem: thrashing (i.e., capacity bound code)

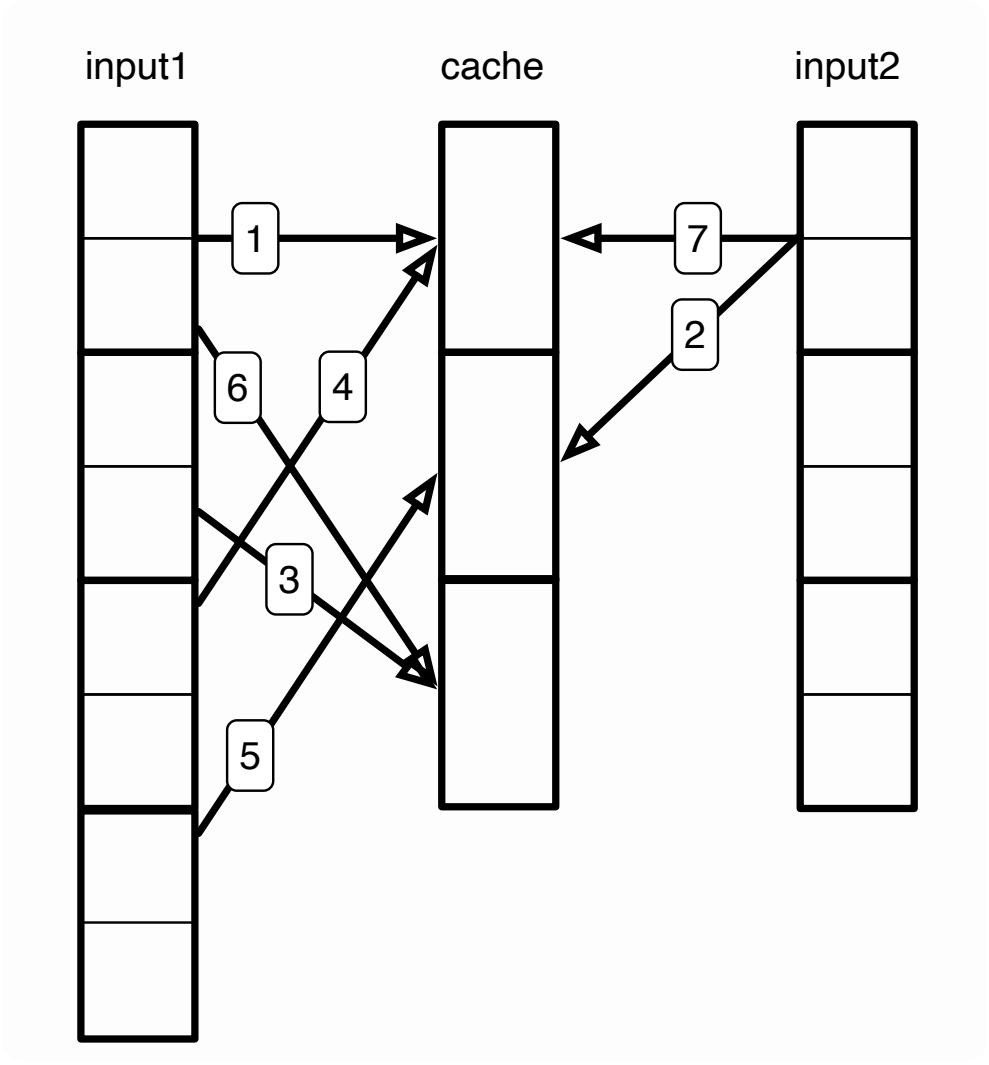
CPU Memory Access Latencies (by hot data set size)



Thrashing

```
static void NestedLoops(benchmark::State& state) {
        auto xrange = state.range(0);
        auto yrange = state.range(1);
        int* input0 = getInput(0);
        int* input1 = getInput(1);
        for(auto _ : state) {
                int sum = 0;
                for(size_t i = 0; i < xrange; i++) {
                        for(size_t j = 0; j < yrange; j++) {
                                sum += input0[i]*input1[j];
                        }
                }
                benchmark::DoNotOptimize(sum);
        }
}
BENCHMARK(NestedLoops);
```

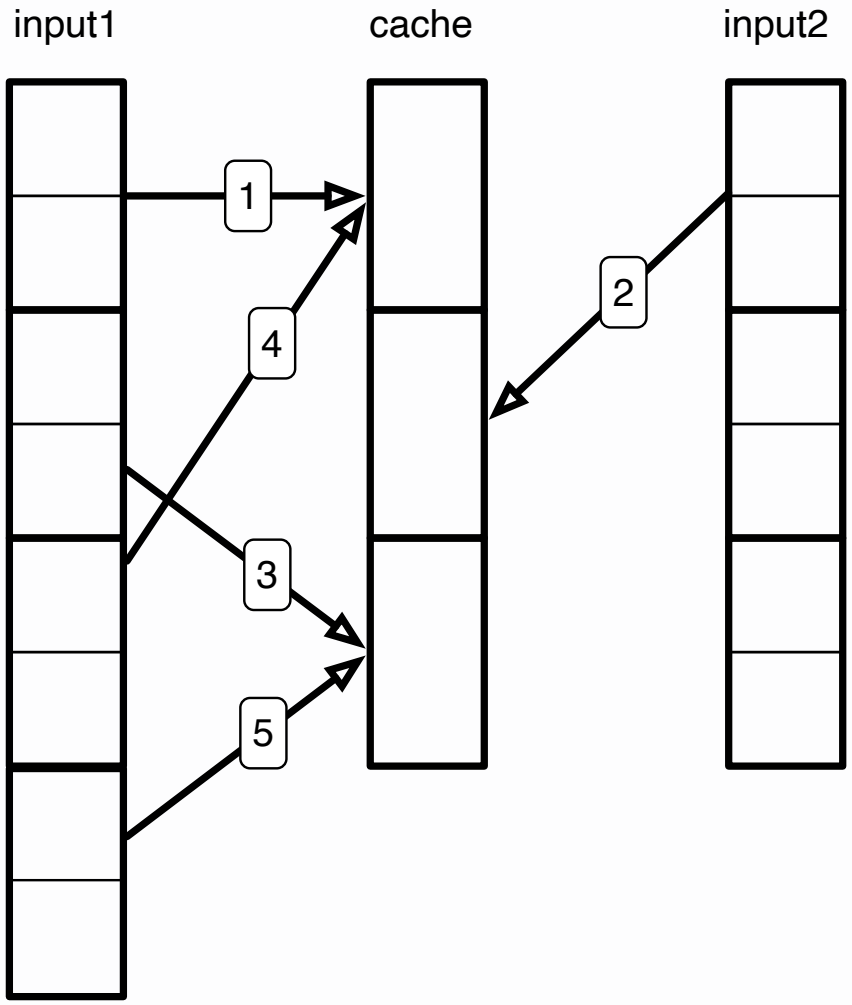
Thrashing



The solution: loop tiling

```
static void TiledLoops(benchmark::State& state) {
        auto xrange = state.range(0);
        auto yrange = state.range(1);
        int* input0 = getInput(0);
        int* input1 = getInput(1);
        for(auto _ : state) {
                int sum = 0;
                for(size_t tileI = 0; tileI < xrange; tileI += tileSize) {
                        for(size_t tileJ = 0; tileJ < yrange; tileJ += tileSize) {
                                for(size_t inTileI = 0; inTileI < tileSize; inTileI++) {
                                        auto i = tileI*tileSize+inTileI;
                                        for(size_t inTileI = 0; inTileI < tileSize;
                                        ↪  inTileI++) {
                                                auto j = tileJ*tileSize+inTileJ;
                                                sum += input0[i] * input1[j];
                                        }
                                }
                        }
                }
                benchmark::DoNotOptimize(sum);
        }
}
BENCHMARK(UntiledLoops);
```

Loop tiling

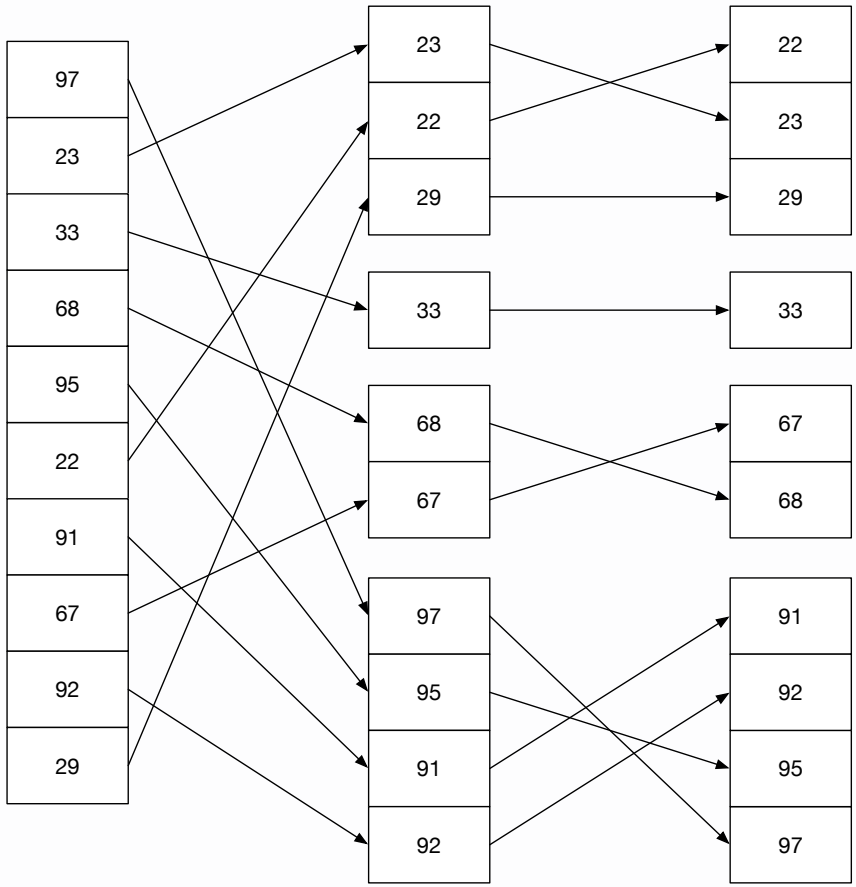


- Two effects:
  - Cache lines on the right are never thrashed
  - Repeated access to cache lines keeps them hot
- Loop tiling is (almost) a no-brainer

There are cases that are not as clear-cut!

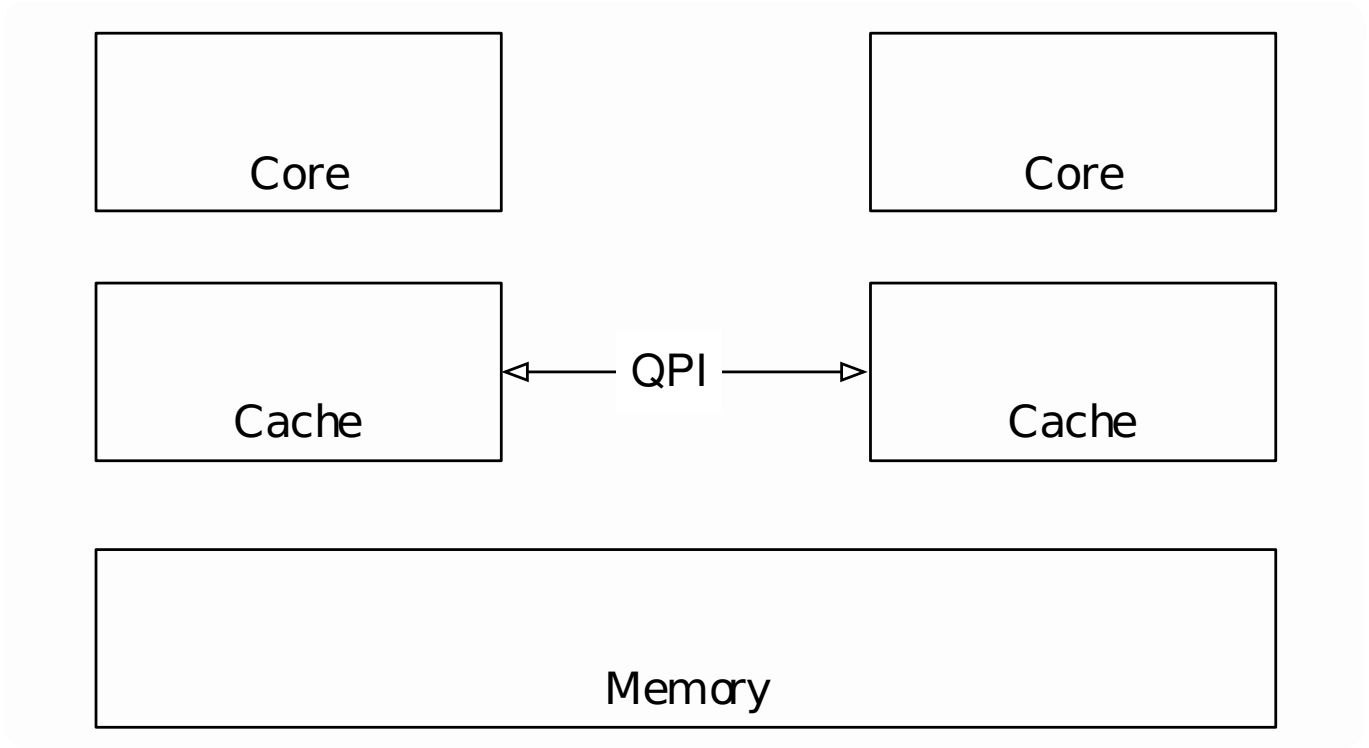
Radix-Sorting

Who feels like explaining this?

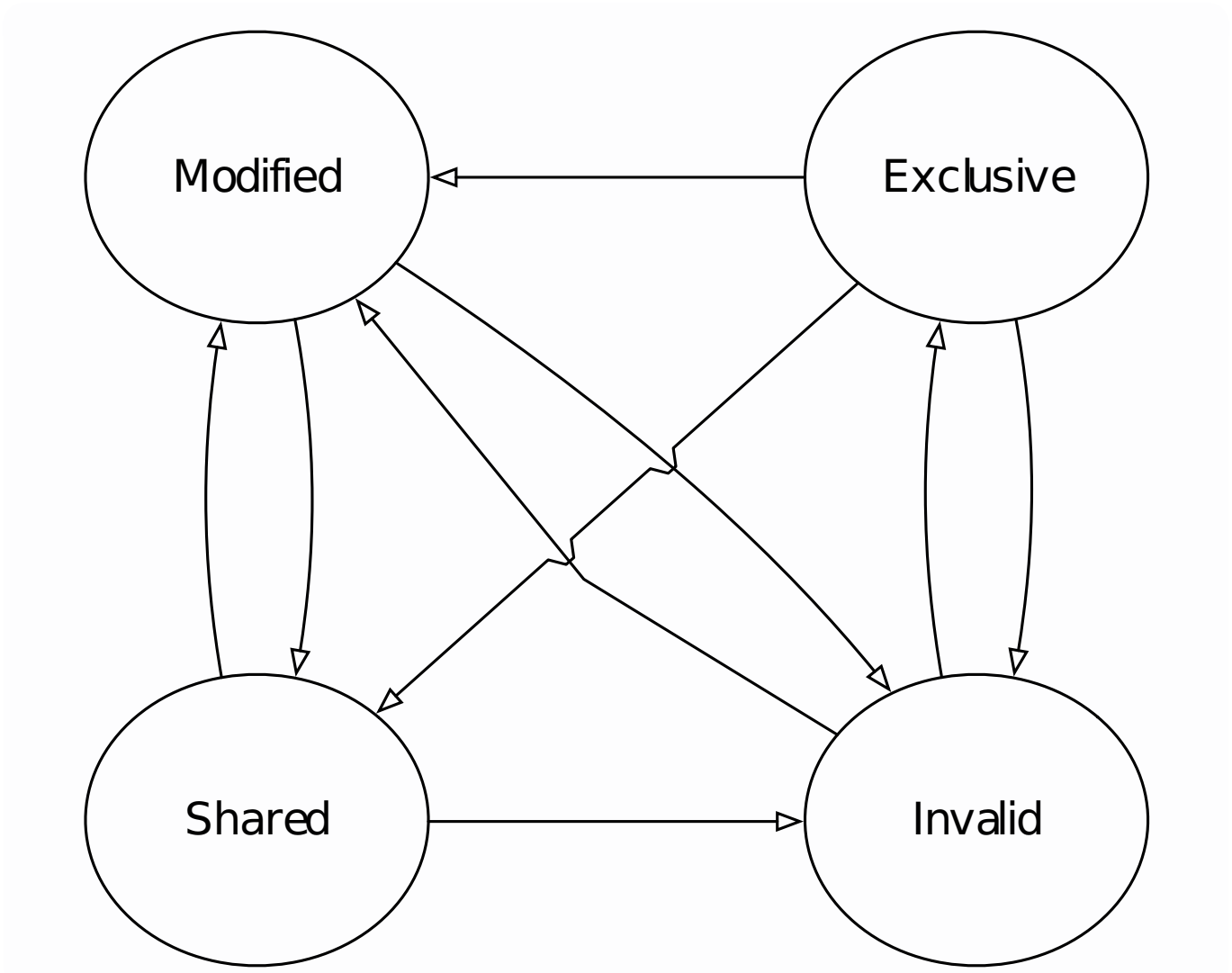


Bottom line: multiple passes can be beneficial!

Multicore effects



MESI



False Sharing - the final control-flow hazard