Introduction to Performance Engineering

Nick Cameron

PingCAP TiDB

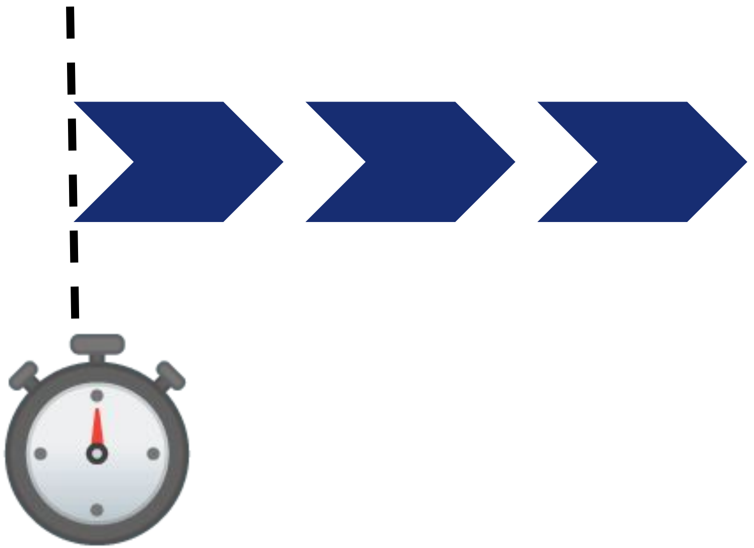
Disclaimers

- I’m not an expert
- Specific to general-purpose hardware
- Doesn’t cover domain-specific issues

Background

Throughput

- Work done in a given time
- QPS

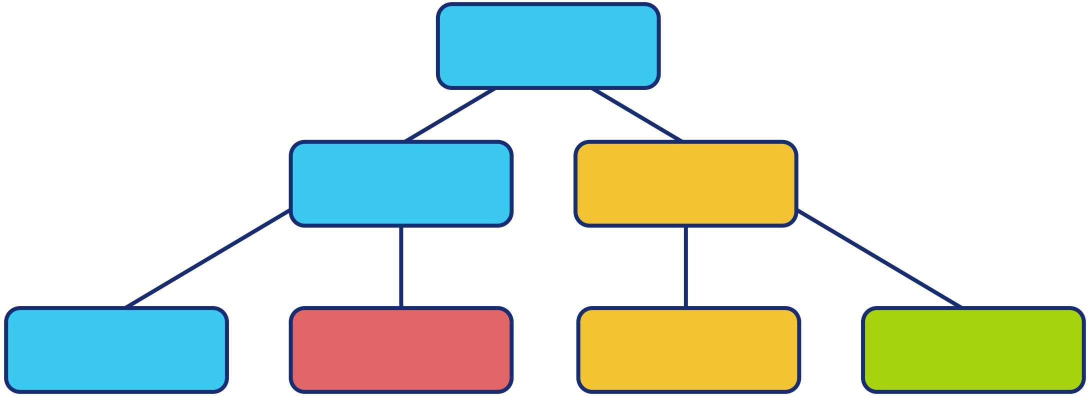


Latency

- Time taken by one unit of work
- µs

Macro-optimisations

- Potentially large gains
- Often trade-offs
- High risk



Micro-optimisations

- No trade-off
- Predictable
- Incremental
- Small gains
  - These compound:
  - 1% per week = 66% per year
  - (opposite is also true: -1%/week = -40%/year)

Hot code

- Focus on hot code (bottlenecks)
- Is it really hot?

Graphics example

60 fps

```
loop {
  once_per_frame();
  ...
}
```

Graphics example

60 fps

16.7 ms

~2 million instructions

```
loop {
  once_per_frame();
  ...
}
```

PingCAP TiDB

Graphics example

4k, 60 fps

```
loop {
  for x in 0..width {
    for y in 0..height {
      once_per_pixel();
    }
  }
  ...
}
```

Graphics example

4k, 60 fps

0.5bn pixels per second

1.9ns

60 instructions

```
loop {
  for x in 0..width {
    for y in 0..height {
      once_per_pixel();
    }
  }
  ...
}
```

Big O

- $O(n)$, $O(e^n)$, etc
- Growth which dominates for large n

Big O

- How big is your n?
  - O(1)
  - O(n), O(n^2), ...
  - O(e^n)

Big O

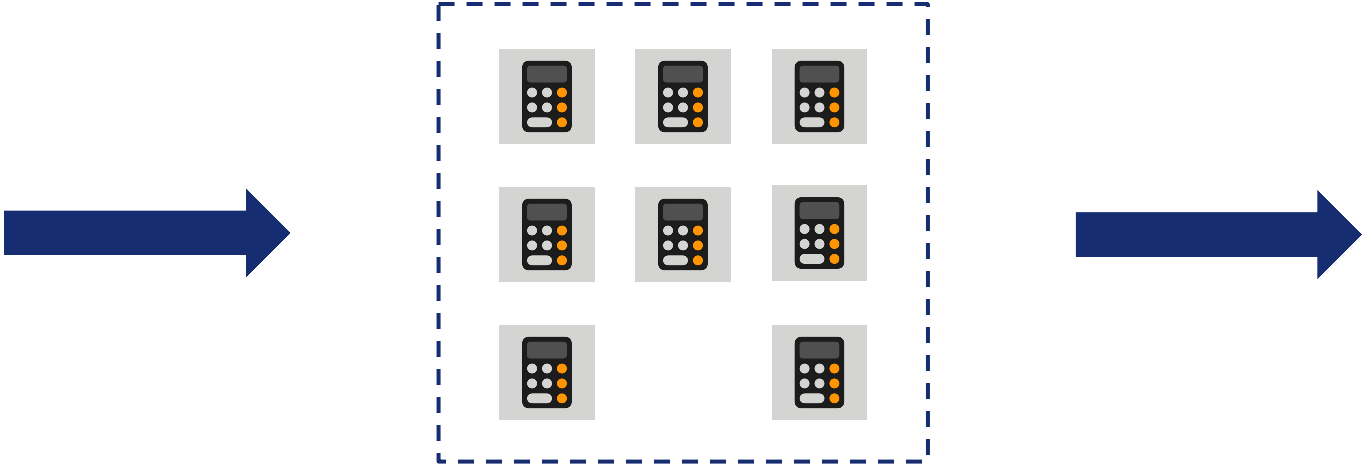
- Worst case
    - average case?
    - amortised cost?
- How big is the constant factor?

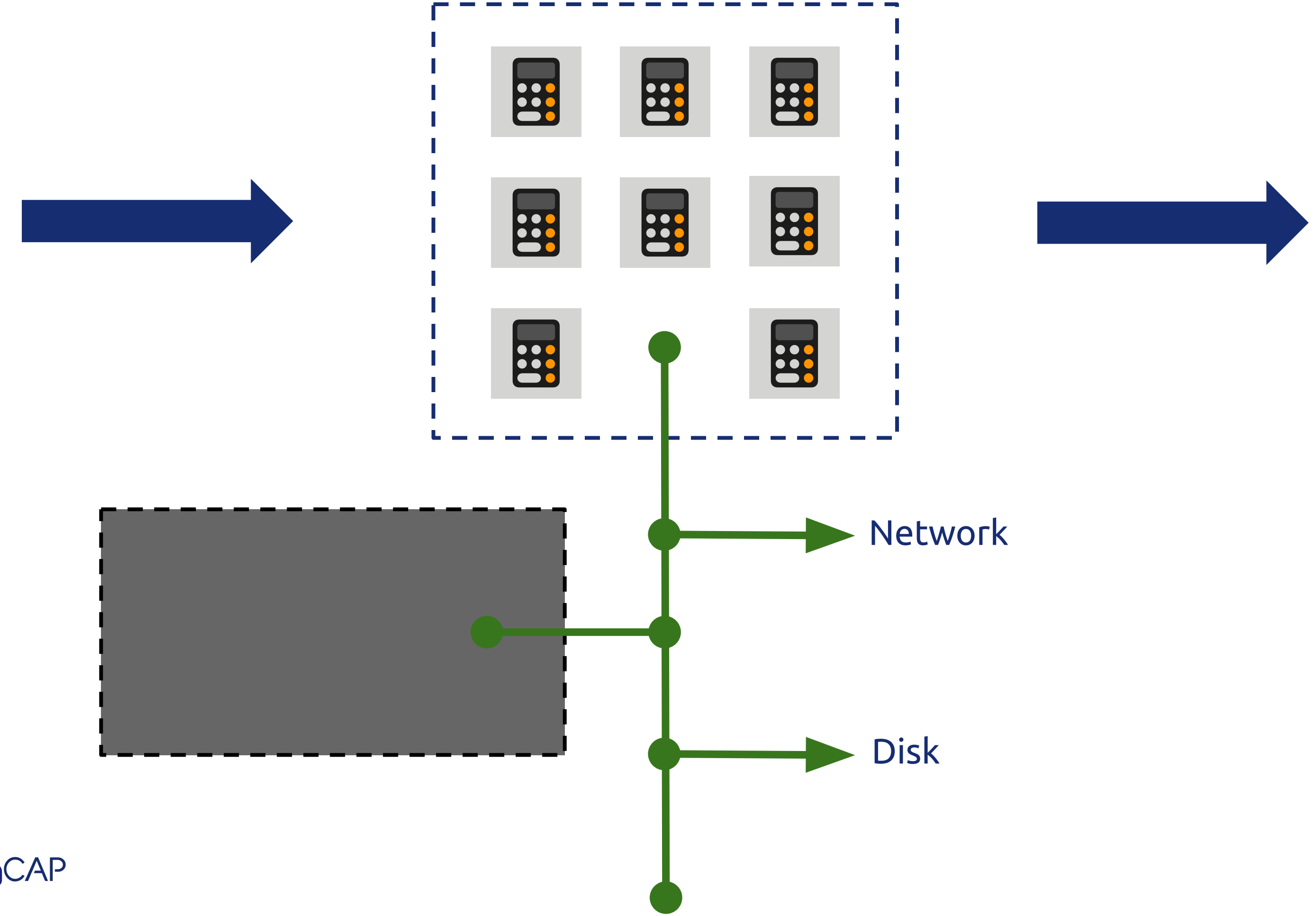
PingCAP

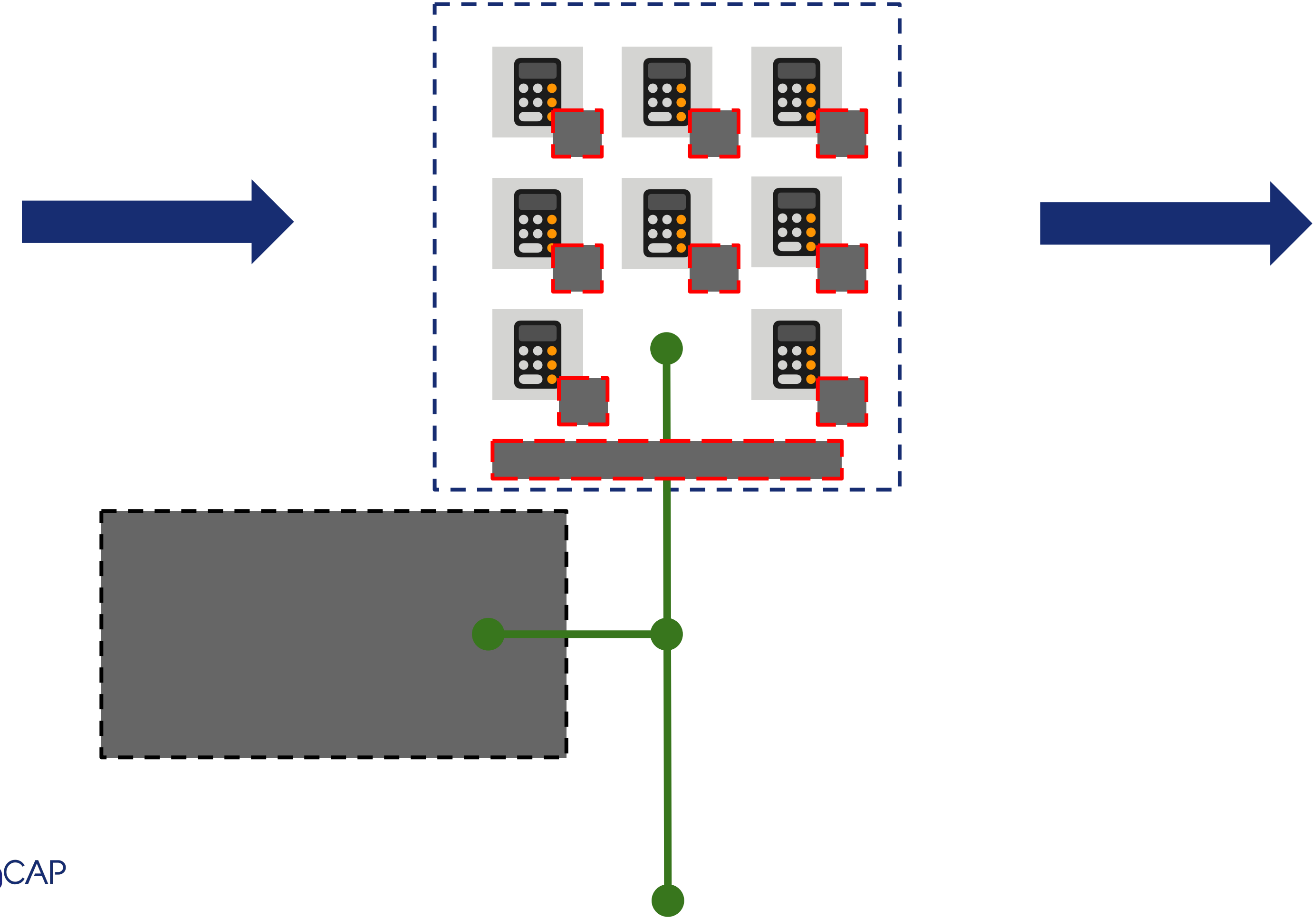
Computer architecture

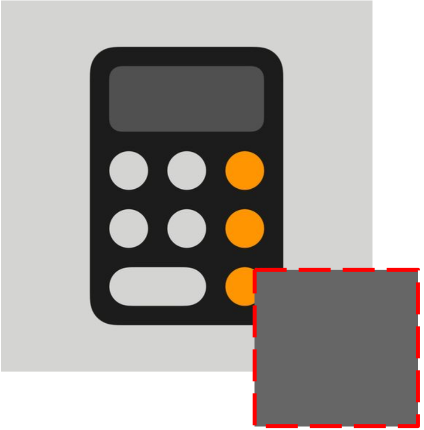
```
lea  ecx, [rsi - 1]
lea  edx, [rsi - 2]
imul rdx, rcx
shr  rdx
```

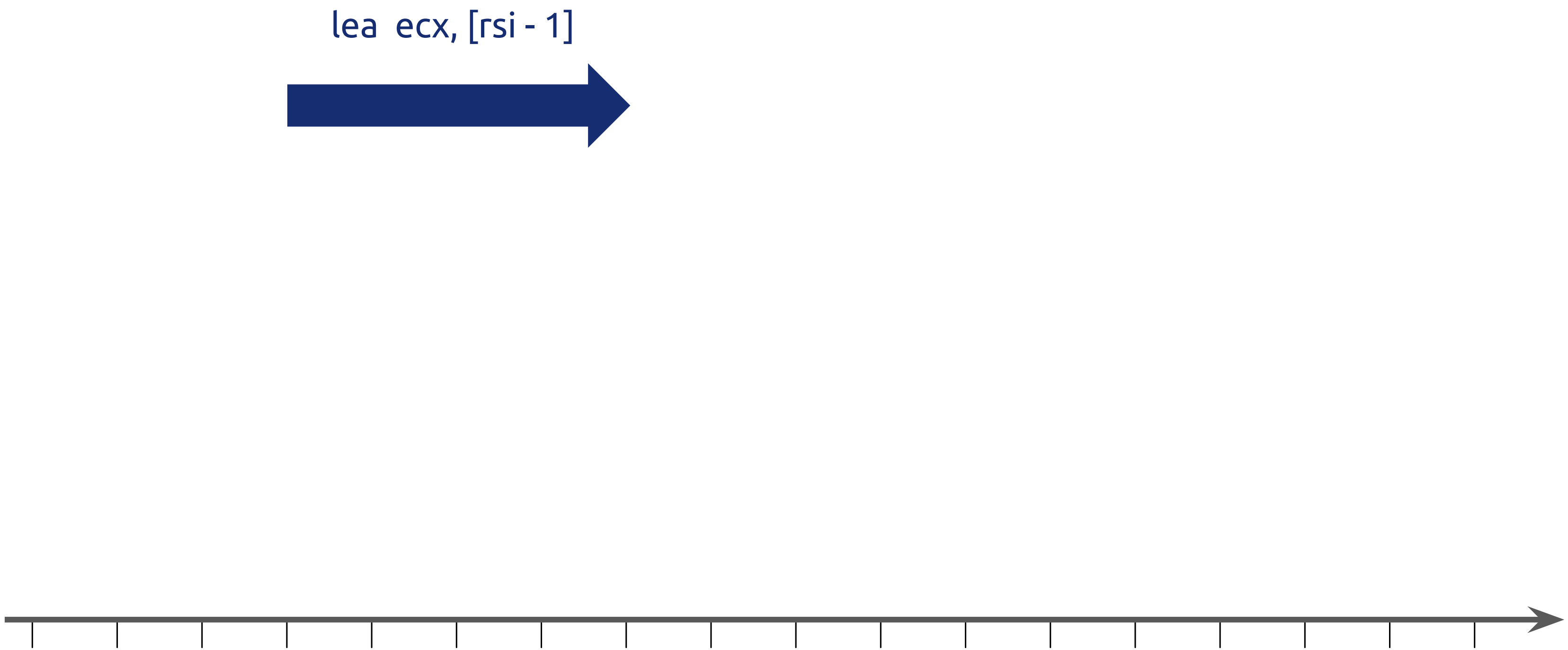
42

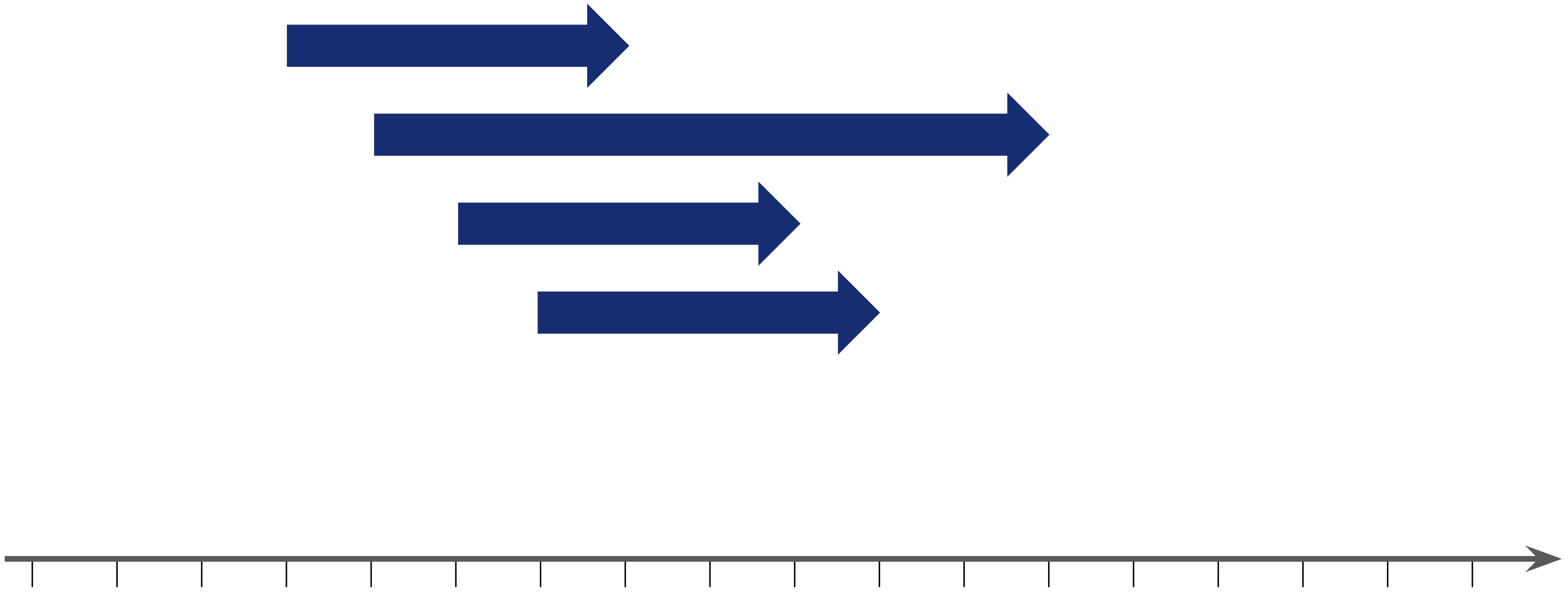


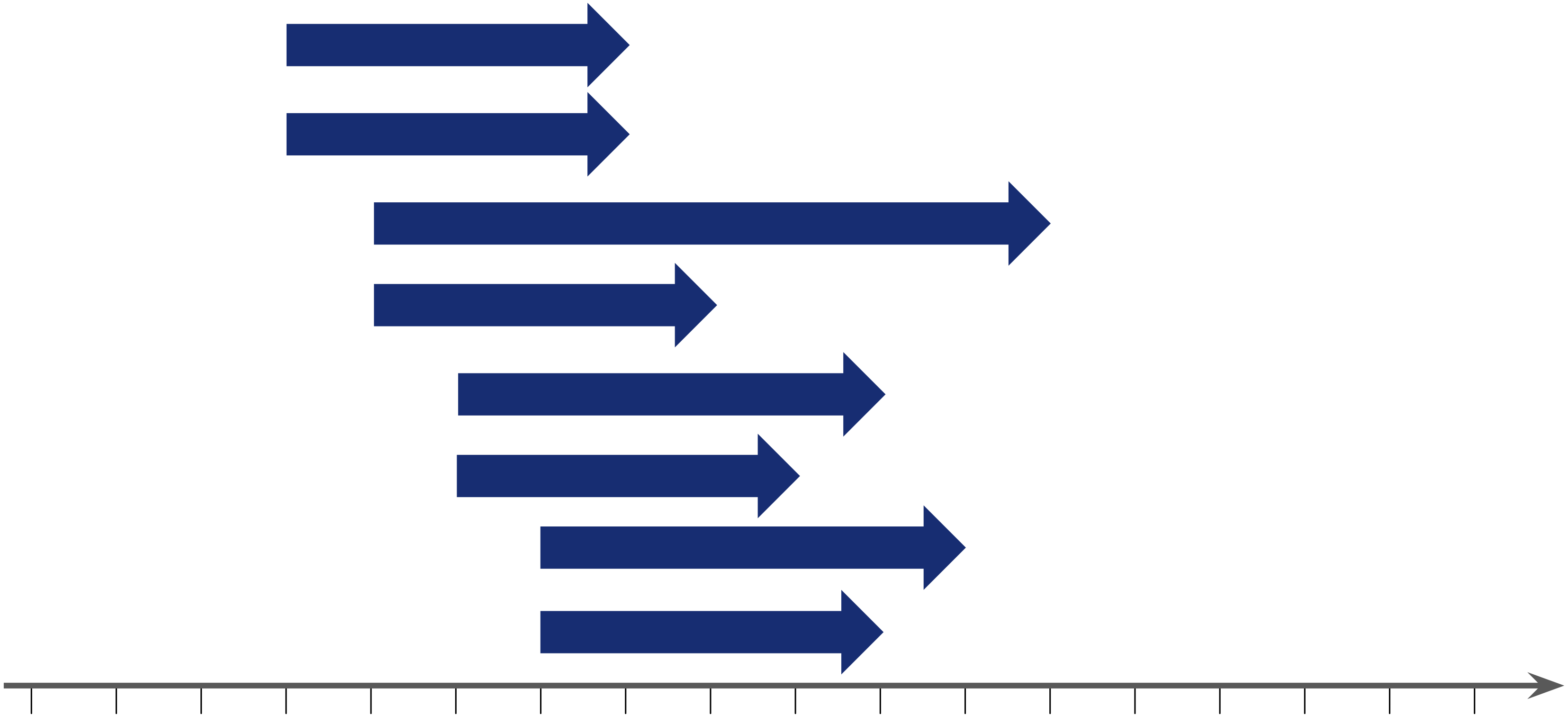

Network
Disk






lea ecx, [rsi - 1]

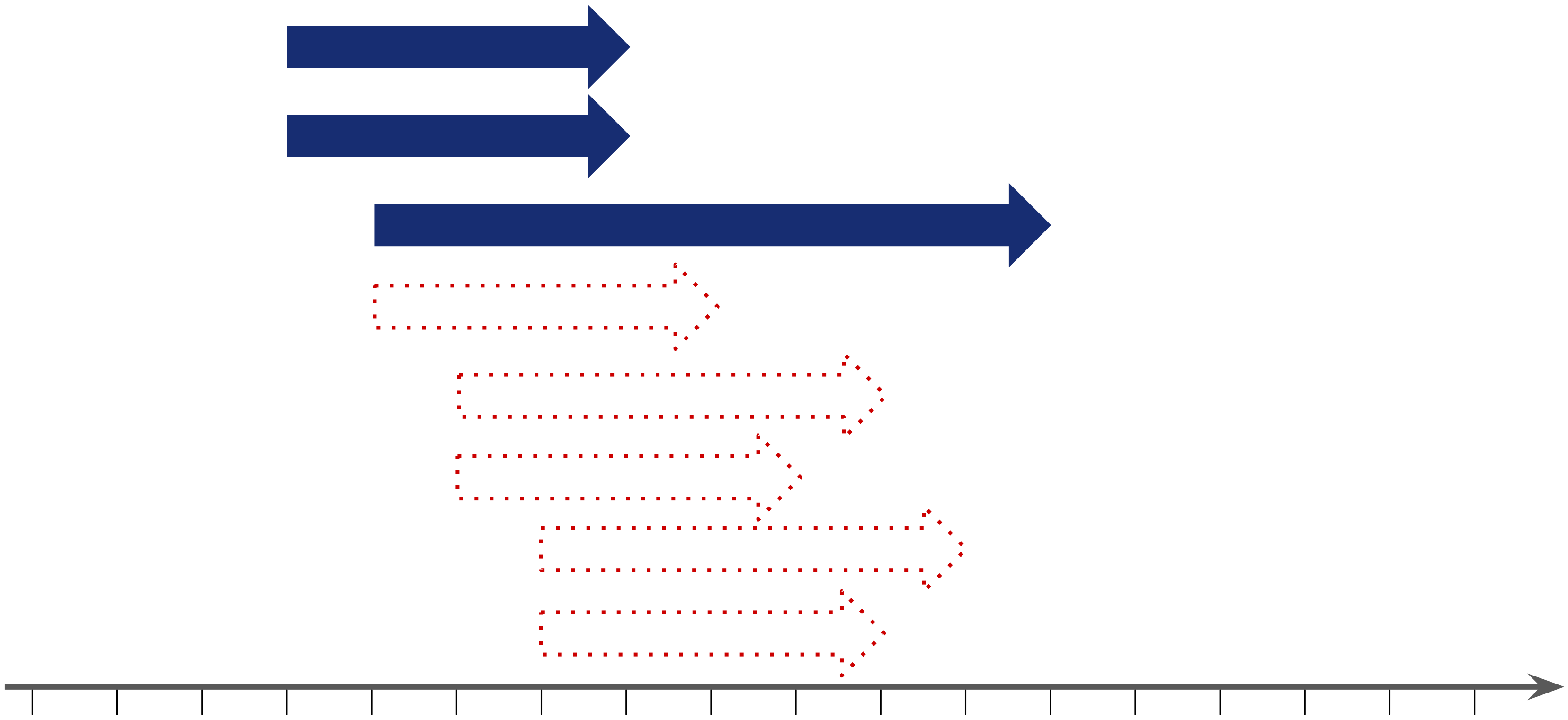


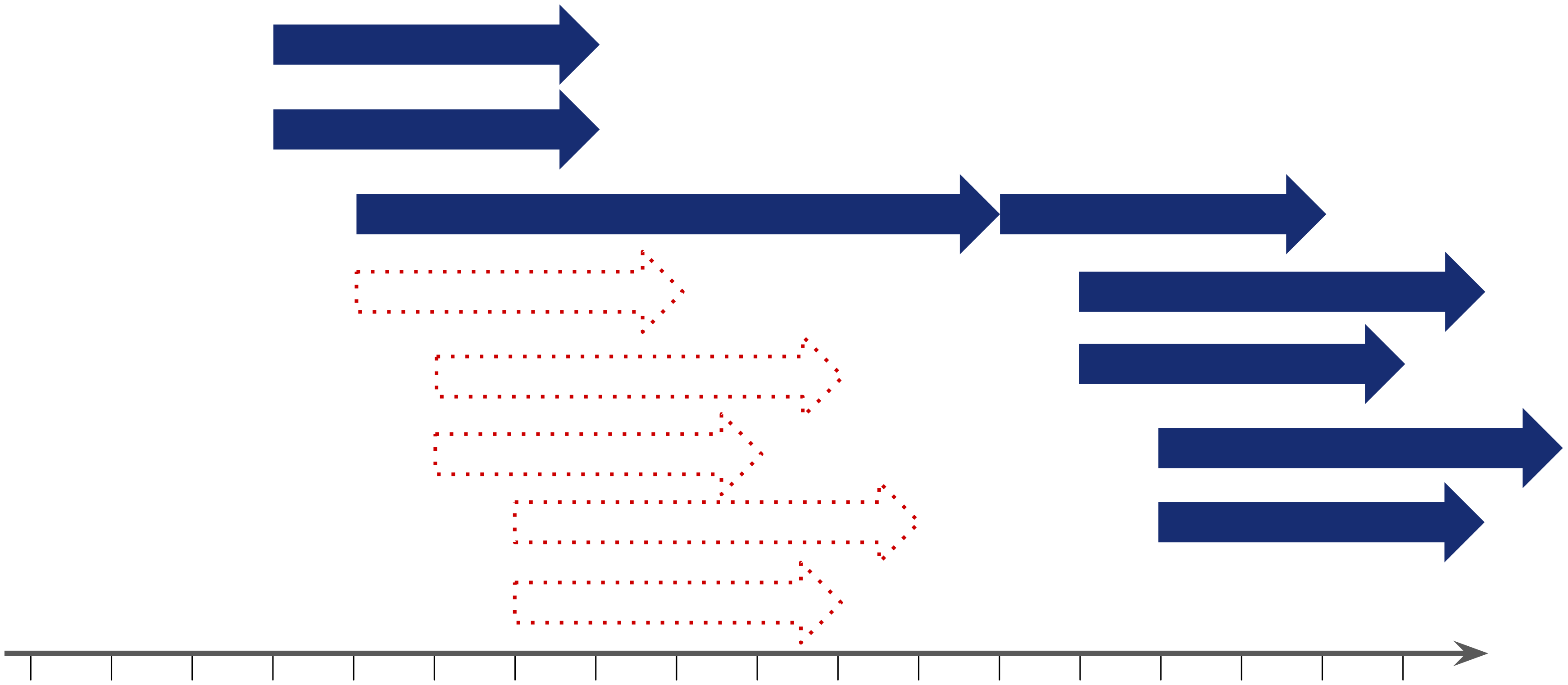


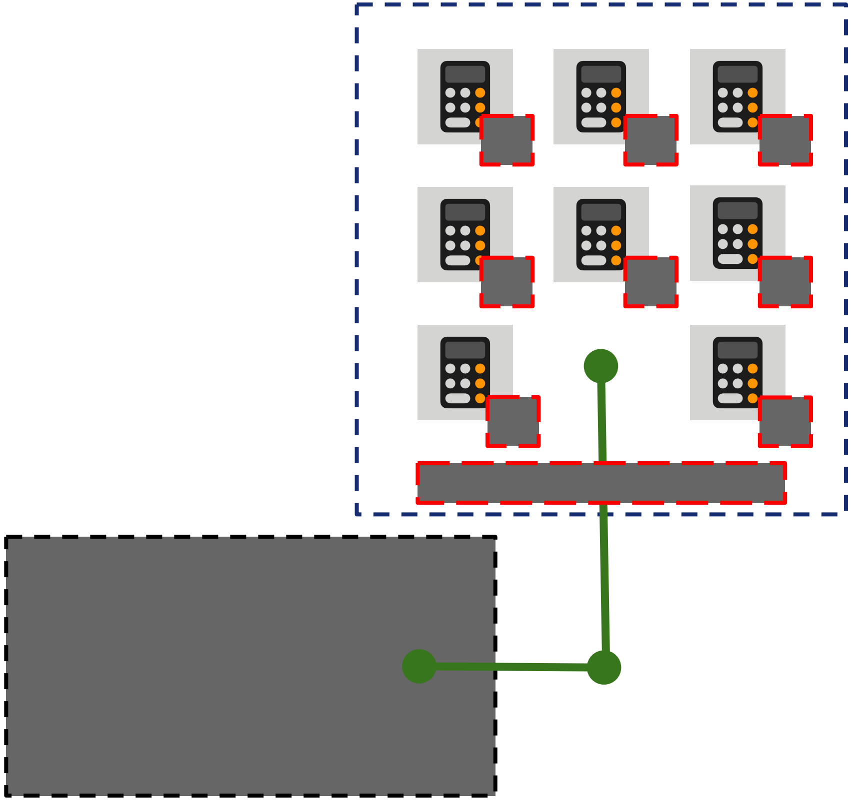
Branch prediction

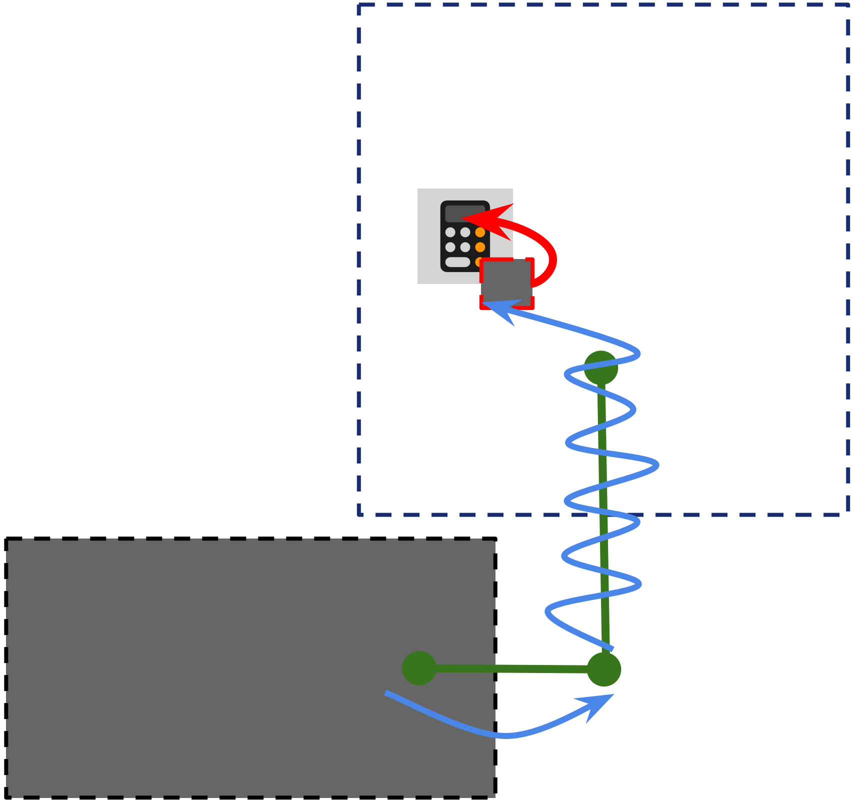
```
    lea  edx, [rsi - 2]
.Ltmp0:
    imul rdx, rcx
    shr  rdx
    test esi, edi
    je   .Ltmp0
    add  eax, ecx
```

PingCAP TiDB



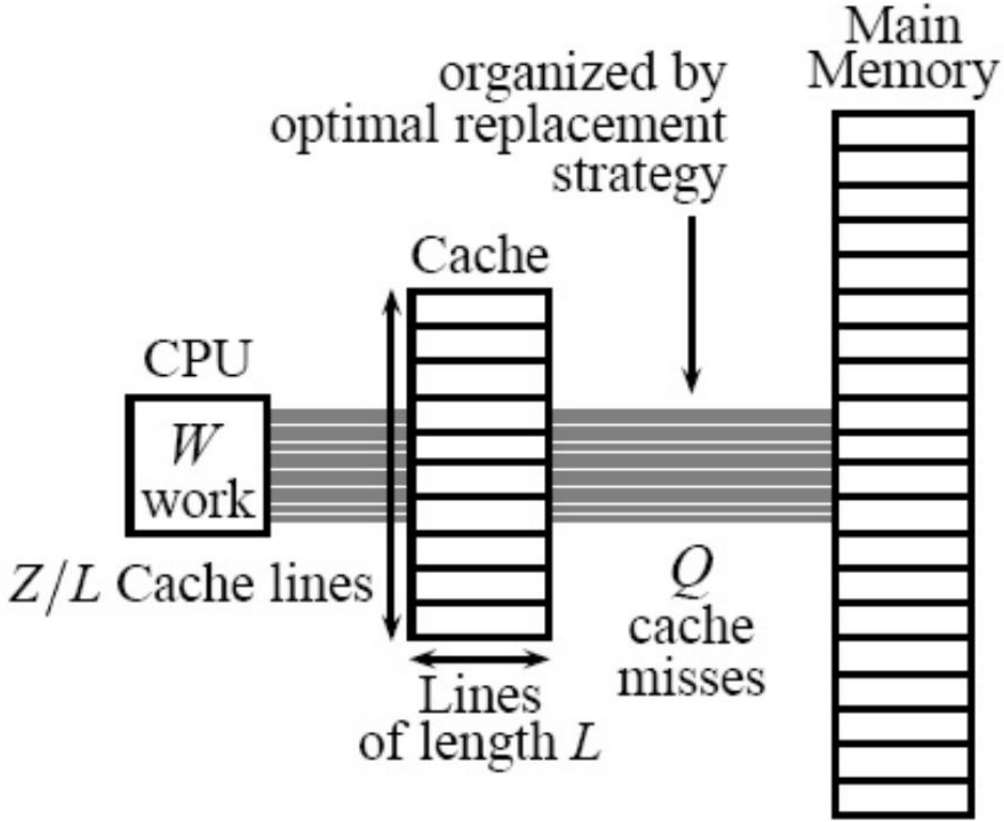






Cache rules everything around me

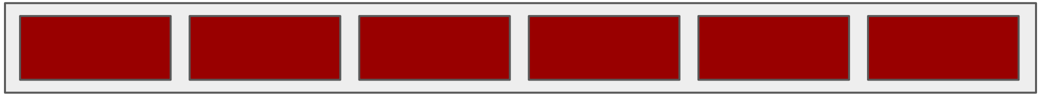
- Memory is the new IO
- Cache behaviour is the most important aspect of performance



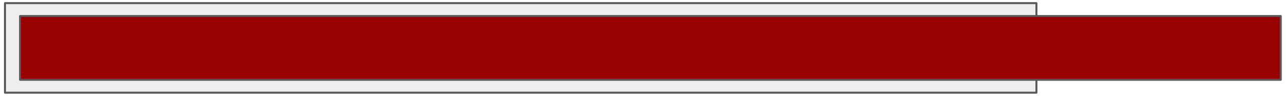
Main Memory
organized by optimal replacement strategy
Cache
CPU
W work
Z/L Cache lines
Lines of length L
Q cache misses

Cache line width

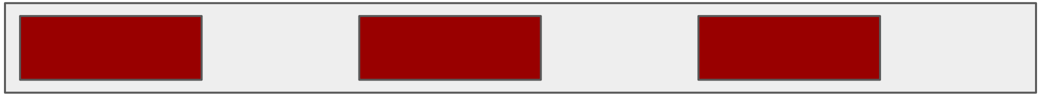
Cache line width



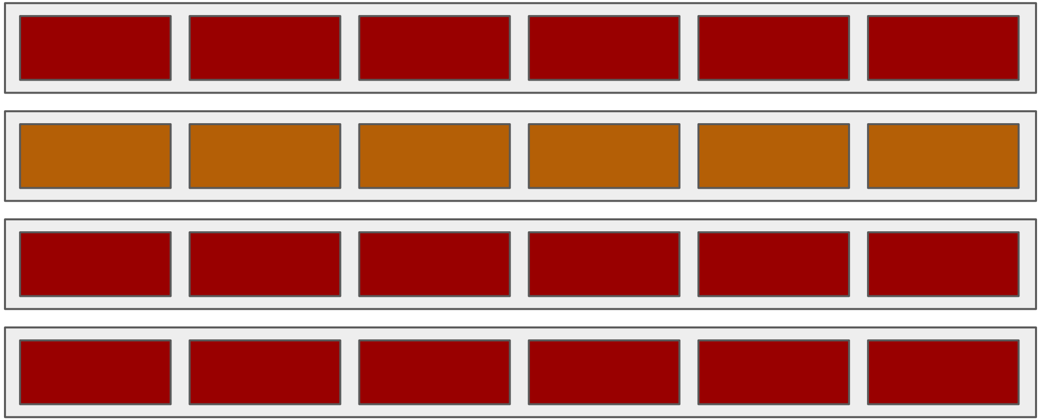
Cache line width



Cache line width

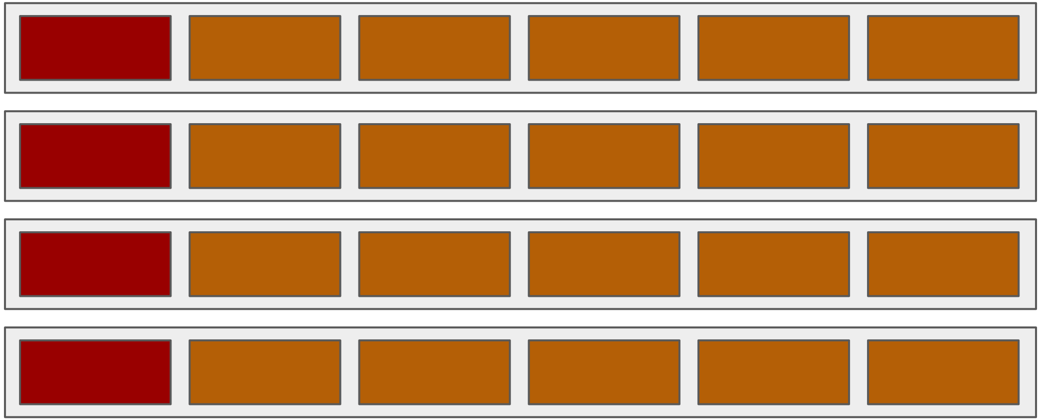


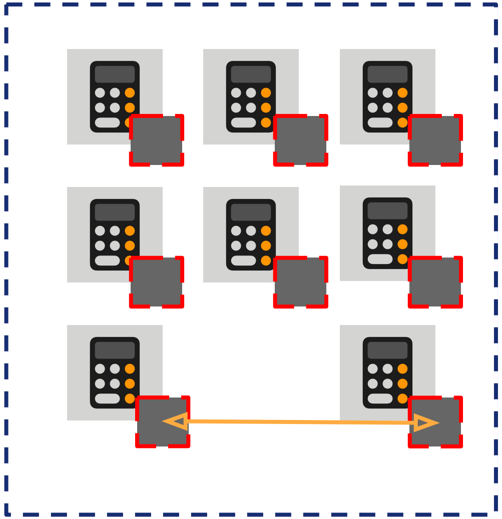
Cache line width



PingCAP TiDB

Cache line width





Optimising compilers

- `x * 4  =>  x << 2`
- `for i in 0..8 { x += i * MAGIC_NUMBER; }  =>  140`

Optimising compilers

- Inlining
- Compilation units

PingCAP

Benchmarking and profiling

Benchmarks

- You have to measure

Benchmarks

- You have to measure!

Benchmarks

- Reproducable
- Realistic

PingCAP TiDB

Benchmarks

- How much of the system are you simulating?
- What is your input data?
- Statistical issues and warm-up
- Environment
- Are you optimizing for the benchmark?

Profiling

- perf
- valgrind
- vtune
- flamegraphs
- metrics
- instrumentation

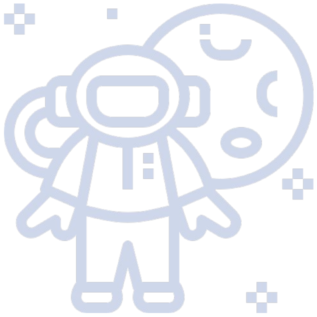
Profiling

- Indirect costs
- Wait time

Optimisation

> The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; **premature optimization is the root of all evil** (or at least most of it) in programming.

Donald Knuth



Premature optimisation

- Costs to optimisation
- You may be wrong
- Simple, clean code is easier to optimise
- CPUs and compilers are pretty smart

Premature optimisation

- Simple, clear code
- Don’t write sloppy code
  - Think about algorithms and concurrency
  - Avoid unnecessary costs
  - Don’t hide costs

Optimisation

- Measure
- Find hot code
  - Make it faster
  - Make it less hot
- Measure again

Optimisation

- Measure
- Find hot code
  - Make it faster
  - Make it less hot
- Measure again

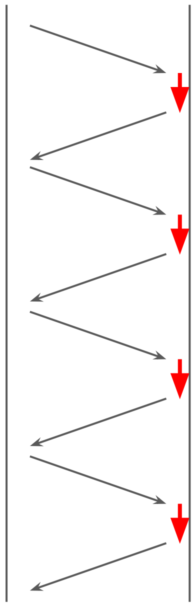
Caching

- Memoization
- Dynamic programming

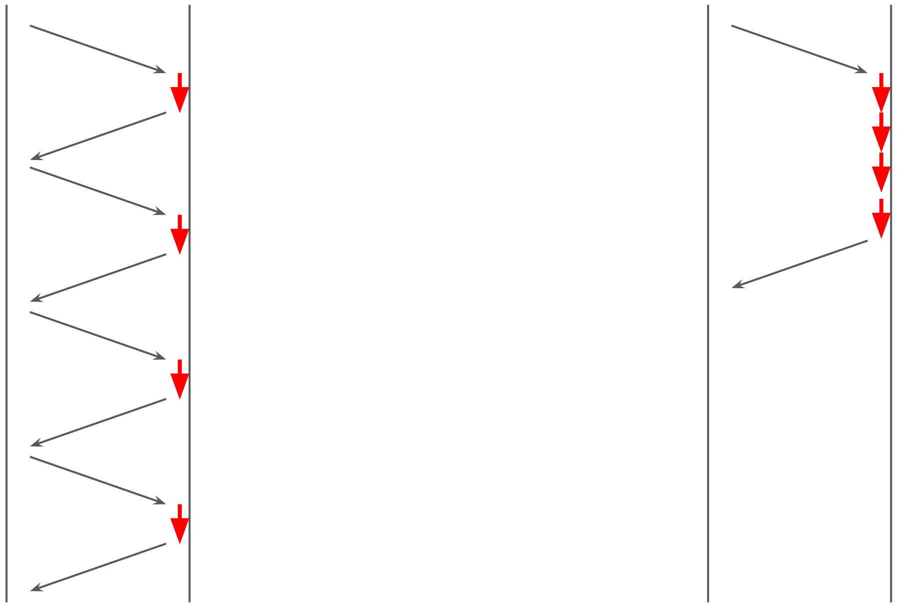
Batching

- Many operations
- Minimise overhead

Batching



Batching



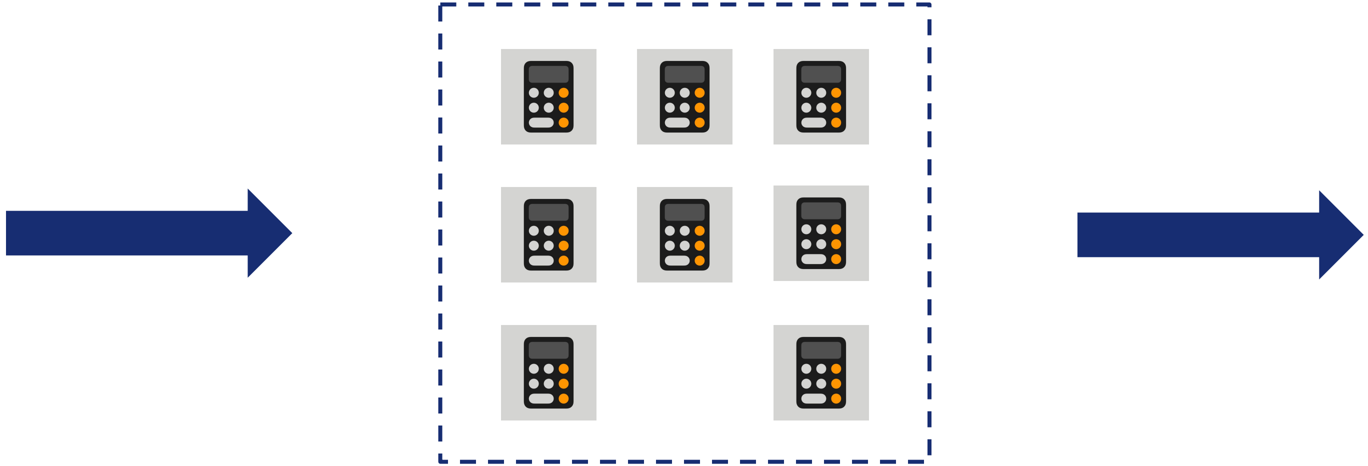
Optimisation

- Measure
- Find hot code
    - Make it faster
    - Make it less hot
- Measure again

IO

- No.

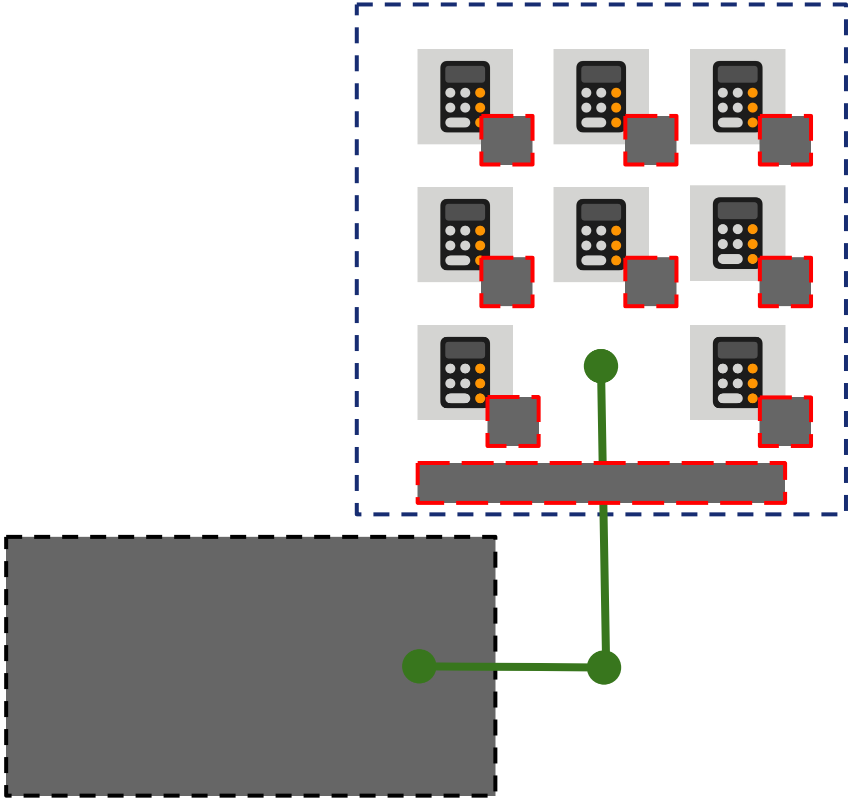
PingCAP TiDB



Concurrency

Concurrency

- No sharing is best.
- Immutable sharing is fine.
  - Split mutable state
- Atomics.
- Locks, etc.



Cache-friendly programming

- Sequential data.
- Small, tightly packed.
- Predictable code.
- Indirection.

PingCAP TiDB

Cache-friendly programming

- Arrays, slices, `Vec`
- Linked lists, graphs

Cache-friendly programming

- Data layout
- Data-oriented design

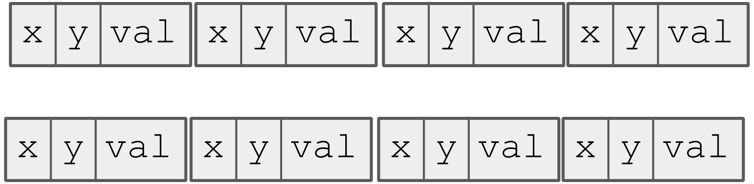
Cache-friendly programming

- Data layout
- Data-oriented design

```
struct Foo {
  x: u32,
  y: u32,
  val: u64,
}
```

Cache-friendly programming

- Data layout
- Data-oriented design

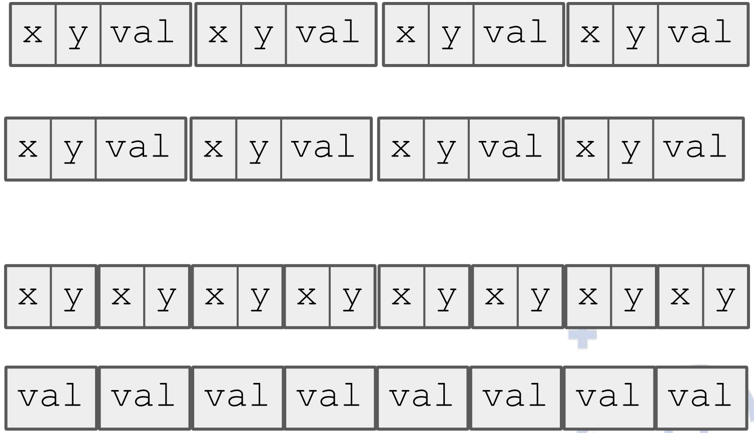



```
struct Foo {
  x: u32,
  y: u32,
  val: u64,
}
```

Cache-friendly programming

- Data layout
- Data-oriented design

```
struct Foo {
  x: u32,
  y: u32,
  val: u64,
}
```



Allocation

- Heap vs stack
  - Dynamic vs static
  - Bookeeping
  - Free
  - GC
- Cache effects
  - size trade-off

Predictable code

- Few branches per loop
- Branch the same way
- Pre-sort mixed data

PingCAP TiDB

Predictable code

```
for x in … {
    if x.abc < 0 {
        …
    }
    if foo(x.xyz) {
        …
    }
}
```

Predictable code

```
for x in … {
    if x.abc < 0 {
        …
    }
}
…
for x in … {
    if foo(x.xyz) {
        …
    }
}
```

Tuning

- OS
- compiler
- runtime
- allocator
- parameters

Summary

- **You have to measure**
- Keep it simple
- Parallelism
- Cache effects
- Allocation

PingCAP

Performant Rust

Concurrency

- TLS
- Crossbeam

Concurrency

- async is fast, but introduces complexity
- worthwhile if you have *lots* of concurrency and lots of time spent blocking on io

Allocation

- `Box`, `Vec`, `String`, ...
- `collect`, `to_string`, ...

Allocation

- Trade-off between size and indirection
- Data structures
  - hybrid data structures
  - **`with_capacity`**
- Arena allocation

Large data

- enum variants
- `repr("C")`
- `-Zprint-type-sizes`

Dispatch

- **`fn foo(x: &impl Bar)`**
- `fn foo<X: Bar>(x: &X)`

Dispatch

- **`fn foo(x: &impl Bar)`**
- `fn foo<X: Bar>(x: &X)`

- **`fn foo(x: &dyn Bar)`**
- `fn foo(x: &Bar)`

Dispatch

- `fn foo(x: &impl Bar)`
    - thin pointer
    - static dispatch
    - monomorphised

Dispatch

- `fn foo(x: &dyn Bar)`
  - fat pointer
  - virtual dispatch
  - polymorphic generated code

Dispatch

- monomorphised code
  - code bloat
  - how big is the function?
  - how many types is it used with?

Dispatch

- virtual dispatch
    - indirect call
    - is it predictable?

Data types

- There are many variations
  - different structgures/algorithms
  - different implementations
  - different constraints
- Don’t pay for what you don’t need
  - Cryptographic hash
  - Secure random numbers
  - ...

PingCAP TiDB

Run time and compile time

- Bounds checks are expensive
  - use iteration
  - if you can’t iterate, assert bounds up front

Run time and compile time

- Iterators
  - `chain`
  - `retain`, `partition`, ...
  - `collect`

PingCAP TiDB

Run time and compile time

- `Arc` and `Rc`
- `Mutex` and `RefCell`
- overflow checks

Run time and compile time

- Compile time
  - is free! (-ish)
  - const fn
  - procedural macros, build scripts
- `assert` and `debug_assert`

Run time and compile time

- Compile time
  - compile times
  - code bloat
  - macro-generated code

The compiler, etc

- rustc and llvm
- Inlining
  - `#[inline]`
  - recursion
- Panicking
  - overheads
  - panic = abort

The compiler, etc

- SIMD

The compiler, etc

- LTO
- `#[bench]`, Criterion
- `-Z`

PingCAP

Thank You !

谢谢

PingCAP

Remember to measure 😉