Low Latency C++ for Fun and Profit

Carl Cook, Ph.D.

@ProgrammerCarl
info@perfict.io

Introduction

About me:

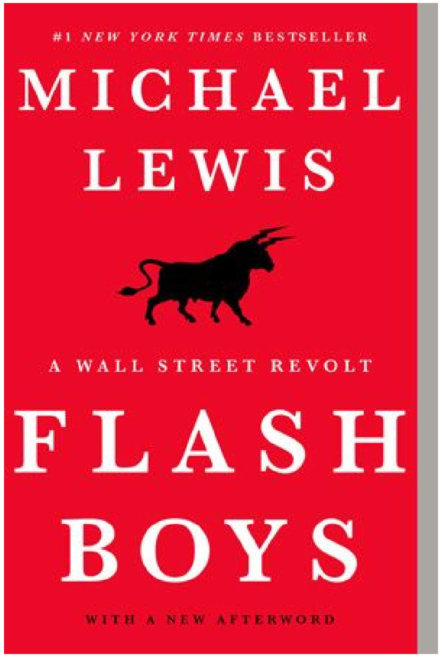
- Freelance software developer
- Experience is with trading companies (mainly)
- A member of ISO SG14 (gaming, low latency, trading)

Contents:

- A 30 second introduction to trading
- Performance techniques for low latency, and then some surprises
- Measurement of performance

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

What is electronic trading/HFT/market making/algo trading?



```
while (true) {
  try_buy_low();
  try_sell_high();
}
```

Why the need for speed?

Electronic market makers aim for the lowest latency possible:

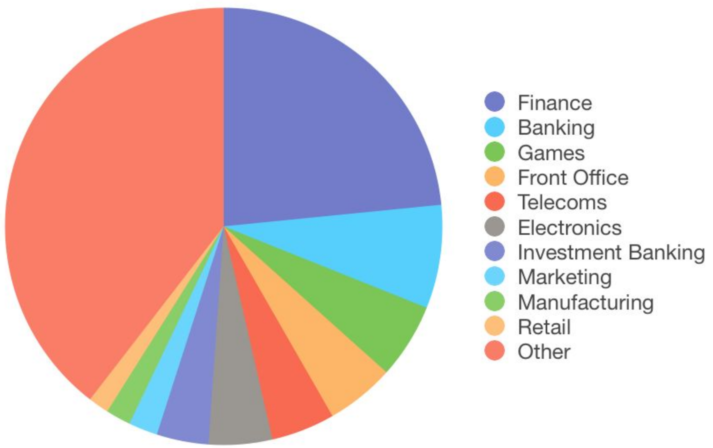
- Fast reaction to market events
  - Allowing now out-of-date orders to be adjusted (before losing money)
  - To be the first to spot a favorable order and try to trade with this

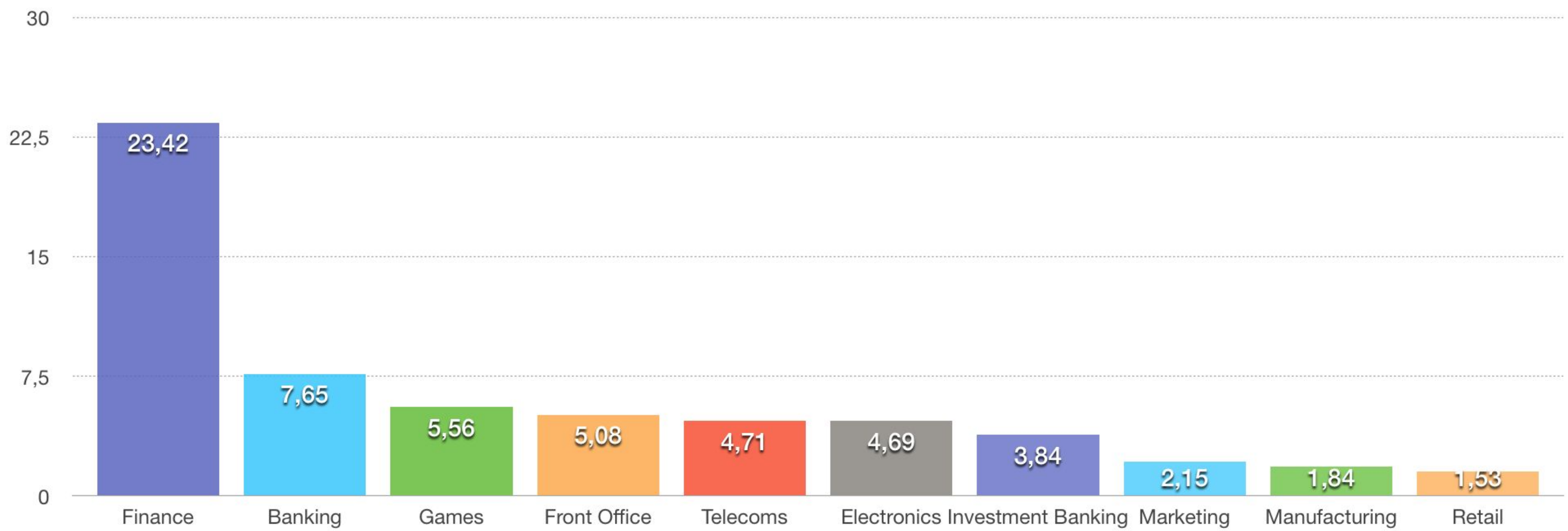
Solving this challenge has some nice spin-offs to other industries:

- More efficient code: longer battery life/drone flight time/power savings
- Faster/more responsive autonomous vehicles
- Better general application performance
- Continually improving hardware
- ...

C++ in finance

Source: Jetbrains





Technical challenges of low latency trading

“If you’re not at all interested in performance, shouldn’t you be in the Python room down the hall?”

– Scott Meyers

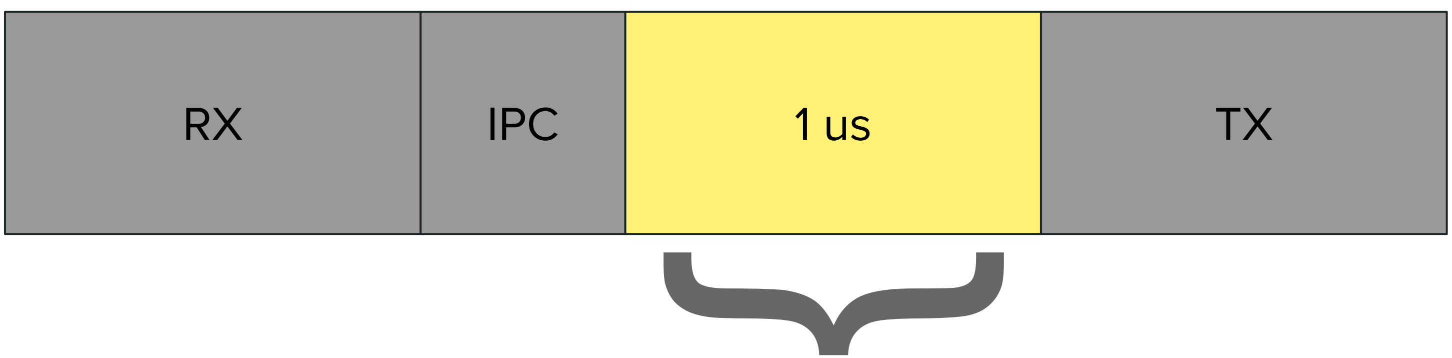
The ‘Hotpath’

- 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
- A lot can go wrong in a few microseconds

Execution time is a limited resource

If the target is 3.5us wire to wire (for example), then:

- 1us for RX of market data message from exchange
- 1us for TX of order message to exchange
- Maybe 0.5us of misc IPC, and jitter that’s hard to get rid of
- Leaves approximately 1us for the actual trading code
    - Arguably around 3K CPU cycles/12K instructions
    - But think about memory latency, pipeline stalls, cache misses, etc



{Time it takes for light to travel 300 metres}

The role of C++

From Bjarne Stroustrup:

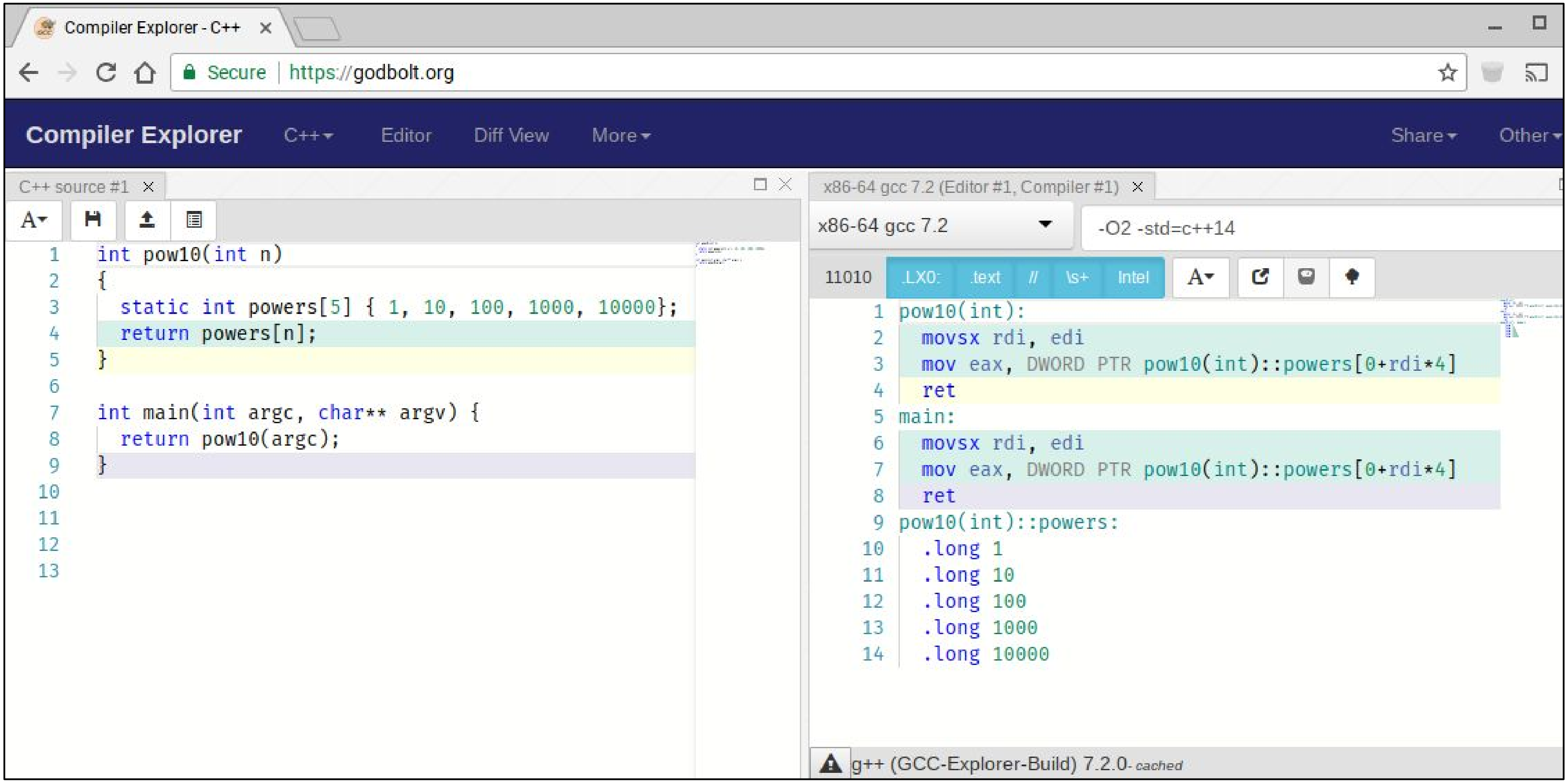
"C++ enables zero-overhead abstraction to get us away from the hardware without adding cost"

But: even though C++ is good at saying what will be done, there are 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:

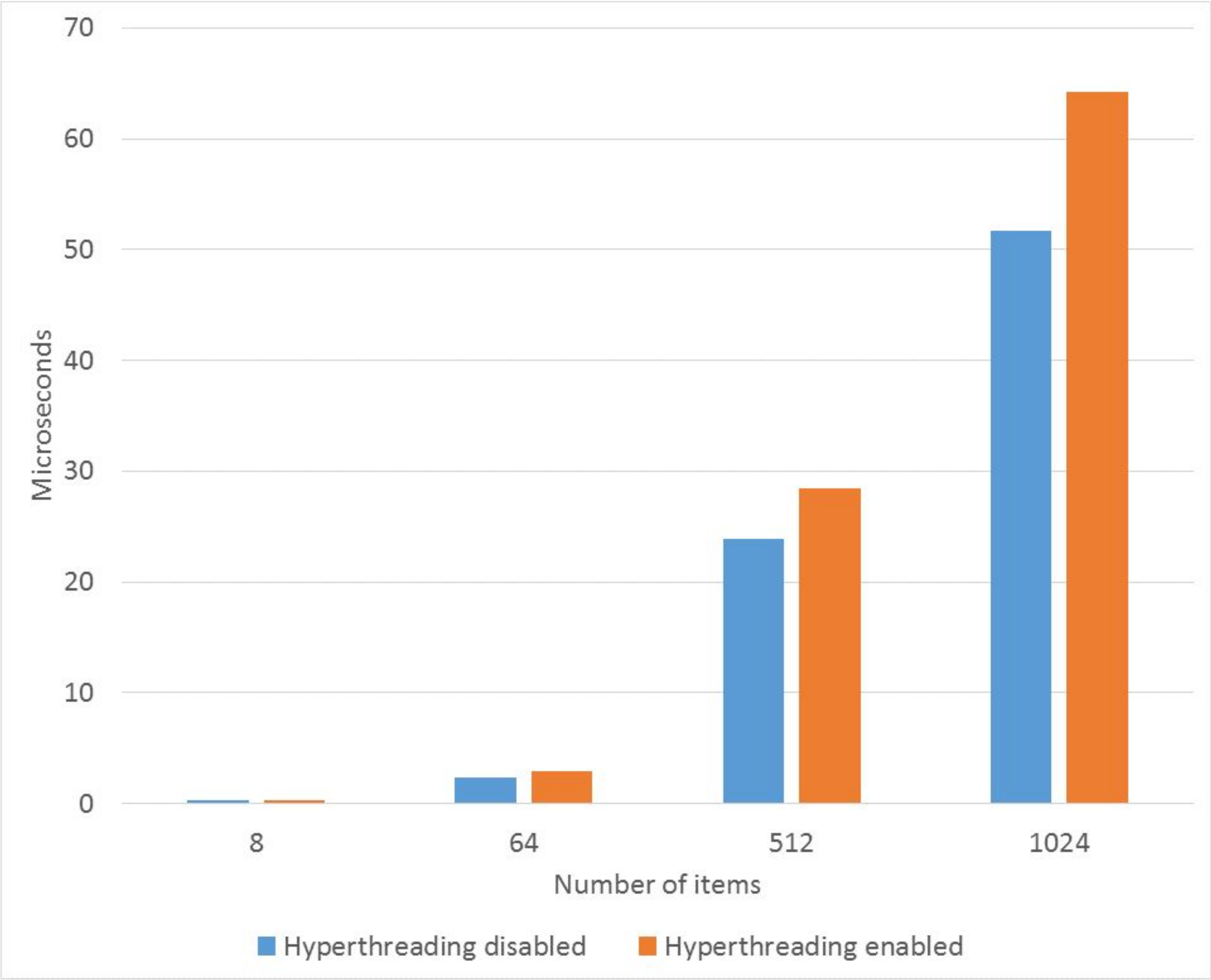


The importance of system tuning (results on the next page)

```
std::vector<int> items;
items.reserve(1024);

void SortVector(benchmark::State& state) {
  for (auto _ : state) {
    const auto N = state.range(0);
    items.resize(N);
    for (int i = 0; i < N; ++i)
      items[i] = rand() % N;
    std::sort(items.begin(), items.end());
  }
}

BENCHMARK(Sort)->Range(8, 1024);
```



Same:

- Hardware
- Operating system
- Binary
- Background load

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

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 if (config.UseOrderSenderB())
    return std::make_unique<OrderManager<OrderSenderB>>();
  else
    throw;
}

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

Memory allocation

- Allocations are of course 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
- Be aware that destructors may be inlined
  - This can start trampling your instruction cache

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 basically zero cost if they don't throw
    - Maybe some slight code reordering, but the cost is negligible
- 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

Branch reduction

Branching approach:

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

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

Multi-threading

Multithreading is best avoided for latency-sensitive code:

- Synchronization of data via locking is going to be 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, i.e.:
  - Maybe you can live with out-of-sequence updates
  - Maybe the machine architecture prevents torn reads/writes, preserves ordering of stores and loads (etc)

Data lookups

The software engineering textbooks would 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;
...
```

Actually, denormalized data is not a sin:

- Chances are there is space in the cacheline that you read to have pulled in the extra field, avoiding an additional lookup

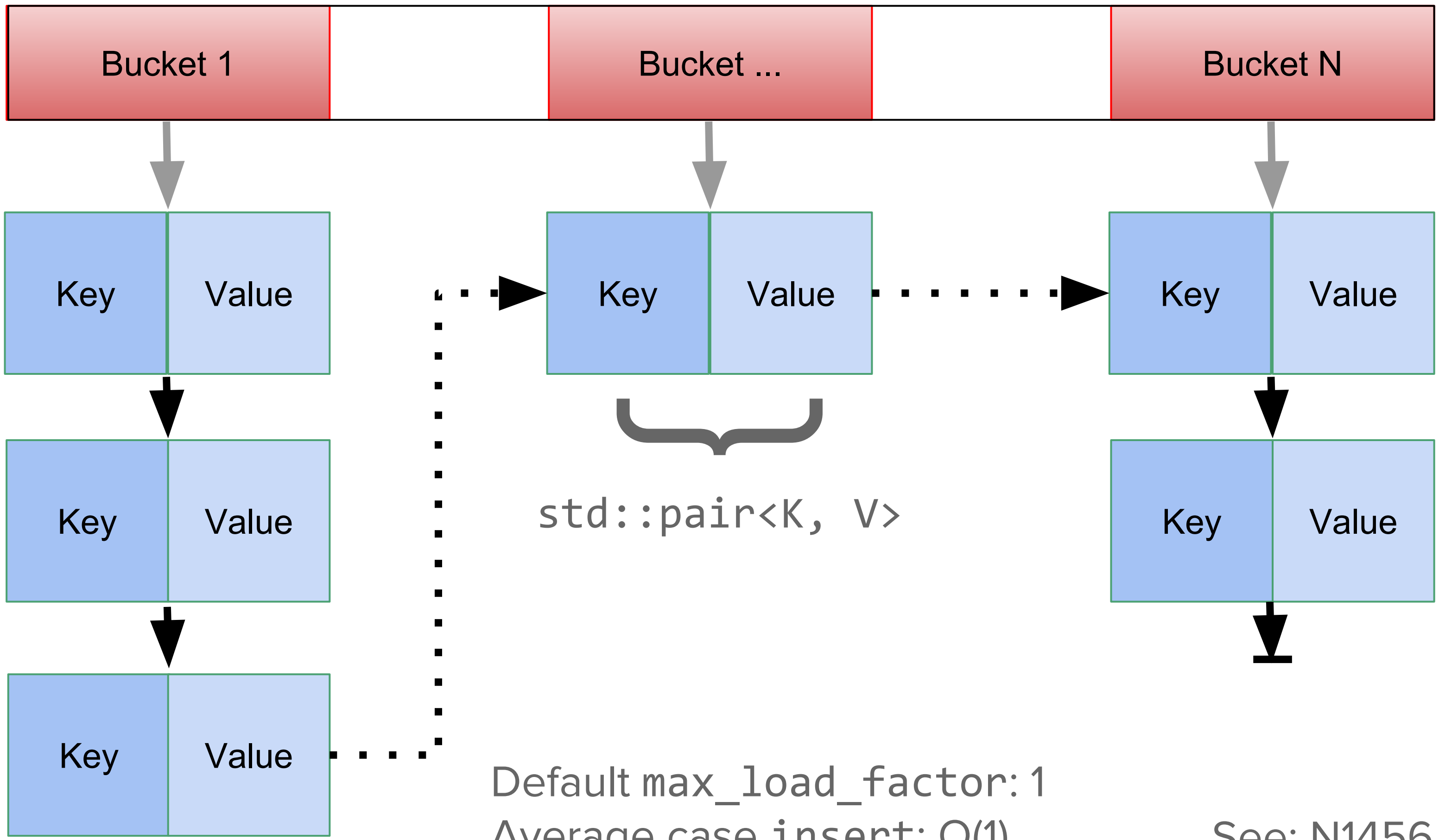
```
struct Market {
  int32_t id;
  char shortName[4];
  int16_t quantityMultiplier;
    ...
}
```

```
struct Instrument {
  float price;
  int16_t quantityMultiplier;
    ...

}
```

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

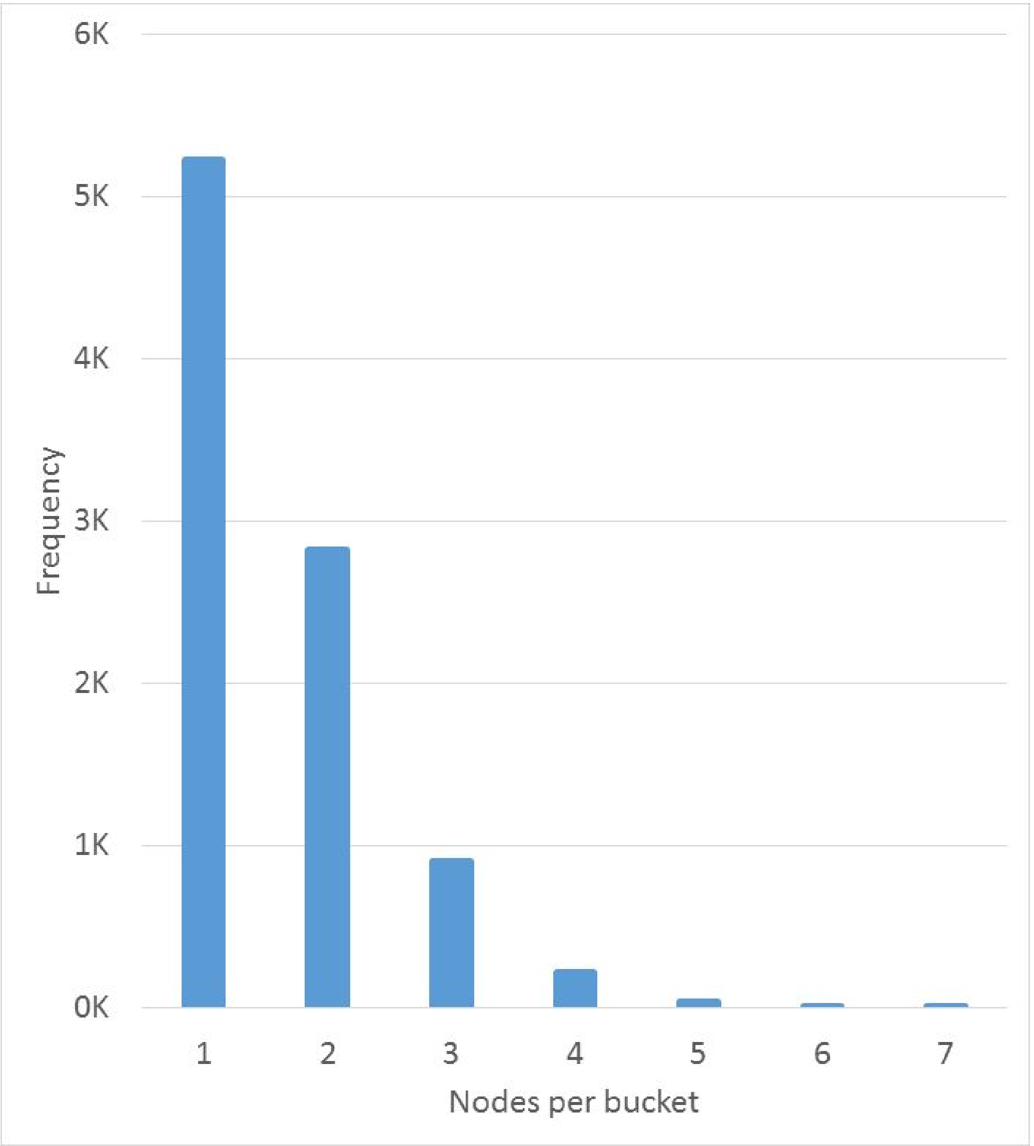
Fast associative containers (`std::unordered_map`)



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

See: N1456

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



Complexity of `find`:

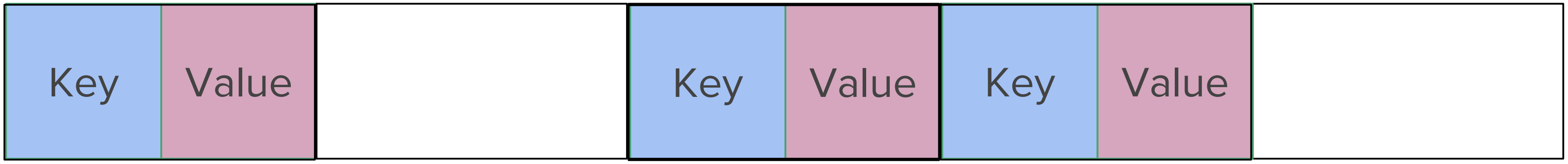
Average case: O(1)

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`



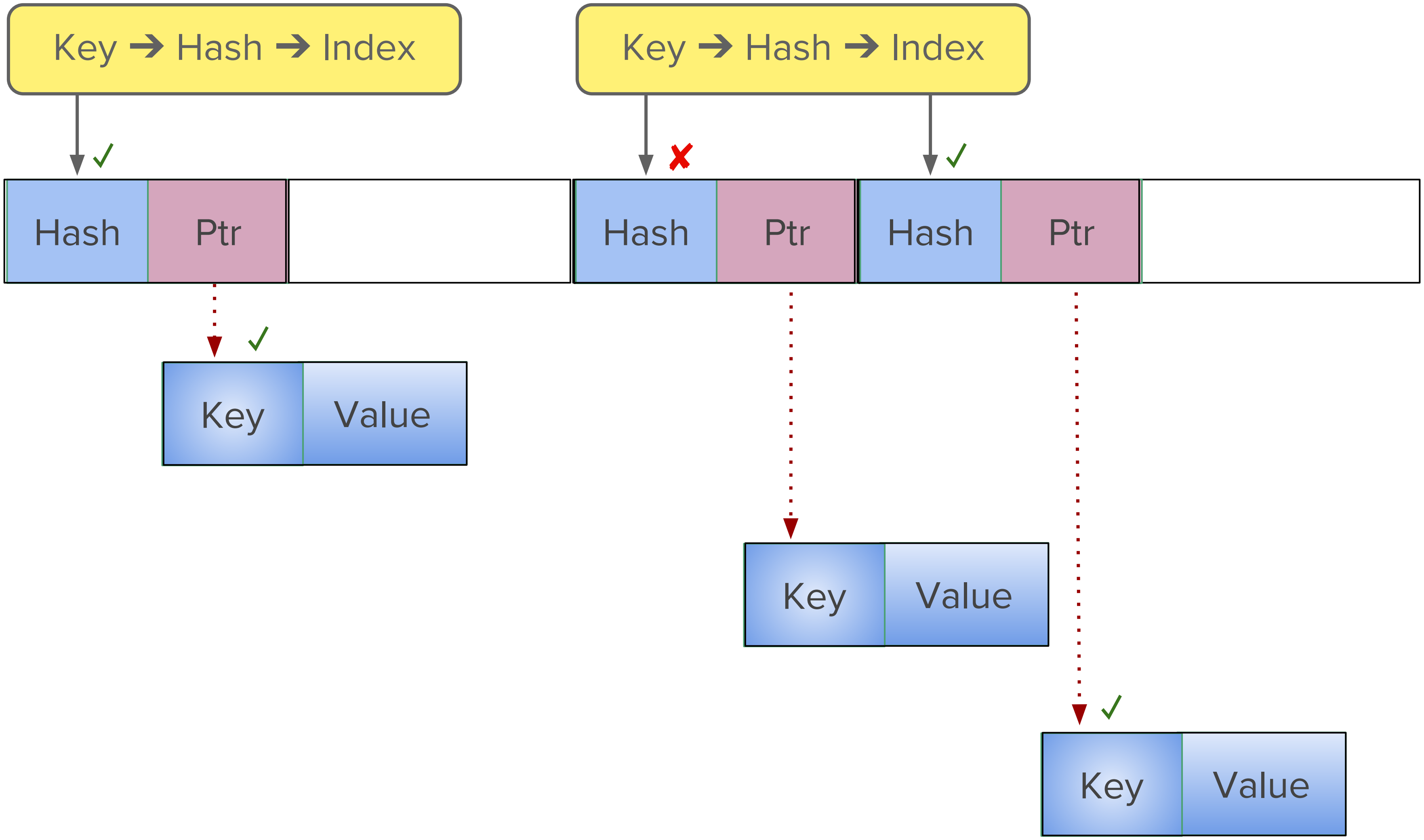
✓ Key/Value pairs are in contiguous memory - no pointer following between nodes

✘ Complexity around collision management

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

Goals:

- Minimal memory footprint
- Predictable cache access patterns (no jumping all over the place)



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
```

Branch prediction hints

```
#define likely(x)     __builtin_expect((x),1)
#define unlikely(x)   __builtin_expect((x),0)
```

- You may recognise these from the linux kernel source
- The compiler often picks the right case in the first place, but there’s no guarantee

gcc with no hints

```
int GetErrorCode() {
  return rand() % 255 + 1;
}

int main(int argc, char**) {
  if (argc > 1)
    return GetErrorCode();
  else
    return 0;
}
```

```
main:
  cmp edi, 1 // argc
  jle .L7
  sub rsp, 8
  call rand
  mov ecx, 255
  cdq
  idiv ecx
  lea eax, [rdx+1]
  pop rdx
  ret
.L7:
  xor eax, eax // zeros ebx
  ret
```

Now with branch prediction hints

```
int GetErrorCode() {
  return rand() % 255 + 1;
}

int main(int argc, char**) {
  if (unlikely(argc > 1))
    return GetErrorCode();
  else
    return 0;
}
```

```
main:
  cmp edi, 1
  jg .L12
  xor eax, eax
  ret
.L12:
  sub rsp, 8
  call rand
  mov ecx, 255
  cdq
  idiv ecx
  lea eax, [rdx+1]
  pop rdx
  ret
```

- These “likely” attributes are useful if something called very rarely needs to be fast when called (i.e. expect more efficient assembly code to be generated)
- In all other cases:
  - Write your code to avoid branches, and
  - Train the hardware branch predictor (more about this later)
    - This is the dominant factor

See https://wg21.link/P0479 for a proposal to standardize these attributes

See https://groups.google.com/a/isocpp.org/forum/#!forum/sg14 for a lively debate on this proposal

`((always_inline))` and `((noinline))`

- `((always_inline))` and `((noinline))` can be useful
  - Means: inlining is preferred/inlining should be avoided
  - But be careful: measure
- Please note that the `inline` keyword is not really what you are looking for
  - Mainly means: multiple definitions are permitted

A quick example: forcing a method to be not inlined (for good reason)

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

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

Default gcc generated code

```c
void get_error_code() { ... }

int main(int argc, char**) {
  if (argc > 1)
    return get_error_code();
  else
    return 0;
}
```

```
get_error_code:
  ...
  ret
main:
  cmp edi, 1 // argc register
  jle .L6
  jmp get_error_code
.L6:
  xor eax, eax // zeros eax
  ret // eax is the ret val
```

Forcing get_error_code to be inlined

```
__attribute__((always_inline))
void get_error_code() { ... }

int main(int argc, char**) {
  if (argc > 1)
    return get_error_code();
  else
    return 0;
}
```

```
main:
  cmp edi, 1
  jle .L6
  get_error_code instruction 1
  get_error_code instruction ..
  get_error_code instruction N
  mov eax, [error code]
  ret
.L6:
  xor ebx, ebx // zeros ebx
  ret
```

Combining inlining hints and branch prediction hints

Combining `noinline` with “unlikely” branch prediction

```c
__attribute__((noinline))
void get_error_code() { ... }

int main(int argc, char**) {
  if (unlikely(argc > 1))
    return get_error_code();
  else
    return 0;
}
```

```asm
get_error_code:
  ...
  ret
main:
  cmp edi, 1
  jg .L7
  xor eax, eax
  ret
.L7:
  jmp get_error_code
```

Other gcc compiler hints for cache locality

`__attribute__((hot))`:

Puts all functions into a single section in the binary, including ancestor functions

`__attribute__((cold))`:

Puts functions into a different section (and will avoid inlining)

This is somewhat useful - basically does the same as inlining of hot functions and no-inlining of cold functions

Prefetching

`__builtin_prefetch` can also be useful (if you know that the hardware branch predictor won't be able to work out the right pattern)

Example (of a binary search loop):

```
// next mid val after this iteration if we take the low path
 __builtin_prefetch(&array[(low + mid - 1)/2]);
// next mid val after this iteration if we take the high path
 __builtin_prefetch(&array[(mid + 1 + high)/2]);

 int mid = (low + high) / 2;
 if (array[mid] == key) return mid;
 if (array[mid] < key) low = mid + 1; // search high path
 else high = mid - 1; // search low path
```

Bonus: you can also prefetch the instruction cache

Compiler attributes <TL/DR>

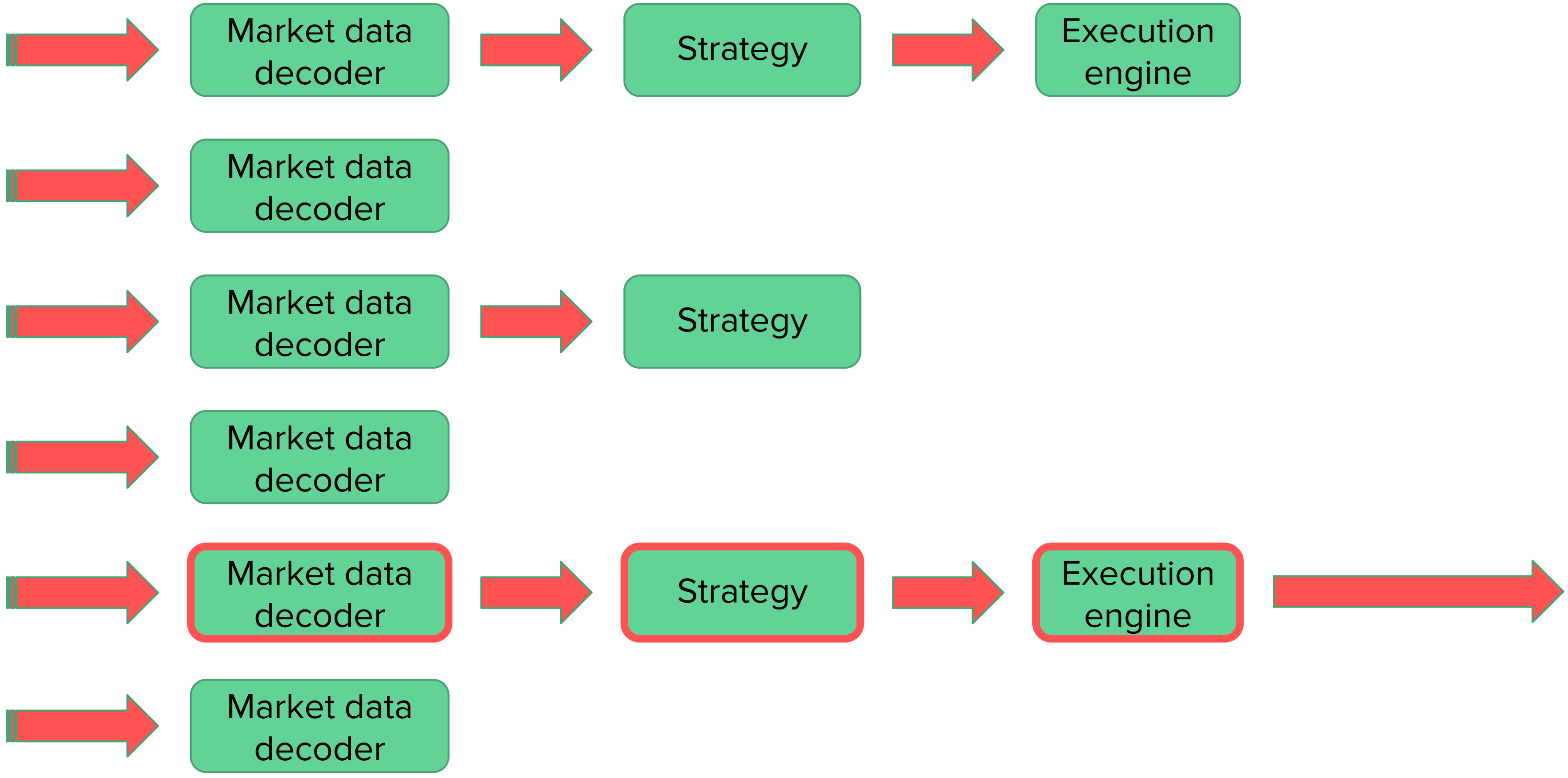
Pick one:

- Code with no (or minimal) branches
- `__attribute__((always_inline))` and `__attribute__((noinline))`
- `__builtin_expect()`
- `__attribute__((hot))` and `__attribute__((cold))`
- `__builtin_prefetch()`

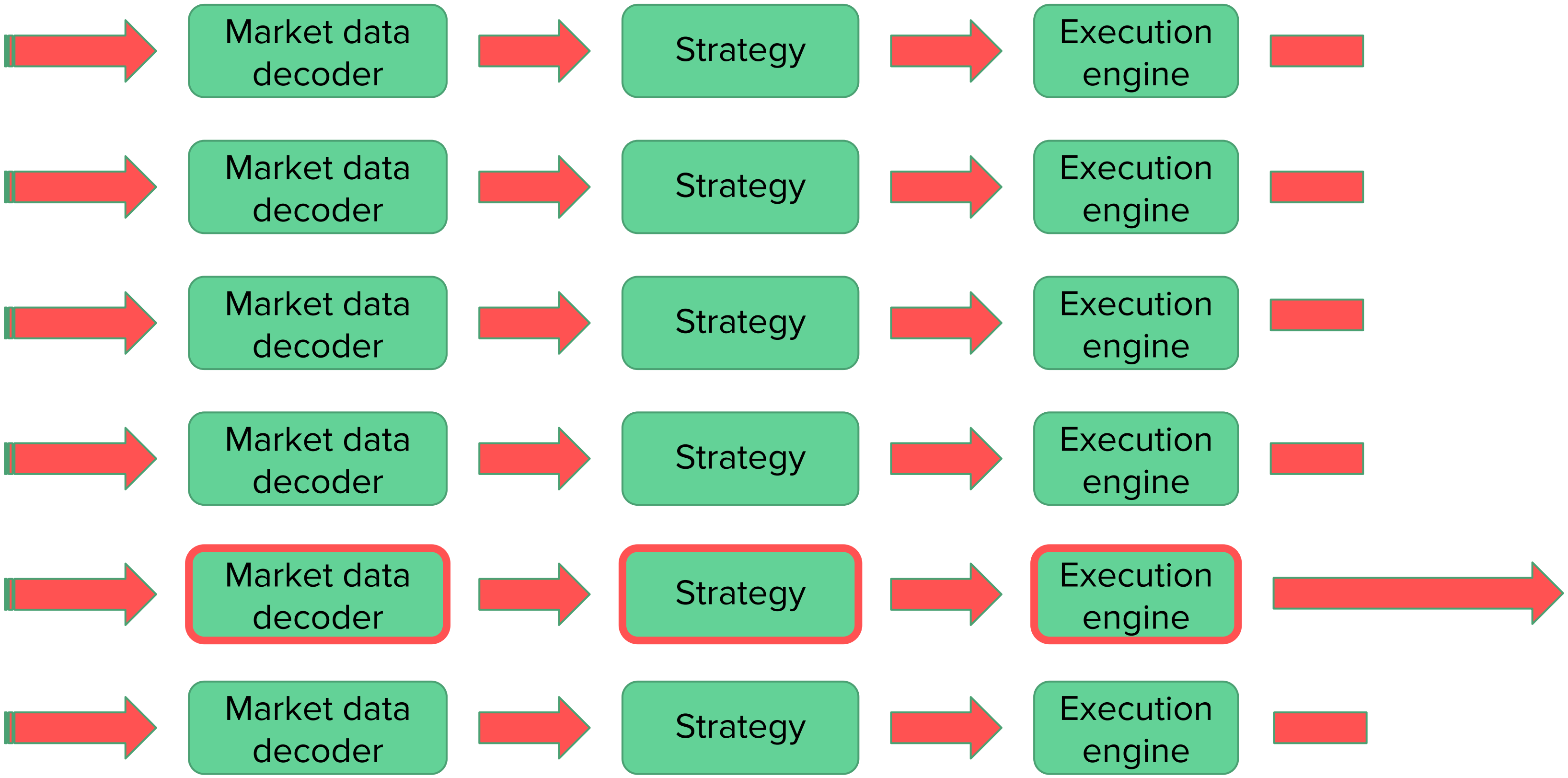
Usually you will see no further gain if you apply several of the above

Keeping the caches hot - a better way!

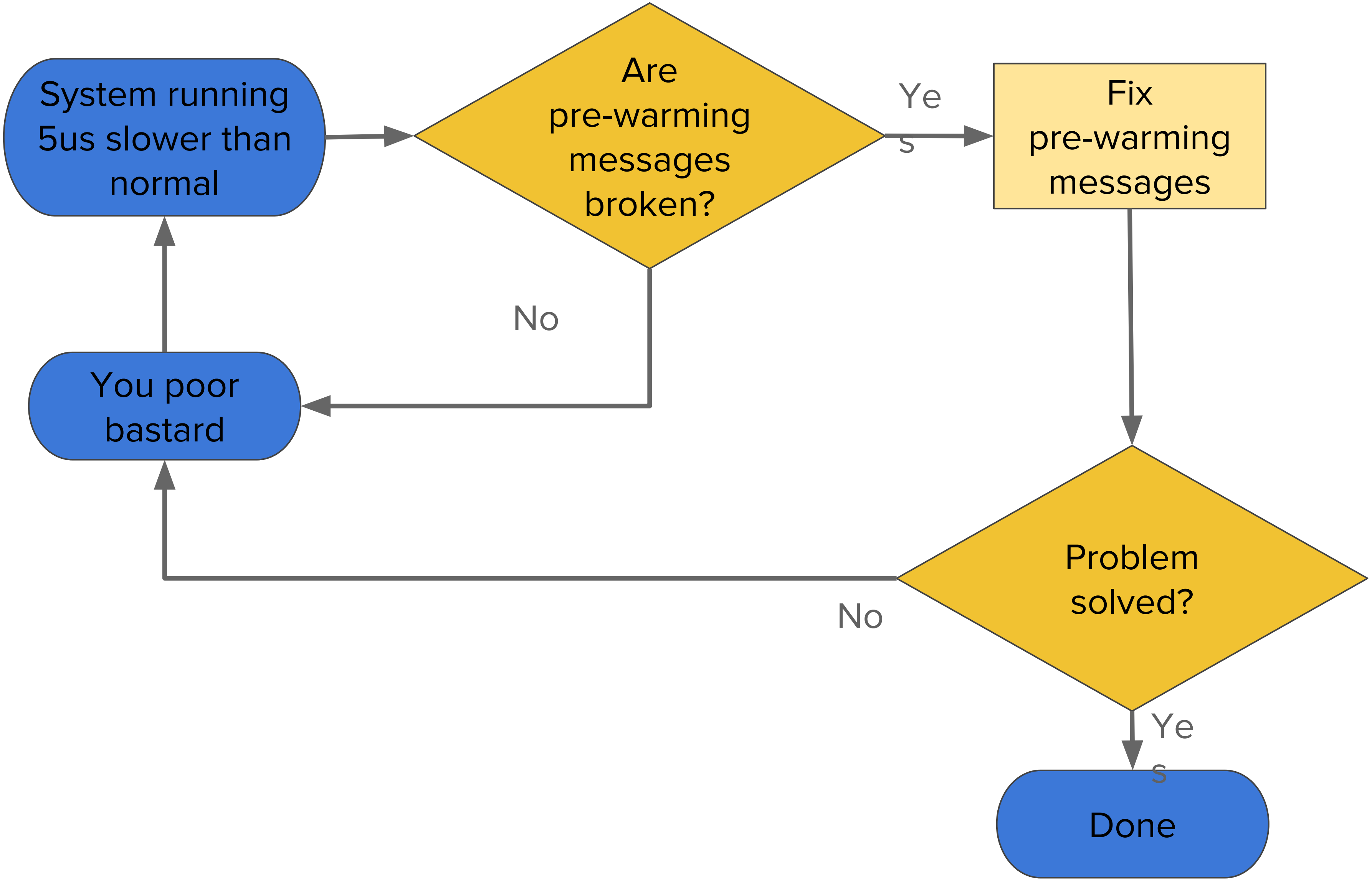
Remember, the full hotpath is only exercised very infrequently - your cache has most likely been trampled by non-hotpath data and instructions



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



Bonus: this also correctly trains the hardware branch predictor


System running 5us slower than normal
Are pre-warming messages broken?
Yes
Fix pre-warming messages
No
You poor bastard
Problem solved?
No
Yes
Done

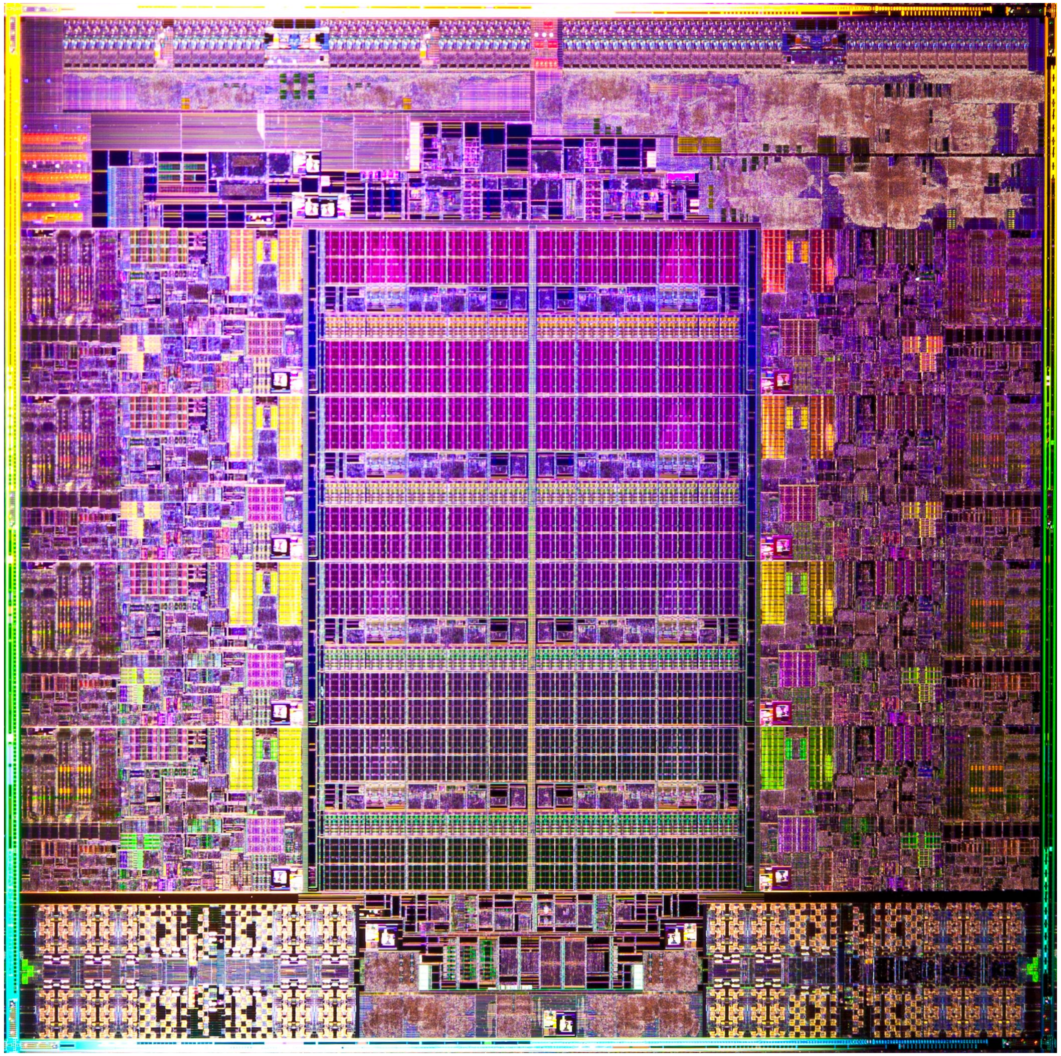
Hardware/architecture considerations

Quick recap:

- A server can have N physical CPUs (one CPU attaches to one socket)
  - Each CPU can have N cores (ignoring hyperthreading per core)
    - Each core has a:
      - L1 data cache (~32KB)
      - L1 instruction cache (~32KB)
      - Unified L2 cache (~512KB)
  - All cores share a unified L3 cache (~50Mb)

Source:
Intel Corporation

Intel Xeon E5 processor



Source:
Intel Corporation

- Don’t share L3 - disable all other cores (or lock the cache)
  - This might mean paying for 22 cores but only using 1
- 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

Small string optimization support

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

- This will only work:
  - With `gcc` 5.1 or greater, and if the string is 15 characters or less
  - In `clang` if the string is 22 characters or less
- In `gcc`, `std::string` has C.O.W. semantics (prior to `gcc` 5.1)
  - This gets expensive (during copying/destruction) due to atomics
  - First mentioned by Herb Sutter in *1999*
- If you use a ABI compatible linux distribution such as Redhat/Centos/Ubuntu/Fedora, then you are probably still using the old `std::string` implementation (even with the latest versions of `gcc`):
  - C.O.W and no SSO support

`std::string_view` (to the rescue)

Provides allocation-free substrings and string literals

```
std::map<std::string, Instrument, std::less<>> instruments;
instruments.find(std::string_view{"FACEBOOK"})->second;

std::string name{"FACEBOOK"};
instruments.find(name.substr(1,3)); // "ACE"
```

Available in most C++17 compilers, and in C++14 as `std::experimental::string_view`

Avoiding `std::string` (and allocations)

- Consider something like `inplace_string`:
  - No allocation, compile time bounds checking, and full std::string interface
  - https://github.com/david-grs/inplace_string

```
using InstrumentName = inplace_string<16>;
InstrumentName instrumentName {"IBM"};
assert(InstrumentName::npos == instrumentName.find("GOOGLE"));
```

- Implicitly convertible to `std::string` if required

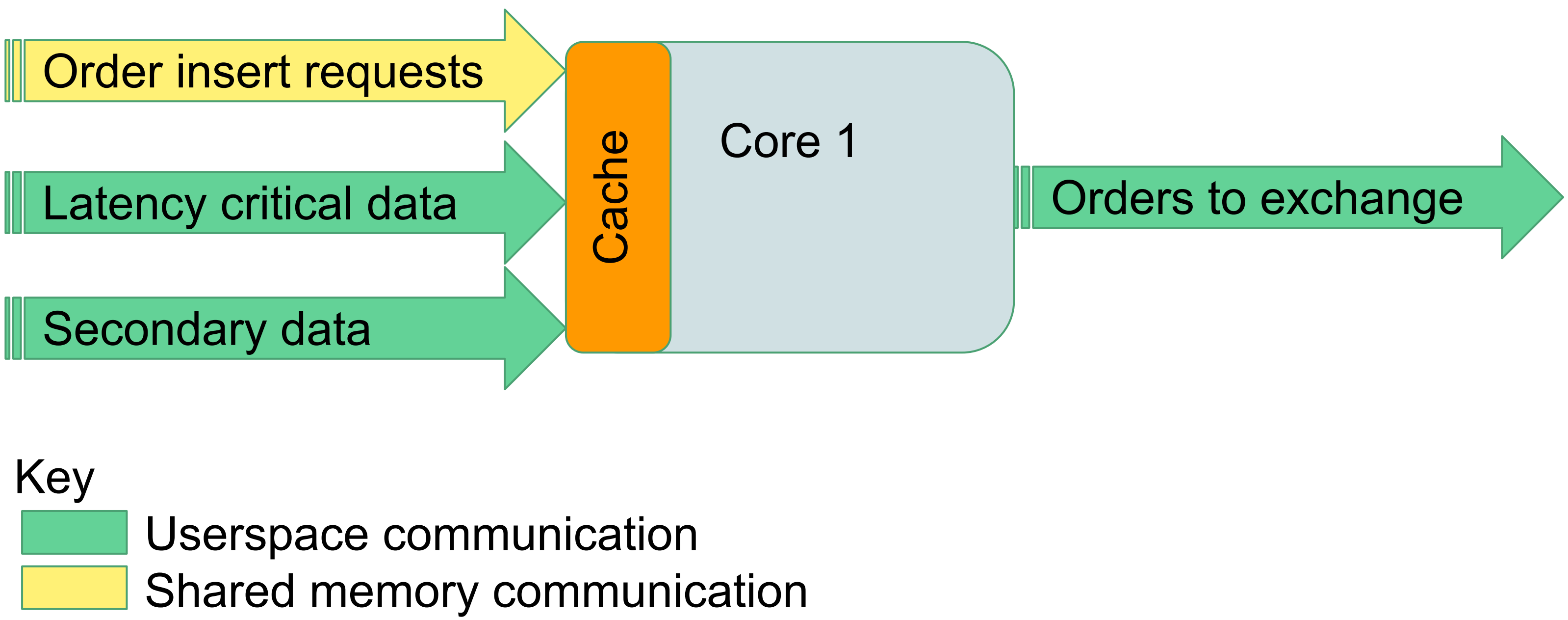
```
std::string str{instrumentName};
```

- In production, with a sample size of 1024, inserting 6 elements into a vector

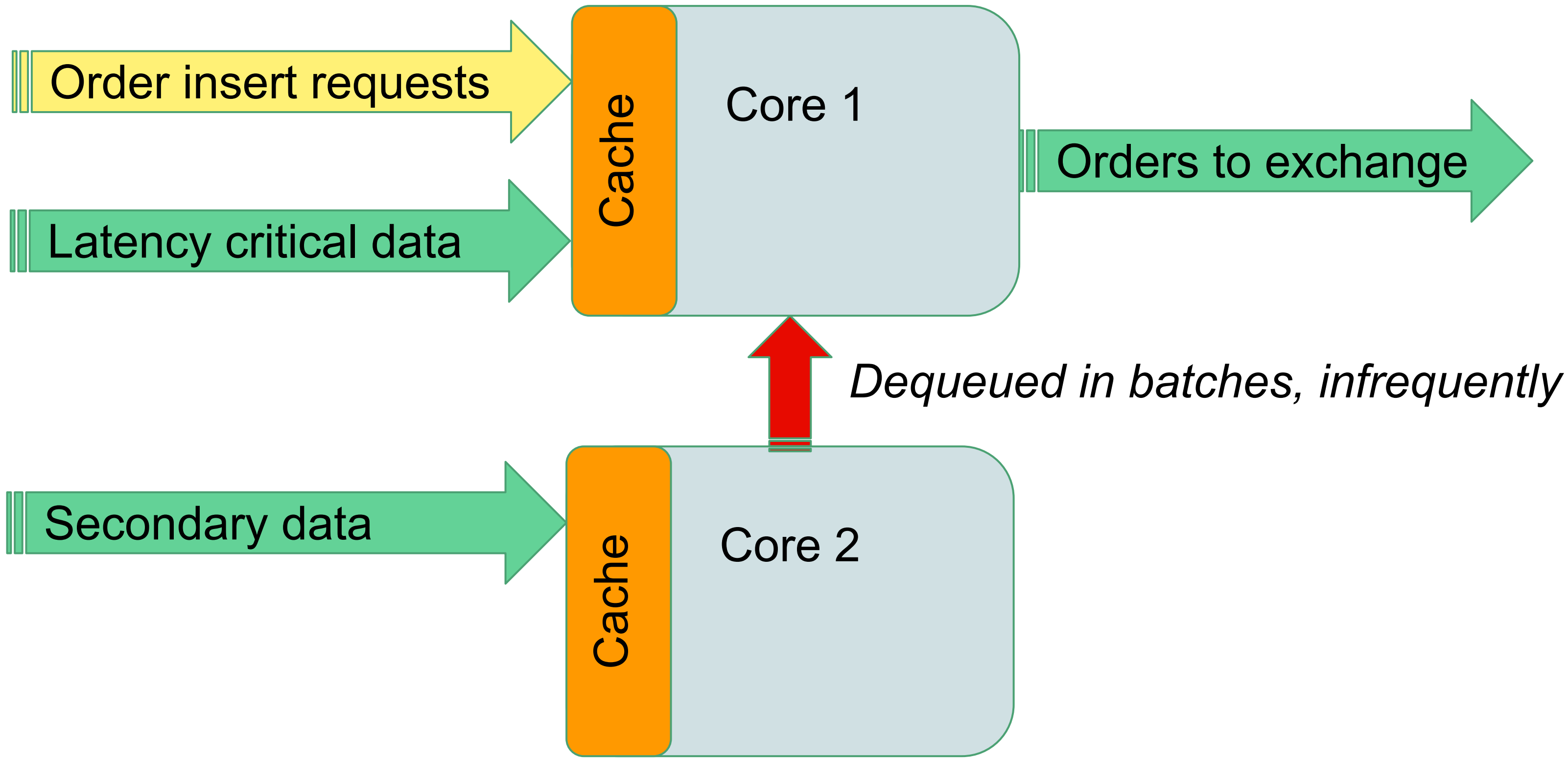
`std::string`	min=918ns	mean=3,003ns	max=29,518ns
`inplace_string<16>`	min= 28ns	mean= 61ns	max= 1,829ns

Userspace networking vs cache

- Userspace means we can receive data (prices, etc) without any system calls
- But there can be too much of a good thing:
  - All secondary data goes through the cache, even if we don’t use the data
  - When items go into the cache, other items are evicted



Alternative setup:



Key

- Userspace communication
- Shared memory communication
- Single writer/single reader lock free queue

Watch your enums and switches

```
enum Enum { Good, Bad, Ugly };

int main(int argc, char**) {
   switch ((Enum)argc) {
       case Good: Handle("GOOD");
       break;
       case Bad:  Handle("BAD");
       break;
       case Ugly: Handle("UGLY");
       break;
   }
}
```

```
main:
  sub rsp, 8
  test edi, edi
  je .L8
  cmp edi, 1
  je .L3
  cmp edi, 2
  je .L4
```

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
  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::pow` can be slow, really 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) = 1.000000000000140
std::pow(base, exp2) = 1.000000000000151
```

```
Benchmark                               Time      Iterations
--------------------------------------------------------------
pow(base, exp1) [glibc 2.17]        53 ns        13142054
pow(base, exp1) [glibc 2.21]        53 ns        13142821
pow(base, exp2) [glibc 2.17]    478195 ns            1457
pow(base, exp2) [glibc 2.21]     63348 ns           11113
```

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: seeing 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 isn’t a 100% guarantee that your system is now faster

✘ Sampling profilers (e.g. `gprof`) are not what you are looking for

- They miss the key events

✘ Instrumentation profilers (e.g. `valgrind`) are not what you are looking for

- They are too intrusive
- They don’t catch I/O slowness/jitter (they don’t even model I/O)

✘ Microbenchmarks (e.g. google benchmark) are not what you are looking for

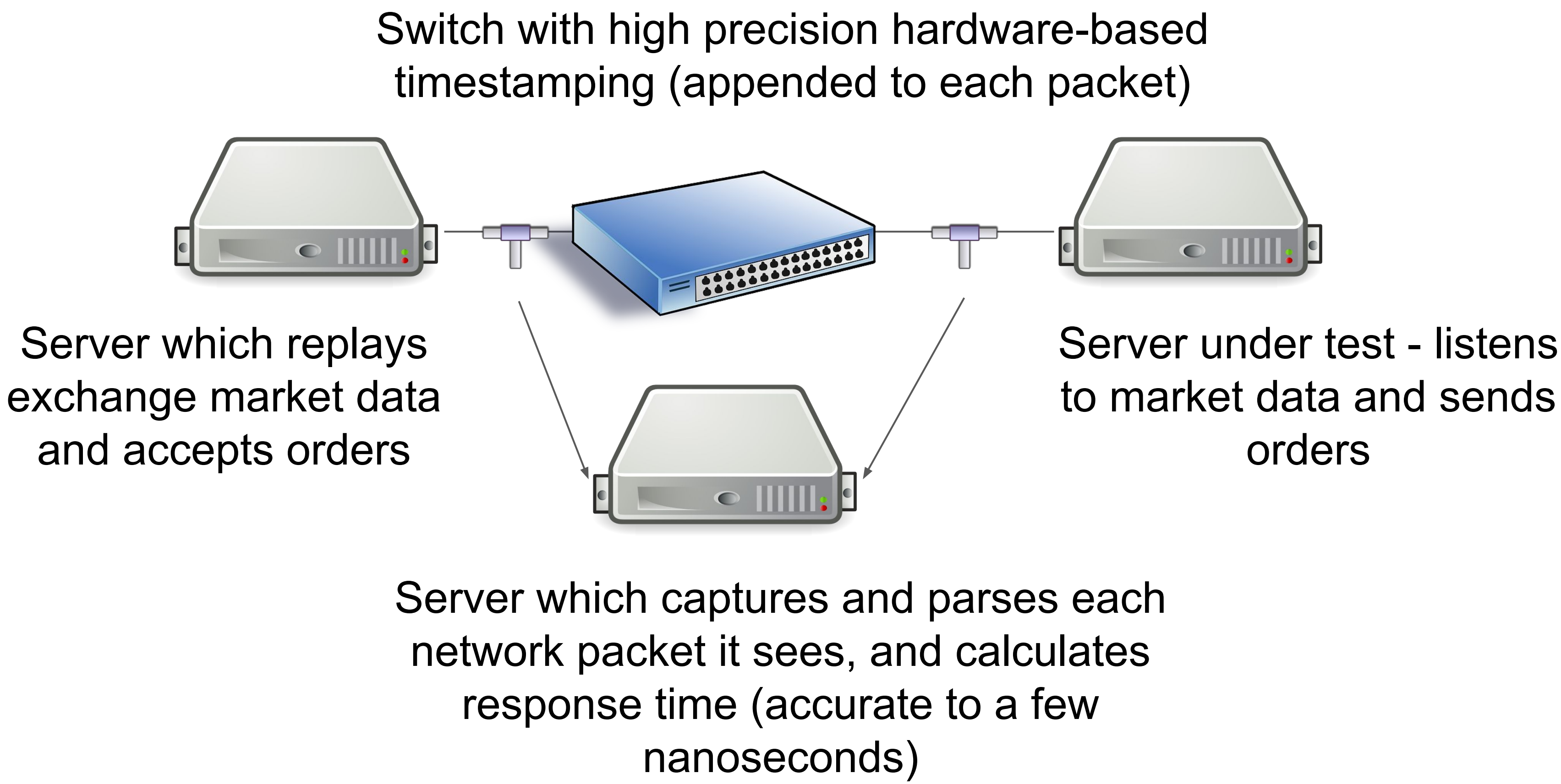
- They are not representative of a realistic environment
- Takes some effort to force the compiler to not optimize out the test
- Heap fragmentation can have an impact on subsequent tests

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

- Performance counters can be useful (e.g. `linux perf`)
  - E.g. # of cache misses, # of pipeline stalls
- Consider just comparing certain types of instruction counts
  - `objdump -S my_binary | cut -c 33-34 | grep j | wc -l`
- High-resolution timestamping can be useful (e.g. the hardware TSC)
  - Doesn’t need to be in sync with clock time
    - Just needs to be constant across samples
  - If you want actual nanoseconds:
    - Calibrate with wallclock time every few milliseconds

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

(Many trading companies do this)



Summary

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

– Alan Perlis

- Know C++ well, including your compiler
- Know the basics of machine architecture, and how it will impact your code
- Do as much work as possible at compile time
- Aim for very simple runtime logic
- Accurate measurement is essential
- Assume nothing: a lot can be surprising, and compilers, hardware and operating systems are always changing

Thanks for listening!