Efficient Code

Holger Pirk

Slides as of 08/02/21 09:26:03

Imperial College London

Bottleneck

Definition: Bottleneck

The resource with the highest utilization

Challenge:

Can we build a system without a bottleneck – one where all resources are equally utilized?

Imperial College London

A first attempt at balance

Amdahl said

A balanced computer system needs 1 MB of main memory capacity and 1Mbit per second of I/O bandwidth per MIPS of CPU performance. – Gene Amdahl (paraphrased by Hennessy & Patterson)

- Do we buy into that?

Imperial College London

Let's calculate. . .

- My laptop's CPU has

```
IPS = 2.9 (* GHz *) * 4 (* Cores *) * 4 (* Execution slots/cycle *)
```

46.4

```
Bandwidth = 37.5 (* GB/s *) * 8 (* bits/byte *)
```

300

```
Memory = 16 (* GB *) * 1024 (* MB/GB *)
```

- Verdict: this thing is **way** underpowered in terms of CPU
- Who would design a system like that?

Imperial College London

Balance is not a constant, . . .

- Datatypes are larger today than they were in Amdahl's time
- CPUs have all kinds of Co-processors (on-die GPUs, DMA engines, . . .)
- Cache lines are larger, causing wasted memory bandwidth, etc.
- **However**, . . .

Imperial College London

. . . balance is a function of code

- Hardware optimizations have varying impact on code
- There is no such thing as a balanced system (not even balanced applications), only balanced sections of code
- However, **perfect balance is an ideal** – practically unachievable

Imperial College London

Amdahl got one thing right. . .

- The fundamental tradeoff is between CPU and Bandwidth efficiency
- We distinguish *balanced*, *compute-bound*, *latency-bound* and *bandwidth-bound* code

Imperial College London

You can influence bottlenecks

- High-level techniques:
  - Choice of algorithm
  - Memoization
  - Compression
  - . . .
- Low-level techniques
  - Those are our focus

Imperial College London

Let's focus on compute-bound code first

Imperial College London

When CPU-efficiency matters

- For CPU-bound applications, i.e., those that
  - are poorly implemented
  - operate on small (cache-resident) datasets
  - are math-heavy (especially floating point math)
 - apply memory-oriented optimizations

Imperial College London

CPU-efficiency: target metric

- Well, wallclock time, obviously
- Cycles per Instruction (CPI) is not that useful as an optimization metric since the instructions are a parameter
- Stall cycles are a good indicator (though not perfect either)
- Stall cycles are caused by hazards:

Imperial College London

CPU-efficiency: Hazards

- **Control-hazards** - stalls due to data-dependent changes in the control flow
- **Structural hazards** - stalls due to the lack of execution resources (registers, execution ports, ...)
- **Data-hazards** - stalls due to operands (i.e., data) not being available on time

Imperial College London

Recall: We have discussed Hazards before

Imperial College London
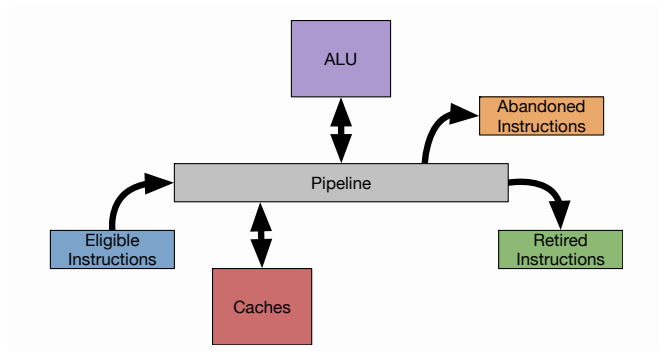
CPU optimizations is all about mitigating these

Imperial College London

In part, that is done in hardware

Imperial College London

Pipelined Execution exploits independence of stages of different instructions
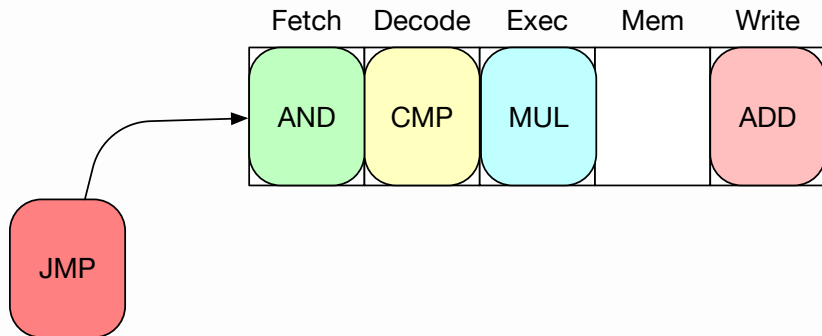
CPU Pipeline Design

Fundamental design decisions, i.e., the tools we work with:

- Is the system executing instructions speculatively?
- Is the system executing instructions dynamically parallel?
  - Superscalar execution
  - Note that this is not multi-core parallelism!
- Is the system executing instructions out-of-order?
- Is the system executing instructions statically parallel?
  - Statically bundled packages (SIMD or VLIW)

Imperial College London

Speculative execution

- Keeps the pipeline filled if no instructions are eligible
- We have covered these
- Addresses (some) control hazards

Imperial College London

Superscalar Execution
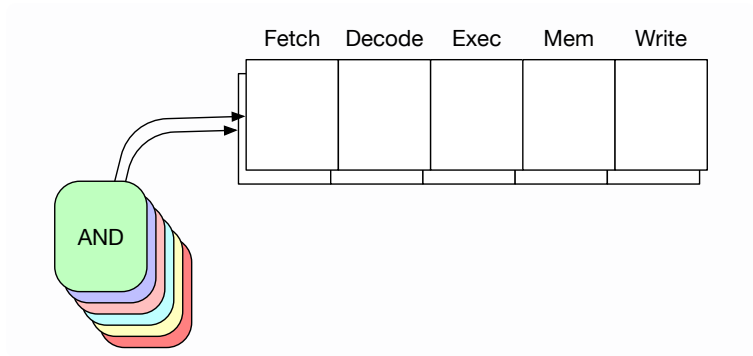
Pipelined Execution Different instructions can be in different stages

Superscalar Execution Different instructions can be in the same stage

Imperial College London

Out-of-order execution exploits independence of the instructions

- Kicks in after decoding decode-phase
- Instructions that have no unsatisfied dependencies move on to execution stage
- Examples:
  - In the statement `a*b+d*e+b*b`, `b*b` is scheduled before `d*e` when `b` is L1 cache-resident
  - Evaluate an integer addition even if there are floating point additions waiting for execution ports
- Addresses data and structural hazards, leaves control hazards

SIMD

Single Instruction Multiple Data
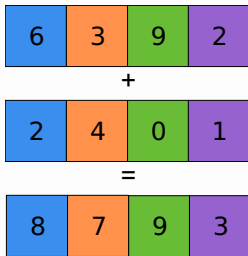
- The CPU performs the same operation on multiple data items
- Introduced for multimedia but useful for general computing
- Applications range from the trivial to the mind-blowing
- Every CPU-generation has new instructions: MMX, SSE, SSE2, AVX, AVX2, AVX512
- They use specialized vector-registers

Imperial College London

A sidenote: **VLIW**

- SIMD means
  - single instruction
  - the same operation on multiple data items
- **V**ery **L**ong **I**nstruction **W**ords
  - single instruction
  - different operations on multiple or the same data item
  - Basically compiler-controlled superscalar execution

Imperial College London

Now that we know the tools. . .

Imperial College London

... let us discuss:

Imperial College London
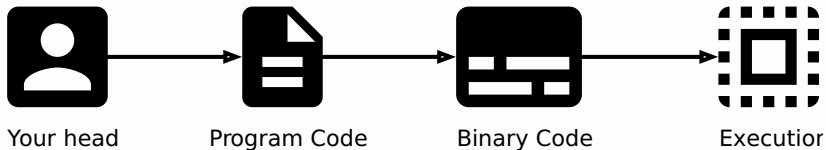
Write **runtime** predictable code

- Do not evaluate code in the critical path whenever possible
- Evaluate code as early as possible

Imperial College London

What do I mean by that?

Imperial College London

The code execution process

Your head → Program Code → Binary Code → Execution

Imperial College London

But who said it has to be three transformations?

Imperial College London

Partial evaluation (make the compiler work for you)

- Idea: Treat programs evaluation as a multi-phased process
  - In every phase, you know more about the result
  - i.e., you can evaluate more of it/create a more specialized program
  - the fix point is obviously the actual result

Imperial College London

Examples of partial evaluation

- Function inlining
- JiT compilation
- Symbolic Programming
- Constant expression evaluation
  - Careful: when are things truly constants?

Imperial College London

Constants

```
int result(int input) {
  return input*3*5;
};

int result2(int input) {
  int three = 3-1;
  int five = 4+1;
  return input*three*five;
};

int three = 3;
int five = 5;
int result3(int input) {
  return input*three*five;
};
```

- see https://godbolt.org/z/X3nQb2
- What constitutes a constant depends on your language's semantics

Imperial College London

Lifting expensive operations

Lifting Moving work from a section that is executed often into one that is executed seldomly

- Standard example moving invariants out of a loop:
    - `for (size_t i = 0; i < N; i++) output[i] = 7*8;`
    - `int tmp = 7*8; for (size_t i = 0; i < N; i++) output[i] = tmp;`
- Can be generalized to *loop specialization*

Loop specialization

Example

```c
#include <stdlib.h>
void scaleVector(int* input, size_t inputSize, int scale) {
  for(size_t i = 0; i < inputSize; i++)
    input[i] *= scale;
};

void scaleVectorSmart(int* input, size_t inputSize) {
  if(scale != 1) {
    if(scale == 2)
      for(size_t i = 0; i < inputSize; i++)
        input[i] <<= 1;
    else
      for(size_t i = 0; i < inputSize; i++)
        input[i] *= scale;
  }
};
```

The problem with loop specialization

- Leads to code duplication
- You can do it by hand but...
- ...the compiler could easily do these for you! However,
- How many special cases should it generate?
- Compilers need help
- Introducing: Metaprogramming...

Imperial College London

Metaprogramming – ...what you can do for your compiler

The idea

- Generating all these special cases at compile-time
- Allows the compiler to apply optimizations for the special cases

Implementations

- Macros in LISP (and it's dialects), Haskell, Scala
- Templates in C++, Macros in C (they are called the same but aren't)
- Generics in Python, Java, C#

Imperial College London

Metaprogramming in C++

Metaprogramming – ...what you can do for your compiler

Turn this code...

```
void scaleVector(int* input, size_t inputSize, int scale) {
  for(size_t i = 0; i < inputSize; i++)
    input[i] *= scale;
};
int useIt(int* input, size_t size) {
  scaleVector(input, size, 2);
  scaleVector(input, size, 1);
  scaleVector(input, size, 0);
}
```

...into this

```
template <int scale> void scaleVectorPE(int* input, size_t inputSize) {
  for(size_t i = 0; i < inputSize; i++)
    input[i] *= scale;
};
int useIt(int* input, size_t size) {
  scaleVectorPE<2>(input, size);
  scaleVectorPE<1>(input, size);
  scaleVectorPE<0>(input, size);
}
```

Imperial College London

Metaprogramming – . . . what you can do for your compiler

Same example

```cpp
#include <functional>
#include <map>
#include <stdlib.h>
using namespace std;
template <int scale> void scaleVectorPE(int* input, size_t inputSize) {
  for(size_t i = 0; i < inputSize; i++)
    input[i] *= scale;
};

void scaleVector(int* input, size_t inputSize, int scale) {
  map<int, function<void(int*, size_t)>> const specialCases //
      {{0, scaleVectorPE<0>},                               //
       {1, scaleVectorPE<1>},
       {2, scaleVectorPE<2>},
       {4, scaleVectorPE<4>}};
  if(specialCases.count(scale))
    specialCases.at(scale)(input, inputSize);
  else
    for(size_t i = 0; i < inputSize; i++)
      input[i] *= scale;
};
```

Imperial College London

Function inlining is an instance of Partial Evaluation

Imperial College London

It turns jumps into branches

Imperial College London

Even more predictable: branch-free code

Imperial College London

Control-dependencies **can** cause control hazards

```
for(size_t i = 0; i < inputSize; i++)
  if(input[i] < high)
    output[outI++] = input[i];
```

Imperial College London

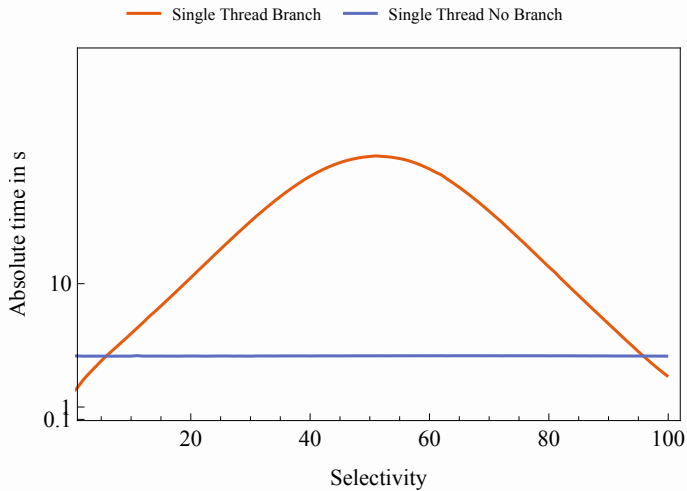# Recall

Imperial College London
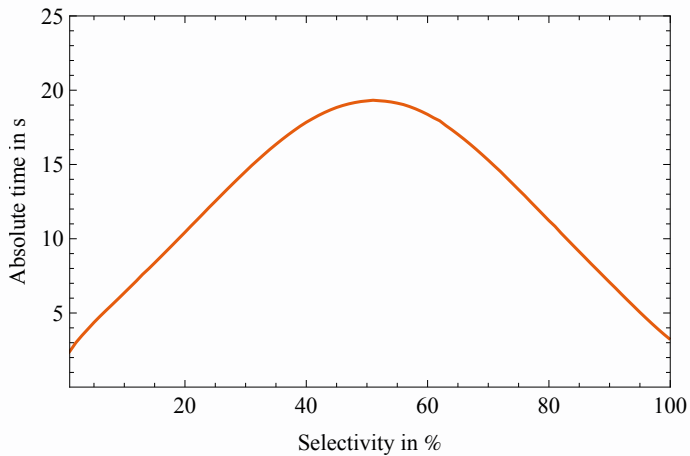
Branch-free code **can** eliminate control hazards

```
for(size_t i = 0; i < inputSize; i++) {
  output[outI] = input[i];
  outI += (input[i] < high);
}
```

- This is called predication or if-conversion depending on who you ask

Imperial College London

Impact

Imperial College London

However. . .

Imperial College London

...Control-dependencies **might not** cause control hazards

Unpredicated code on sorted input

```
---------------------------------------------------------------
Benchmark                   Time           CPU  Iterations
---------------------------------------------------------------
selection/50            49951160 ns    49951326 ns          12
selection/100           52085927 ns    52085751 ns          10
selection/150           54732263 ns    54731816 ns          13
selection/200           57072435 ns    57072634 ns          12
selection/250           59132204 ns    59132429 ns          11
selection/300           62251051 ns    62250581 ns          11
selection/350           66982496 ns    66982226 ns          10
selection/400           67639907 ns    67640118 ns           9
selection/450           70828641 ns    70828503 ns           9
selection/500           73505346 ns    73505682 ns           9
selection/550           73040486 ns    73040182 ns           9
```

- Bottom line: Predictability is input-dependent

Imperial College London

Guideline

- Branch-predictors are the way they are for a reason
- Lots of code is already very predictable
- Predication/if-conversion turns control dependencies into data dependencies
  - Data dependencies cause data hazards, predication amplifies these
  - only apply after measuring

Imperial College London

So, now the work per-iteration is 100% predictable

Imperial College London

We can do more if the number of iterations is predictable as well

Imperial College London

SIMD vectorization

- Compilers try to auto-vectorize
    - They succeed for simple cases like this
        - `for (size_t i = 0; i < 1024; i++) out[i] = in1[i]*in2[i]`
    - Write code with vectorization in mind, godbolt your code to be sure
- If they fail: explicitly vectorize using intrinsics
- Intrinsics are c-functions that map directly to hardware instructions:
    - `_mm256_<intrin_op>_<suffix>(<data type> <parameter1>, <data type> <parameter2>,` `<data type> <parameter3>)`
    - Check out this page:
        - https: //software.intel.com/sites/landingpage/IntrinsicsGuide

Imperial College London

SIMD: Example

```cpp
#include <immintrin.h>
#include <benchmark/benchmark.h>
#include <random>
#include <iostream>
using namespace std;

union v8f {
  float floats[8];
  __m256 simdVector;
};
```

Imperial College London

SIMD: Example

```cpp
auto bounds1 = state.range(0);
auto bounds2 = state.range(1);
auto input1 = new int[bounds1];
auto input2 = new float[bounds2];

uniform_int_distribution<mt19937::result_type> randomInRange(0, bounds2);
for(size_t i = 0; i < bounds1; i++)
  input1[i] = randomInRange(rng);
for(size_t i = 0; i < bounds2; i++)
  input2[i] = randomInRange(rng);

for(auto _ : state) {
  float sum = 0;
  for(size_t i = 0; i < bounds1; i++) {
    sum += input2[input1[i]];
  }
  // ...
```

Imperial College London

SIMD: Example

```
float sum = 0;
for(size_t i = 0; i < bounds1 / 8; i++) {
  v8f values{.simdVector = _mm256_i32gather_ps(
                input2, ((__m256i*)input1)[i], sizeof(int))};
  for(size_t i = 0; i < 8; i++)
    sum += values.floats[i];
}
```

Results

```
2019-02-04 21:37:54
Running /Users/hlgr/Projects/performance-engineering-class/Slides/a.out
Run on (8 X 2900 MHz CPU s)
CPU Caches:
  L1 Data 32K (x4)
  L1 Instruction 32K (x4)
  L2 Unified 262K (x4)
  L3 Unified 8388K (x1)
--------------------------------------------------------------
Benchmark                    Time           CPU Iterations
--------------------------------------------------------------
simd/1048576/16         1301911 ns   1298002 ns        557
scalar/1048576/16       1246102 ns   1243267 ns        565
```

Imperial College London

SIMD: Example

```
v8f sums{};
for(size_t i = 0; i < bounds1 / 8; i++) {
  v8f values{.simdVector = _mm256_i32gather_ps(
                 input2, ((__m256i*)input1)[i], sizeof(int))};
  sums.simdVector = _mm256_add_ps(values.simdVector, sums.simdVector);
}
float sum = 0;
for(size_t i = 0; i < 8; i++)
```

Imperial College London

Results

```
2019-02-04 21:37:54
Running /Users/hlgr/Projects/performance-engineering-class/Slides/a.out
Run on (8 X 2900 MHz CPU s)
CPU Caches:
  L1 Data 32K (x4)
  L1 Instruction 32K (x4)
  L2 Unified 262K (x4)
  L3 Unified 8388K (x1)
-------------------------------------------------------------
Benchmark                      Time          CPU Iterations
-------------------------------------------------------------
simd/1048576/16           1301911 ns   1298002 ns        557
scalar/1048576/16         1246102 ns   1243267 ns        565
properSimd/1048576/16      285140 ns    283631 ns       2442
```

Bottom line

- Use intrinsics for hard to auto-vectorize code
- Try to keep data in vector registers
- Godbolt your code

Imperial College London

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

Imperial College London

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**

Imperial College London

Bandwidth vs. Latency Boundness

Imperial College London

Thank you for watching!

Imperial College London

Just kidding :-)

Imperial College London
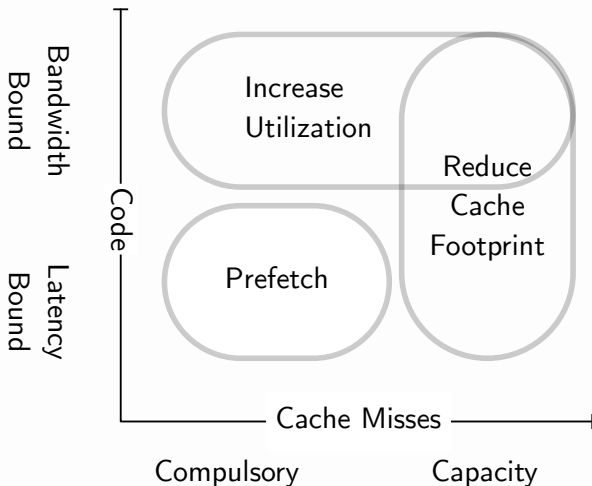
Compulsory Cache Misses

Consider this code

```c
#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)?

- $\dfrac{\text{datasize}}{\text{cache line size}}$

Imperial College London

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

Imperial College London

Asynchonous processing!

Imperial College London

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)

Imperial College London

Hardware Prefetching

Consider this code

```c
#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...

Imperial College London

Software Prefetching

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

This is how you use them:

```c
#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

Imperial College London

But what if you are still memory bound?

Increase cache-line utilization!

Imperial College London

Cache-line utilization

Definition

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

```c
#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;
}
```

Imperial College London

Cache-line utilization

Better cache-line utilization code

```c
#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;
}
```

Imperial College London

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;
}
```

Imperial College London

Cache-line utilization

> ## Good
>
> ```c
> #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;
> }
> ```

The HPC-folks call this Array-of-Structs to Struct-of-Arrays, Database folks call this decomposed storage

Imperial College London

Okay, so what about those capacitive misses?

Imperial College London

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

CPU Memory Access Latencies (by hot data set size)

Imperial College London
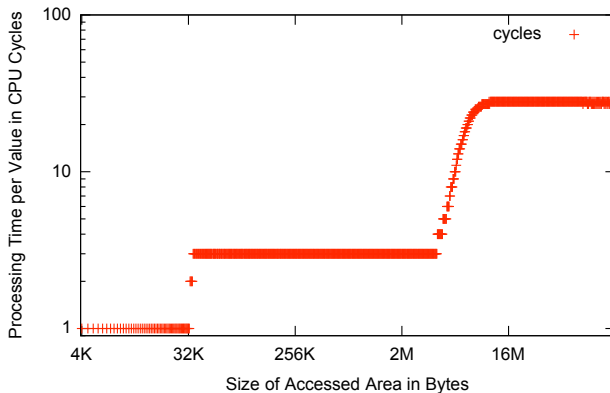
Thrashing

```cpp
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);
```

Imperial College London

Thrashing

Imperial College London
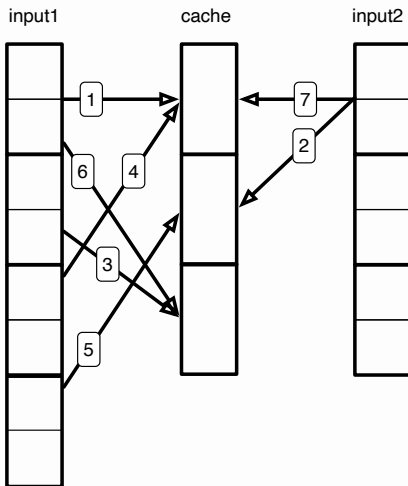
The Solution: Loop Tiling

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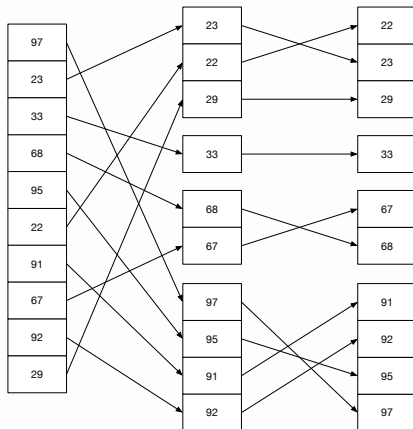
Loop Tiling
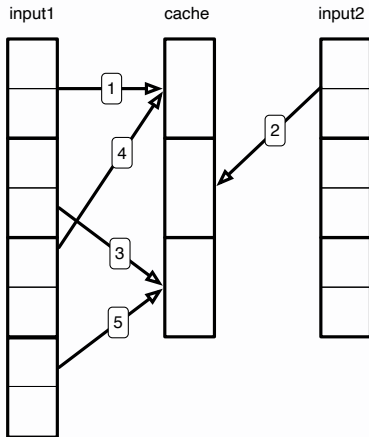
- Two effects:
  - Cache lines on the right are never thrashed
  - Repeated access to cache lines keeps them hot
  - Loop tiling is (almost) a no-brainer

Imperial College London

There are cases that are not as clear-cut!

Imperial College London

Radix-Sorting

Imperial College London

Bottom line: multiple passes can be beneficial!
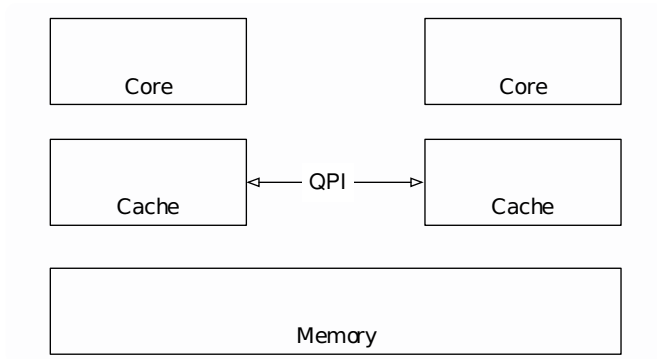
Imperial College London
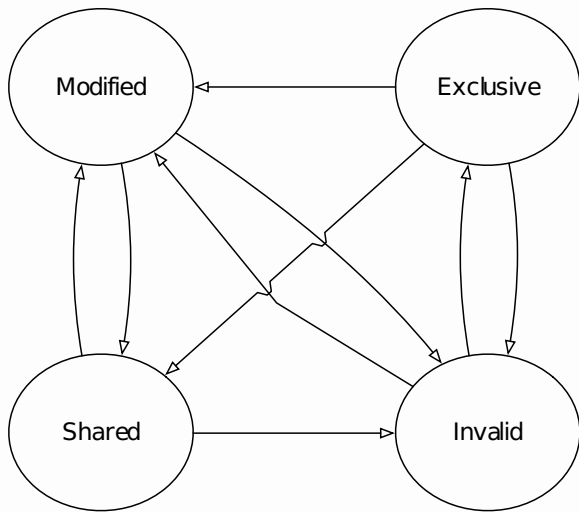
Conclusion

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

Imperial College London
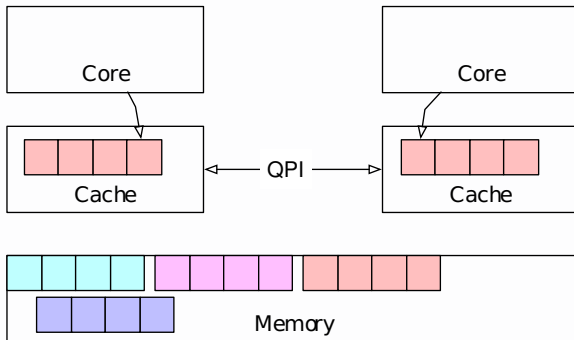
Before I let you go

Imperial College London

Multicore effects

Imperial College London

MESI

Imperial College London

False Sharing - the final control-flow hazard

Imperial College London

Provide feedback, please!

`https://co339.pages.doc.ic.ac.uk/feedback/efficiency`

Imperial College London

Get the slides online


```
https://co339.pages.doc.ic.ac.uk/decks/Efficiency.pdf
```

Imperial College London