When a Microsecond Is an Eternity: High Performance Trading Systems in C++

Carl Cook, Ph.D.

Optiver

Introduction

About me:

- Software developer for several trading/finance companies
- Member of ISO SG14 (gaming, low latency, trading), but not a C++ expert

Today's talk:

- What is electronic market making?
- Technical challenges
- Techniques for low latency C++, and then some surprises
- Measurement of performance

Disclaimer: This is not a discussion covering every C++ optimization technique - it's a quick sampler into the life of developing high performance trading systems

The three minute guide to electronic market making

“The elements of good trading are: (1) cutting losses, (2) cutting losses, and (3) cutting losses. If you can follow these three rules, you may have a chance.”

– Ed Seykota

- Two main activities:
  - Provide (continually updating) prices to the market
  - Spot profitable opportunities when they arise
- Objectives:
  - Make small, profitable, trades regularly
  - Avoid making large bad trades
- Aside from accurate pricing, successful algorithms are the fastest to:
  - Buy low
  - Sell high
- Success means being *[any unit of time]* faster than the competition

Photo by Alvin Loke / CC BY 2.0

Safety first

If anything appears to be wrong:

- Pull all orders, then start asking questions - not the other way round
- A lot could happen in a few seconds in an uncontrolled system

The best approach is to automate detection of failures

Technical challenges

"If you're not at all interested in performance, shouldn't you be in the Python room down the hall?"

– Scott Meyers

- The “hotpath” is only exercised 0.01% of the time - the rest of the time the system is idle or doing administrative work
- Operating systems, networks and hardware are focused on throughput and fairness
- Jitter is unacceptable - it means bad trades

The role of C++

From Bjarne Stroustrup:

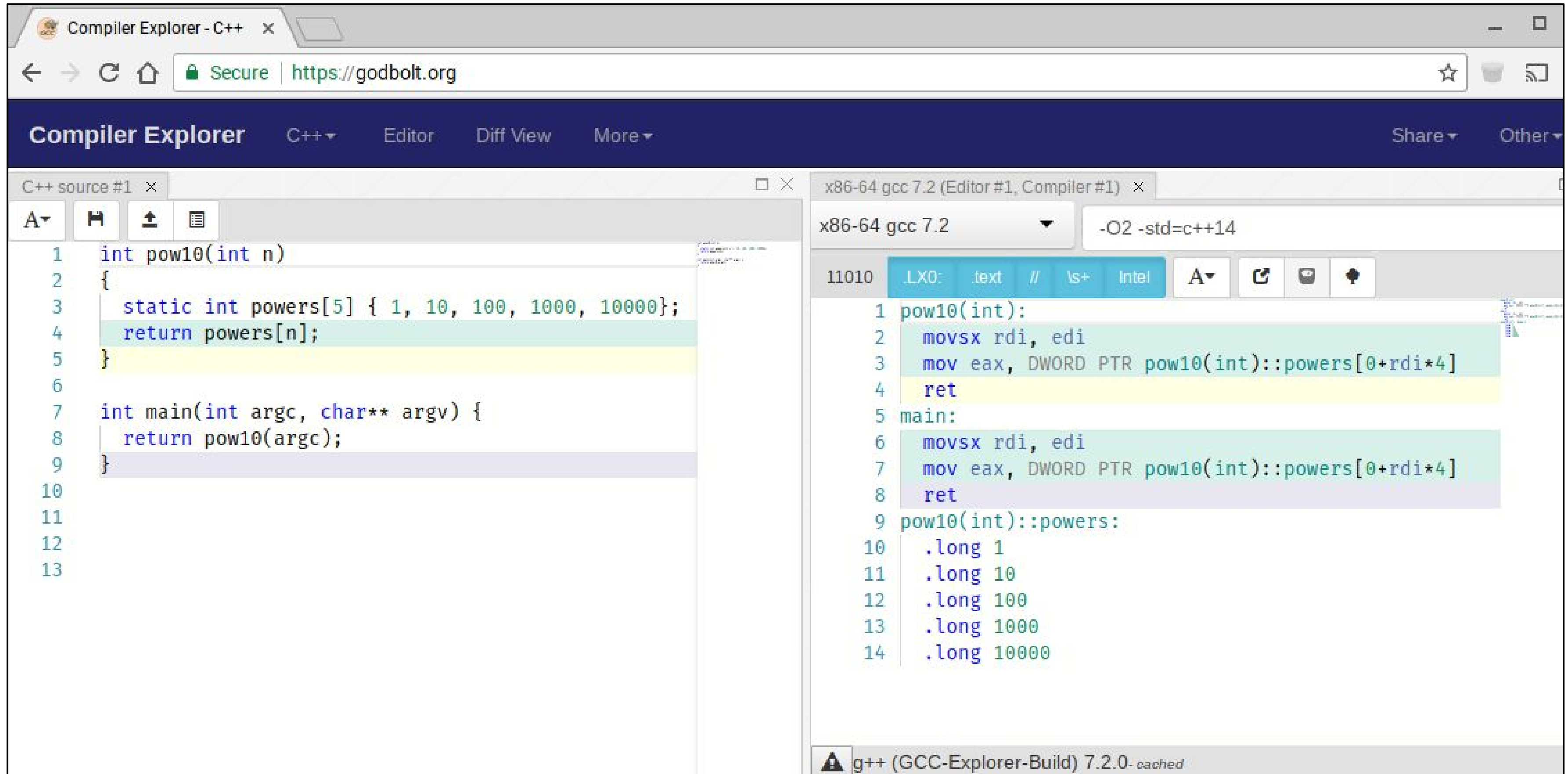
> *“C++ enables zero-overhead abstraction to get us away from the hardware without adding cost”*

However, even though C++ is good at saying what will be done, there are still other factors:

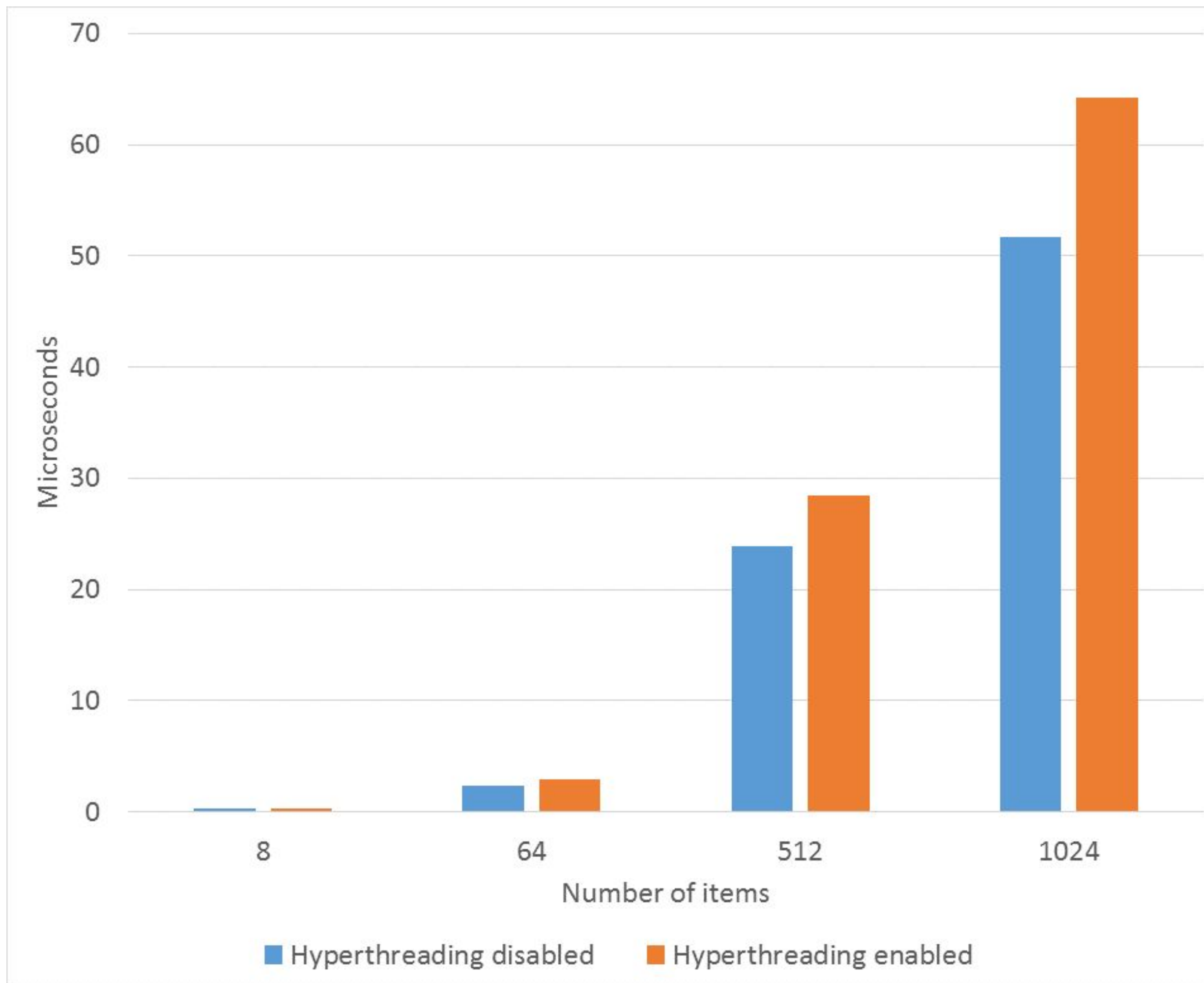
- Compiler (and version)
- Machine architecture
- 3rd party libraries
- Build and link flags

We need to check what C++ is doing in terms of machine instructions...

… luckily there's an app for that:



Calling `std::sort` on a `std::vector<int>`



Same:

- Hardware
- Operating system
- Binary
- Background load

One server is tuned for production (no hyper threading, etc), the other not

How fast is fast?

Burj Khalifa

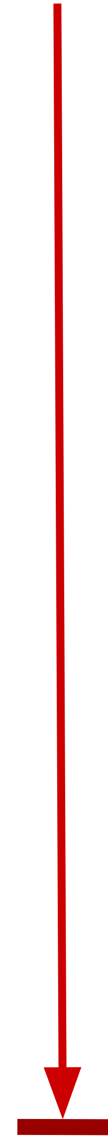
Height:
828 meters
2,722 feet

Speed of light:
~ 1 foot per ns

Photo by Donald Tong / CC BY-SA 3.0

A very good minimum time (wire to wire) for a software-based trading system is around 2.5us

That’s less than the time it takes light to travel from the top of the spire to the ground



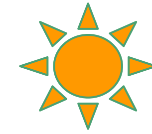


Photo by Donald Tong / CC BY-SA 3.0

Low latency programming techniques

"When in doubt, use brute force."

– Ken Thompson

Slowpath removal

Avoid this:

```
if (checkForErrorA())
  handleErrorA();
else if (checkForErrorB())
  handleErrorB();
else if (checkForErrorC())
  handleErrorC();
else
  sendOrderToExchange();
```

Aim for this:

```
int64_t errorFlags;
  ...
if (!errorFlags)
  sendOrderToExchange();
else
  HandleError(errorFlags);
```

Tip: ensure that error handling code will not be inlined

Template-based configuration

- It’s convenient to have some things controlled via configuration files
    - However virtual functions (and even simple branches) can be expensive
- One possible solution:
    - Use templates (often overlooked, even though everyone uses the STL)
    - This removes branches, eliminates code that won’t be executed, etc

```cpp
// 1st implementation
struct OrderSenderA {
  void SendOrder() {
    ...
  }
};
```

```cpp
// 2nd implementation
struct OrderSenderB {
  void SendOrder() {
    ...
  }
};
```

```cpp
template <typename T>
struct OrderManager : public IOrderManager {
  void MainLoop() final {
    // ... and at some stage in the future...
    mOrderSender.SendOrder();
  }
  T mOrderSender;
};
```

```cpp
std::unique_ptr<IOrderManager> Factory(const Config& config) {
  if (config.UseOrderSenderA())
    return std::make_unique<OrderManager<OrderSenderA>>();
  else
    return std::make_unique<OrderManager<OrderSenderB>>();
}

int main(int argc, char *argv[]) {
  auto manager = Factory(config);
  manager->MainLoop();
}
```

Lambda functions are fast and convenient

If you know at compile time which function is to be executed, then prefer lambdas

```
template <typename T>
void SendMessage(T&& lambda) {
  Msg msg = PrepareMessage();
  lambda(msg);
  send(msg);
}
```

```
SendMessage([&](auto& msg) {
  msg.instrument = x;
  msg.price = z;
    ...
});
```

Memory allocation

- Allocations are costly:
  - Use a pool of preallocated objects
- Reuse objects instead of deallocating:
  - `delete` involves no system calls (memory is not given back to the OS)
    - But: glibc `free` has 400 lines of book-keeping code
  - Reusing objects helps avoid memory fragmentation as well
- If you must delete large objects, consider doing this from another thread

Exceptions in C++

- Don't be afraid to use exceptions (if using `gcc`, `clang`, `msvc`):
    - I've measured this in quite some detail:
        - They are zero cost if they don't throw
- Don't use exceptions for control flow:
    - That will get expensive:
        - My benchmarking suggests an overhead of at least 1.5us
    - Your code will look terrible

Prefer templates to branches

Branching approach:

```
enum class Side { Buy, Sell };

void RunStrategy(Side side) {
  const float orderPrice = CalcPrice(side, fairValue, credit);
  CheckRiskLimits(side, orderPrice);
  SendOrder(side, orderPrice);
}

float CalcPrice(Side side, float value, float credit) {
  return side == Side::Buy ? value - credit : value + credit;
}
```

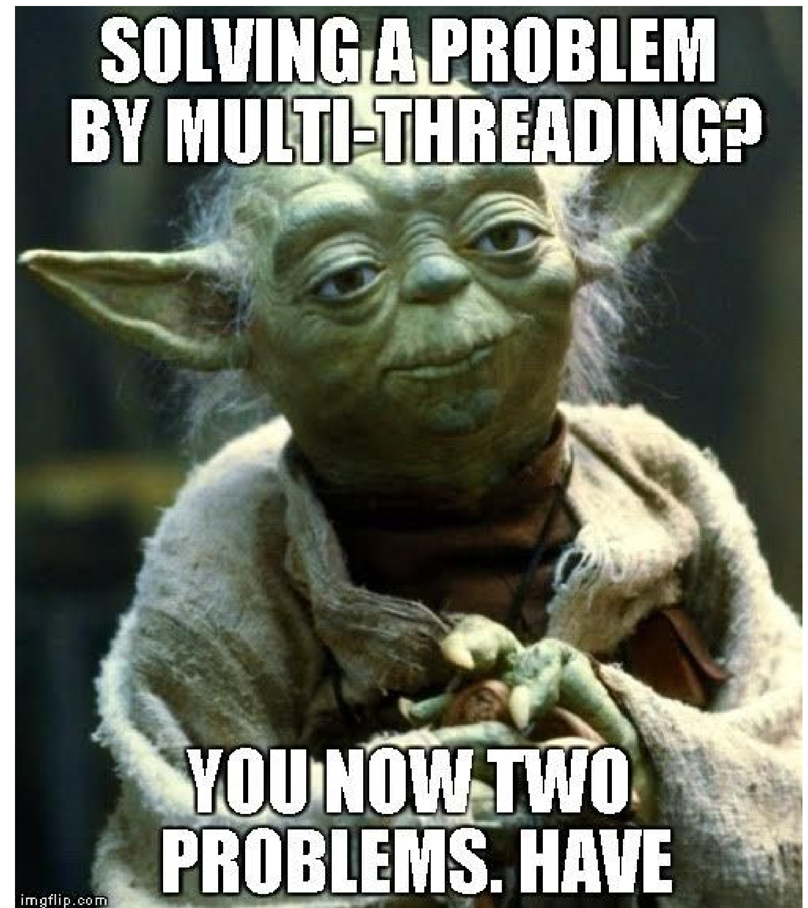
Templated approach:

```
template<Side T>
void Strategy<T>::RunStrategy() {
  const float orderPrice = CalcPrice(fairValue, credit);
  CheckRiskLimits(orderPrice);
  SendOrder(orderPrice);
}
template<>
float Strategy<Side::Buy>::CalcPrice(float value, float credit) {
  return value - credit;
}
template<>
float Strategy<Side::Sell>::CalcPrice(float value, float credit) {
  return value + credit;
}
};
```

Multi-threading

Multithreading is best avoided for latency-sensitive code:

- Synchronization of data via locking will get expensive
- Lock free code may still require locks at the hardware level
- Mind-bendingly complex to correctly implement parallelism
- Easy for the producer to accidentally saturate the consumer



If you must use multiple threads...

- Keep shared data to an absolute minimum
    - Multiple threads writing to the same cacheline will get expensive
- Consider passing copies of data rather than sharing data
    - e.g. a single writer, single reader lock free queue
- If you have to share data, consider not using synchronization, e.g.:
    - Maybe you can live with out-of-sequence updates

Data lookups

Software engineering textbooks typically suggest:

```
struct Market {
  int32_t id;
  char shortName[4];
  int16_t quantityMultiplier;
    ...
}
```

```
struct Instrument {
  float price;
    ...
  int32_t marketId;
}
```

```
Message orderMessage;
orderMessage.price = instrument.price;
Market& market = Markets.FindMarket(instrument.marketId);
orderMessage.qty = market.quantityMultiplier * qty;
...
```

Data lookups

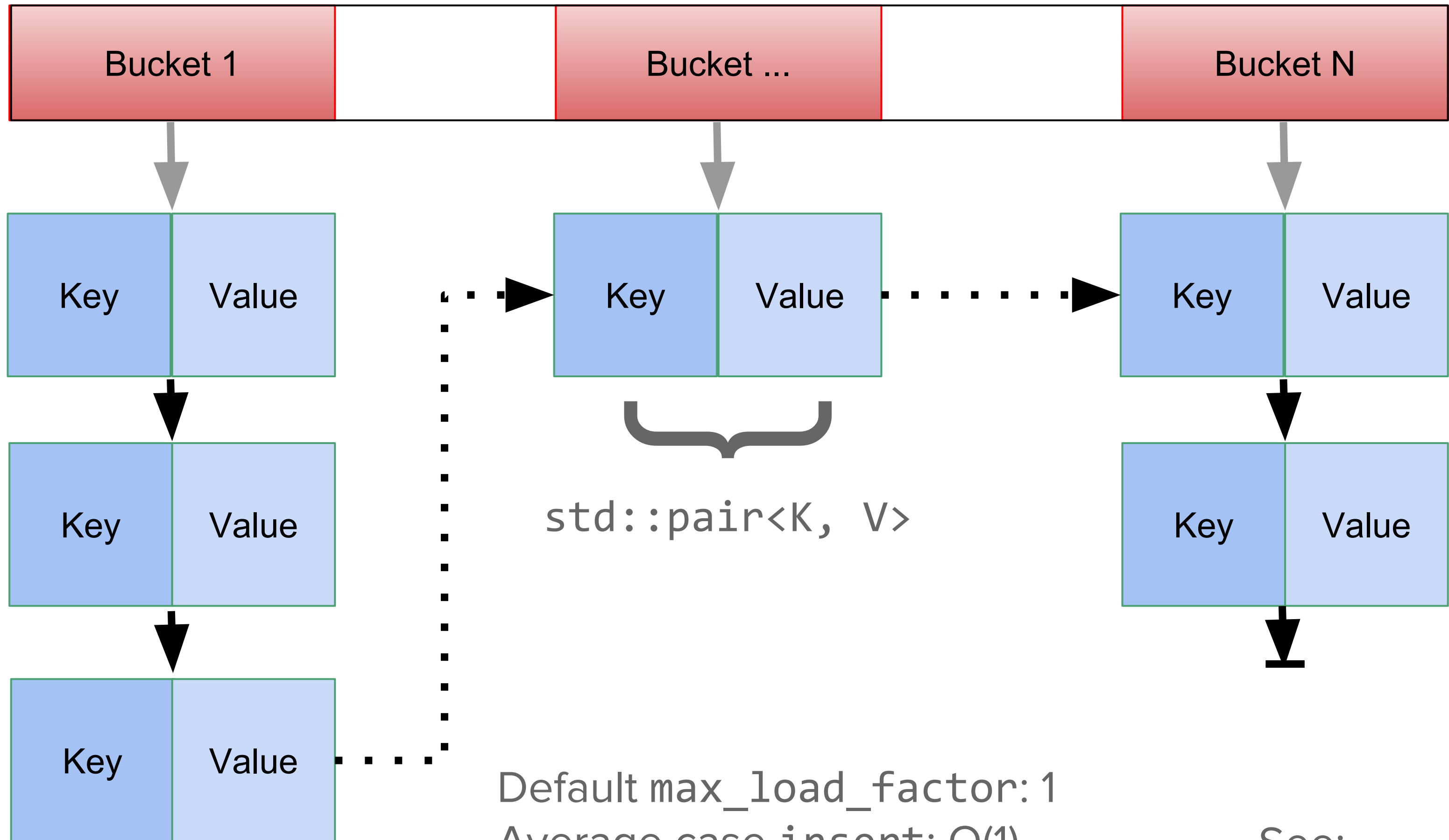
Actually, denormalized data is not a sin. Why not just pull all the data you care about in the same cacheline?

```
struct Market {
  int32_t id;
  char shortName[4];
  int16_t quantityMultiplier;
    ...
}
```

```
struct Instrument {
  float price;
  int16_t quantityMultiplier;
    ...
  int32_t marketId;
}
```

This is better than trampling your cache to “save memory”

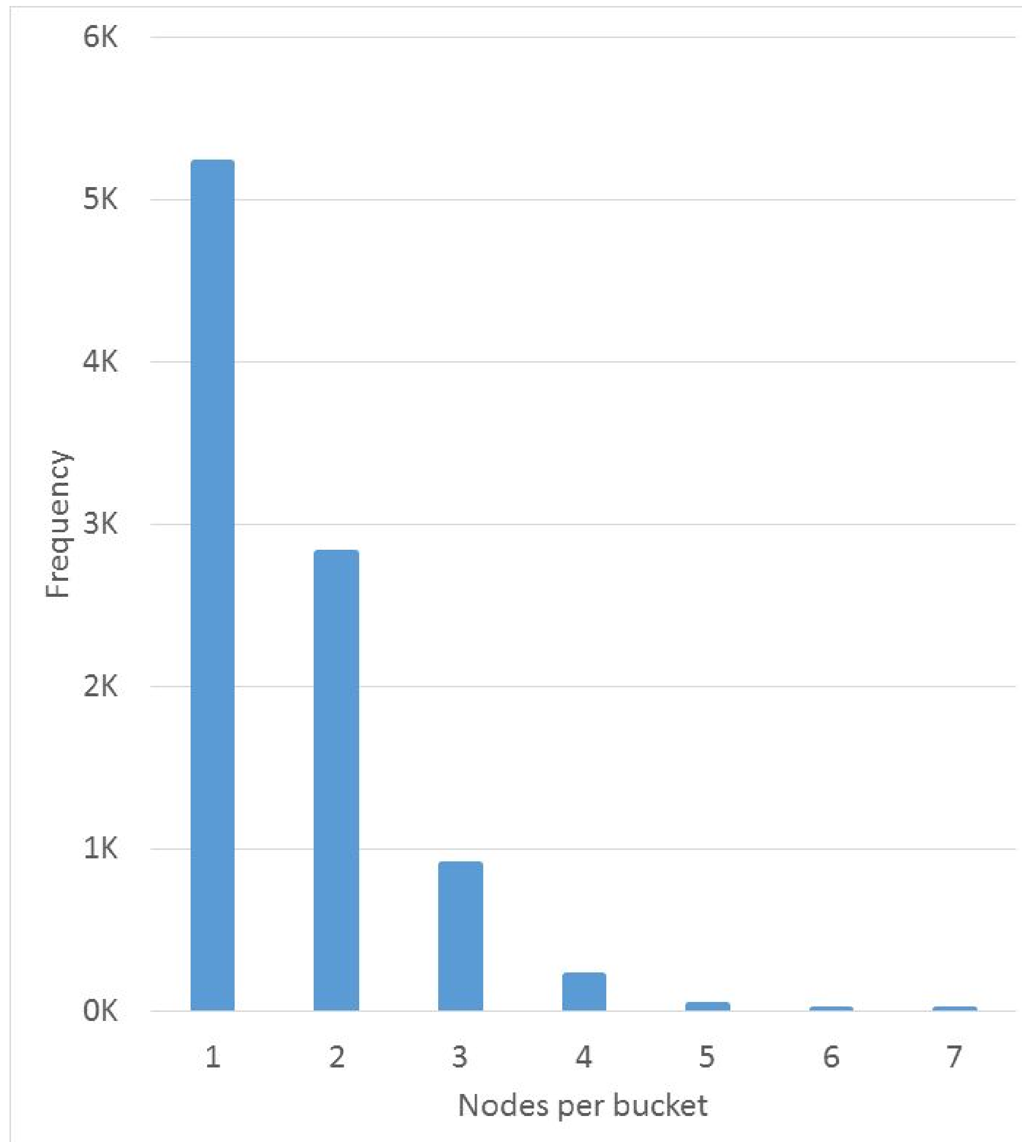
Fast associative containers (`std::unordered_map`)



Default `max_load_factor`: 1
Average case `insert`: O(1)
Average case `find`: O(1)

See:
wg21.link/n1456

10K elements, keyed in the range `std::uniform_int_distribution(0, 1e+12)`



Complexity of `find`:

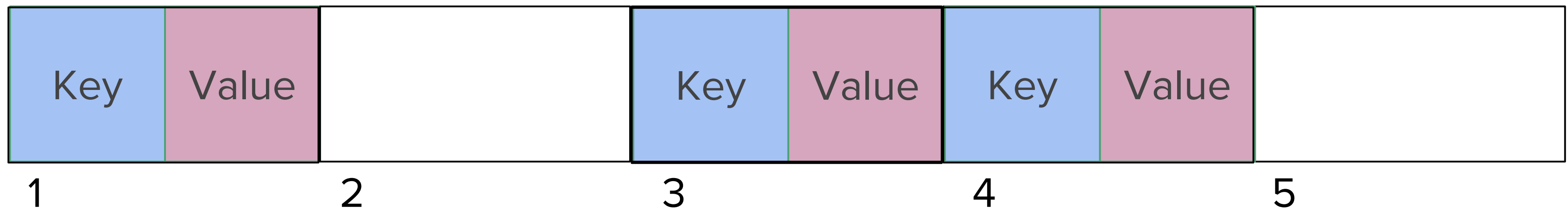
Average case: O(1)

Worst case: O(N)

```
Run on (32 X 2892.9 MHz CPU s), 2017-09-08 11:39:44
Benchmark                                       Time
----------------------------------------------------
FindBenchmark<unordered_map>/10                14 ns
FindBenchmark<unordered_map>/64                16 ns
FindBenchmark<unordered_map>/512               16 ns
FindBenchmark<unordered_map>/4k                20 ns
FindBenchmark<unordered_map>/10k               24 ns
----------------------------------------------------
```

```
                             #    56.54%  frontend cycles idle
                             #    21.61%  backend  cycles idle
                             #     0.67   insns per cycle
                             #     0.84   stalled cycles per insn
branch-misses                #     0.63%  of all branches
cache-misses                 #     0.153% of all cache refs
```

Alternatively, consider open addressing, e.g. Google's `dense_hash_map`



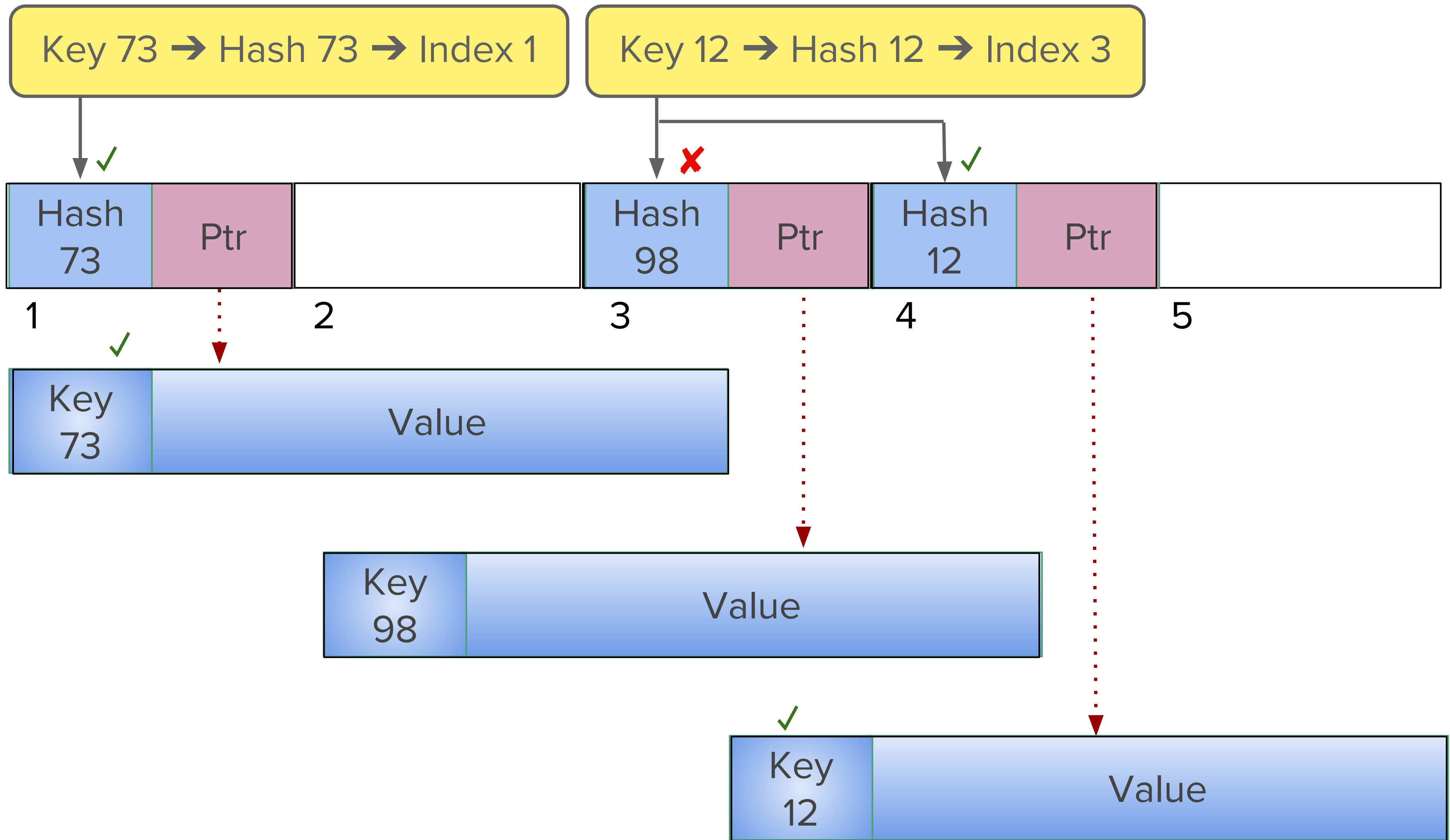
✓ Key/Value pairs are in contiguous memory - no pointer following between nodes

✘ Complexity around collision management

A lesser-known approach: a hybrid of both chaining and open addressing

Goals:

- Predictable cache access patterns (no jumping all over the place)
- Prefetched candidate hash values



It’s possible to implement this as a drop-in substitute for `std::unordered_map`

```
Run on (32 X 2892.9 MHz CPU s), 2017-09-08 11:40:08
Benchmark                                        Time
----------------------------------------------------
FindBenchmark<array_map>/10                      7 ns
FindBenchmark<array_map>/64                      7 ns
FindBenchmark<array_map>/512                     7 ns
FindBenchmark<array_map>/4k                      9 ns
FindBenchmark<array_map>/10k                     9 ns
----------------------------------------------------


                                #    38.26%  frontend cycles idle
                                #     6.77%  backend  cycles idle
                                #     1.6    insns per cycle
                                #     0.24   stalled cycles per insn
branch-misses                   #     0.22%  of all branches
cache-misses                    #     0.067% of all cache refs
```

((always_inline)) and ((noinlne))

- The `inline` keyword is somewhat misunderstood
  - It mainly means: multiple definitions are permitted
- `((always_inline))` and `((noinline))` are a stronger hint to the compiler
  - But be careful: measure

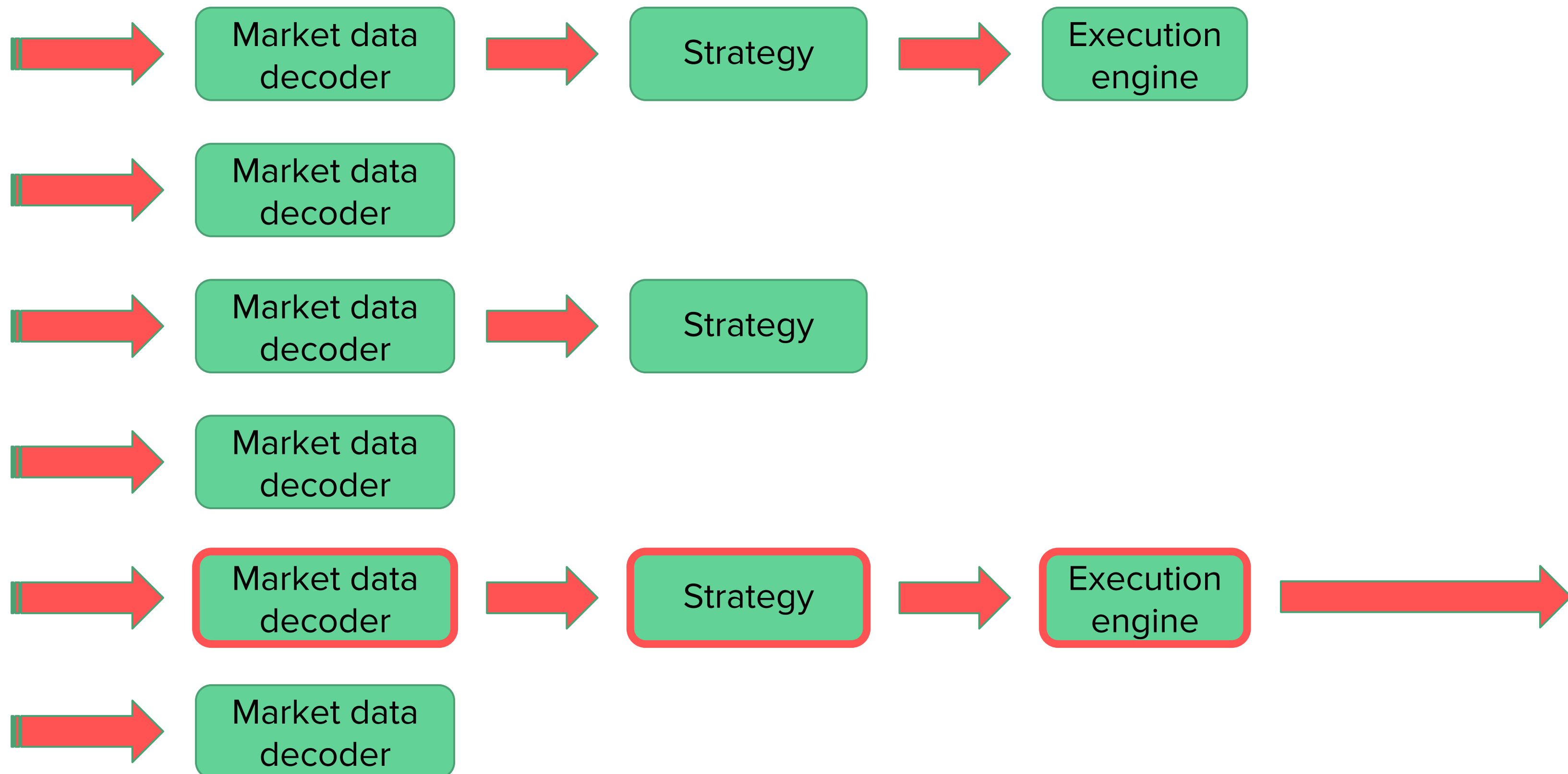
An example: forcing methods to be not inlined

```
CheckMarket();
if (notGoingToSendAnOrder)
  ComplexLoggingFunction();
else
  SendOrder();
```

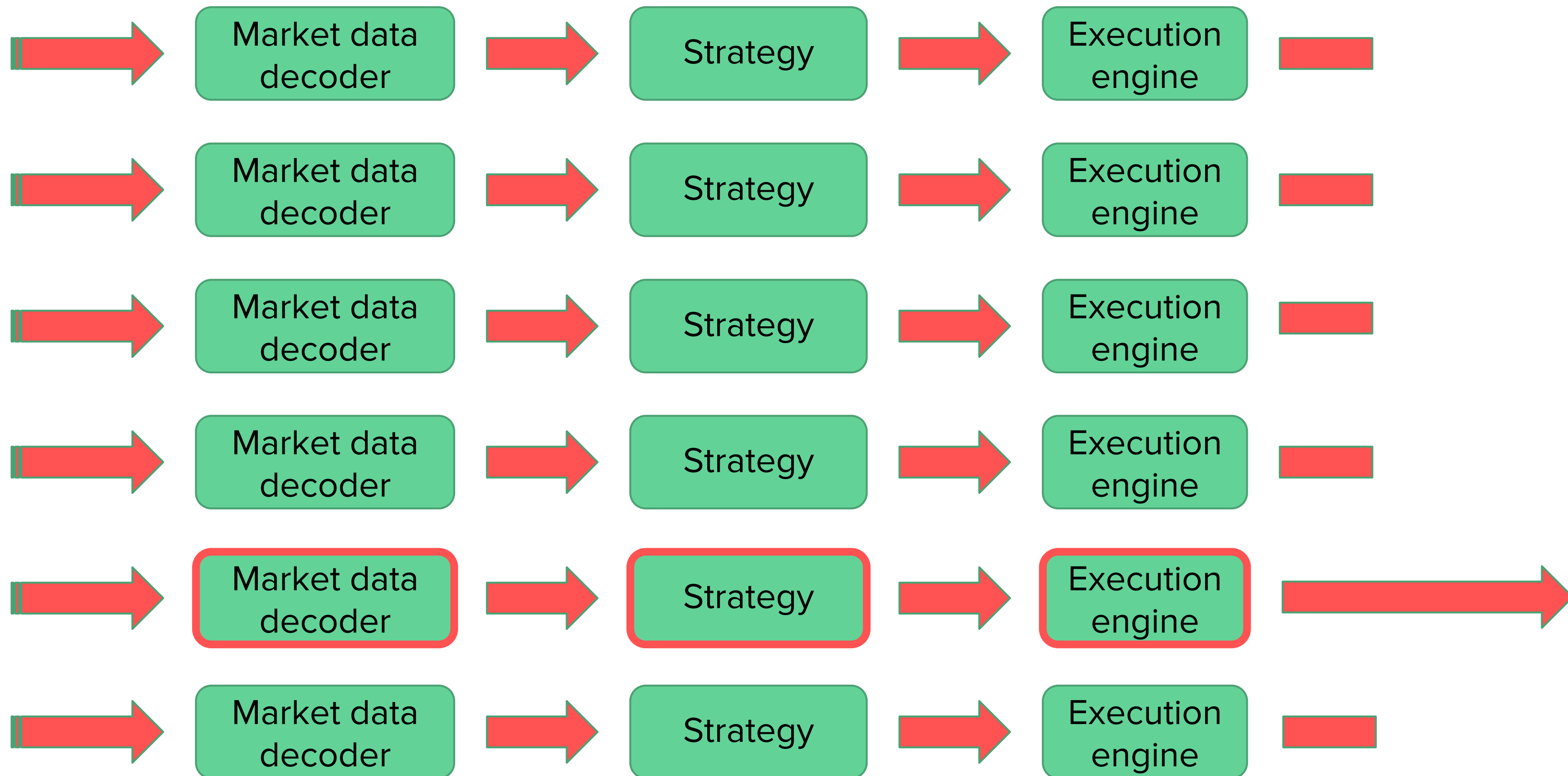
```
__attribute__((noinline))
void ComplexLoggingFunction()
{
  ...
}
```

Keeping the cache hot

Remember, the full hotpath is only exercised very infrequently - your cache has most likely been trampled by non-hotpath data and instructions

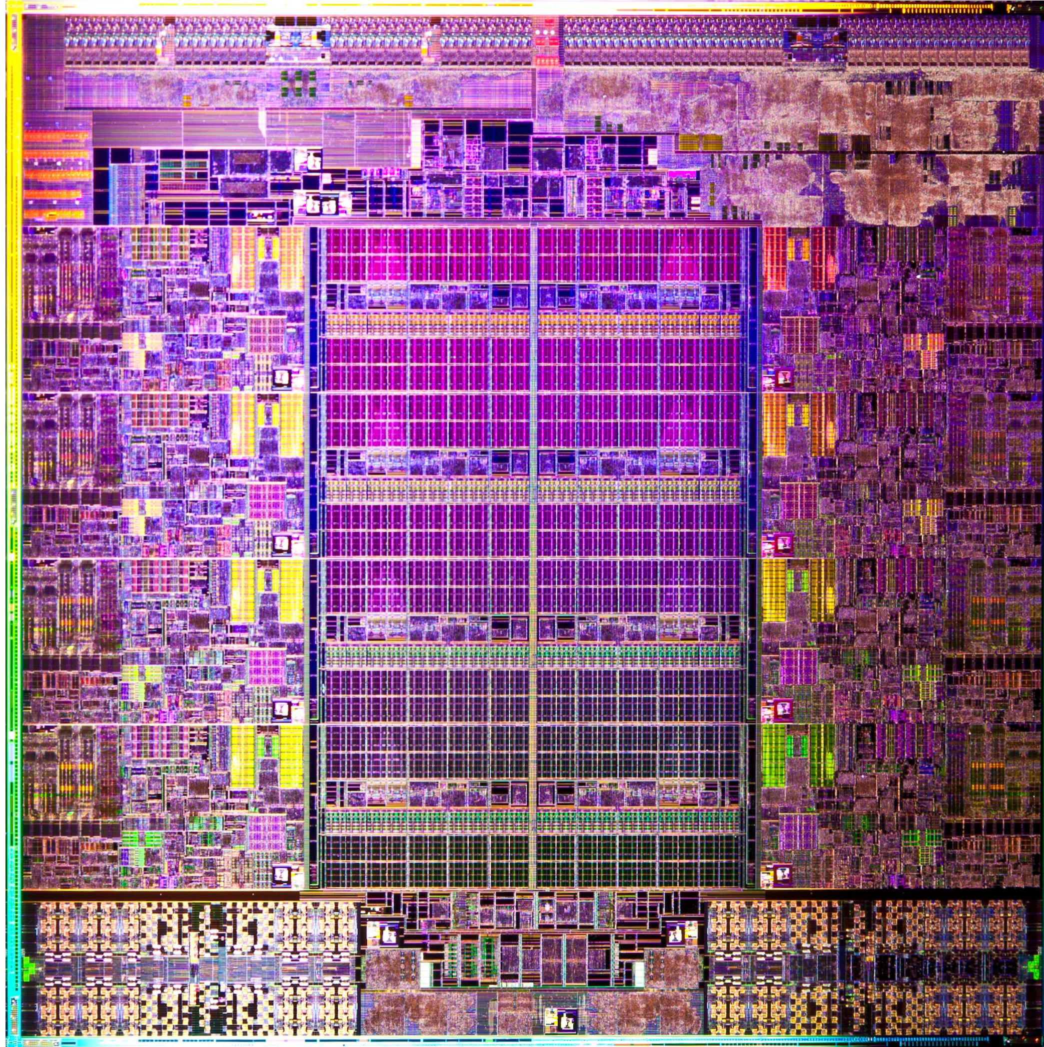


A simple solution: run a very frequent dummy path through your entire system, keeping both your data cache and instruction cache primed



Bonus: this also trains the hardware branch predictor correctly

Intel Xeon E5 processor



Source:
Intel Corporation

- Don’t share L3 - disable all but 1 core (or lock the cache)
- If you do have multiple cores enabled, choose your neighbours carefully:
    - Noisy neighbours should probably be moved to a different physical CPU

Surprises and war stories

"I have always wished for my computer to be as easy to use as my telephone; my wish has come true because I can no longer figure out how to use my telephone."

– Bjarne Stroustrup

Placement new can be slightly inefficient

Quick refresher:

```
#include <new>
Object* object = new(buffer)Object;
```

- However, if you use:
  - Any version of `gcc` without `-std=c++17` or `-std=c++1z`
  - Any version of `gcc` below 7.1 (May 2017)
  - Any version of `clang` below 3.4 (January 2014)
- Placement new will perform a null pointer check on the memory passed in
  - And if null is passed in:
    - The returned object is also null
    - No calls to the constructor or destructor will take place

- Why do compilers check whether memory passed in might be null?
    - The C++ spec was ambiguous about what placement new must do
- Marc Glisse/Jonathan Wakely (wg21.link/cwg1748) clarified this in 2013:
    - Passing null to placement new is now Undefined Behaviour [5.3.4.15]
- For several trading systems written in `gcc`, this inefficiency had a considerable negative performance effect:
    - More instructions mean fewer opportunities to inline and optimize
- There is a workaround: declare a throwing type-specific placement new

```
void* Object::operator new(size_t, void* mem) /* can throw */ {
   return mem;
}
```

Small string optimization support

```
std::unordered_map<std::string, Instrument> instruments;
return instruments.find({"IBM"}) != instruments.end();
```

- This will avoid an allocation with:
  - `gcc` 5.1 or greater, and if the string is 15 characters or less
  - `clang` if the string is 22 characters or less
- However, if you are using `gcc` >= 5.1 and an ABI compatible linux distribution such as Redhat/Centos/Ubuntu/Fedora, then you are probably still using the old `std::string` implementation
  - Including C.O.W. semantics
  - First mentioned (as slow) by Herb Sutter in 1999!

Overhead of C++11 static local variable initialization

```
struct Random {
  int get() {
    // threadsafe!
    static int i = rand();
    return i;
  }
};

int main() {
  Random r;
  return r.get();
}
```

```
Random::get():
  movzx eax, BYTE PTR guard var
  test al, al
  je .L13 // not yet initialized
  mov eax, DWORD PTR get()::i
  ret
.L13
   // acquire and set the guard var
```

5-10% overhead compared to non-static access, even if binary is single threaded

std::function may allocate

```
struct Point {
  double dimensions[3];
};

int main() {
  std::function<void()> no_op { [point = Point{}] {} };
}
```

```
main:
 mov edi, 24
 call operator new(unsigned long)
```

Consider `inplace_function` (D0419R0):

- http://github.com/WG21-SG14/SG14/blob/master/SG14/inplace_function.h
- Drop-in replacement for `std::function`
- Defaults to a 32 byte internal buffer

```
int main() {
  inplace_function<void()> no_op { [point = Point{}]  {} };
}
```

```
main:
 xor eax, eax
 ret
```

```
inplace_function<void(), 16> no_op { [point = Point{}]  {} };
// error: static assertion failed: Closure larger than buffer
```

std::pow can be slow

std::pow is a transcendental function, meaning it goes into a second, slower phase if the accuracy of the result isn't acceptable after the first phase

```
auto base = 1.00000000000001, exp1 = 1.4, exp2 = 1.5;
std::pow(base, exp1) = 1.0000000000000140
std::pow(base, exp2) = 1.0000000000000151
```

```
Benchmark                                Time     Iterations
------------------------------------------------------------
pow(base, 1.4) [glibc 2.17]             53 ns       13142054
pow(base, 1.4) [glibc 2.21]             53 ns       13142821
pow(base, 1.5) [glibc 2.17]         478195 ns           1457
pow(base, 1.5) [glibc 2.21]          63348 ns          11113
```

See http://entropymine.com/imageworsener/slowpow for a nice discussion

Measurement of low latency systems

"Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is."

– Rob Pike

Measurement of low latency systems

- Two common approaches:
  - Profiling: examining what your code is doing (bottlenecks in particular)
  - Benchmarking: timing the speed of your system
- Caution: profiling is not necessarily benchmarking
  - Profiling is useful for catching unexpected things
  - Improvements in profiling results are not a 100% guarantee that your system is now faster

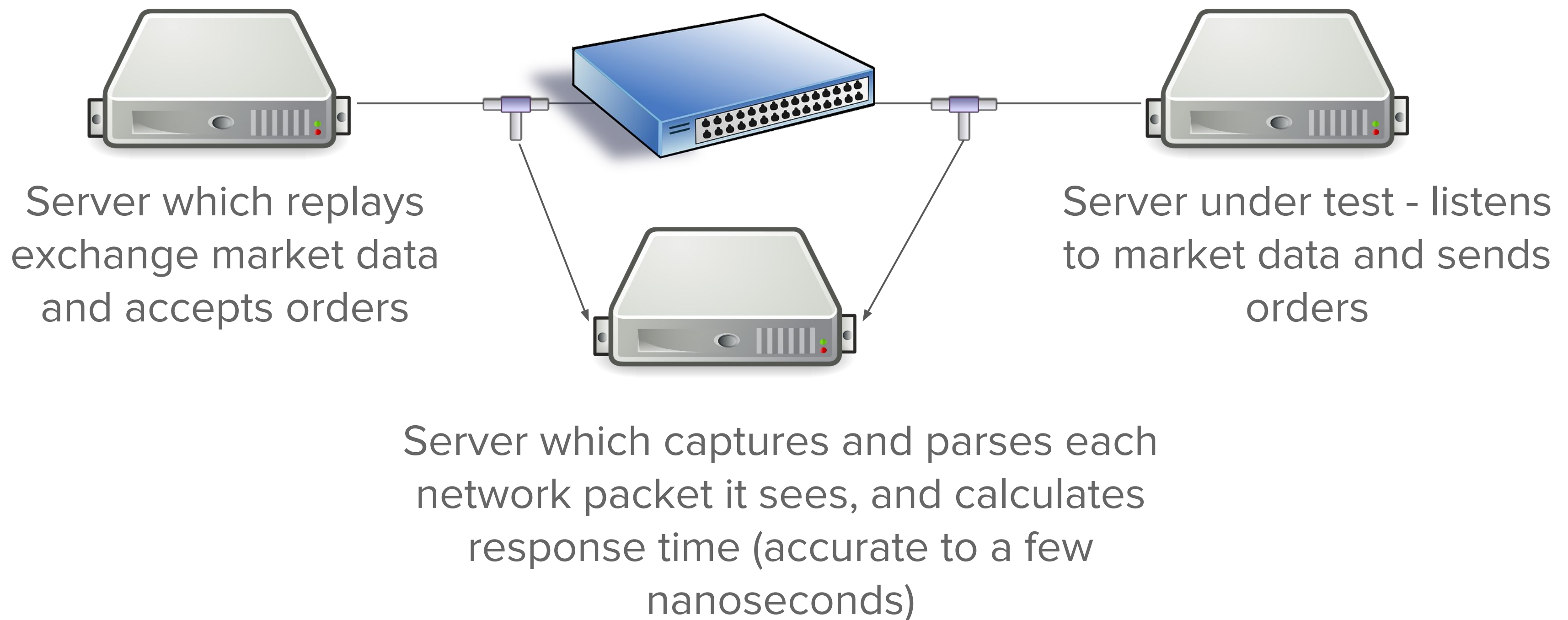
Measurement of low latency systems

- ✘ Sampling profilers (e.g. `gprof`)
  - They miss the key events
- ✘ Instrumentation profilers (e.g. `callgrind`)
  - They are too intrusive
  - They don't catch I/O slowness/jitter
- ✘ Microbenchmarks (e.g. Google benchmark)
  - They are not representative of a realistic environment
  - Takes some effort to force the compiler to not optimize out the test
  - Heap fragmentation can have an impact on subsequent tests

They are all useful in some ways, but not for micro-optimization of code

Measurement of low latency systems

✓ Most useful: measure end-to-end time in a production-like setup



Summary

“A language that doesn't affect the way you think about programming is not worth knowing.”

– Alan Perlis

- Have a good knowledge of C++ well, including your compiler
- Understand the basics of machine architecture, and how it will impact your code
- Aim for very simple runtime logic:
    - Compilers optimize simple code the best
    - Prefer approximations instead of perfect precision where appropriate
    - Do expensive work only when you have spare time
- Conduct accurate measurement - this is essential

Thanks for listening!

carl.cook@gmail.com