You're probably leaving some C++ performance "on the table"

Thomas Rodgers
rodgert@twrodgers.com
FCA Design Group

Background

- I’ve been a C++ developer since 1989
  - The last 20+ years in Finance IT
- Member of the ISO C++ Standards Committee
  - SG1 - Concurrency & Parallelism Working Group
  - SG14 - Low Latency/Game Dev Working Group

Today’s talk

- Cover some common performance ‘gotchas’ with C++
- Cover some things the C++ Committee is working on to make Finance Developer’s lives easier

Disclaimer

- This is not a talk on system tuning
- This talk is not strictly geared at HFT
- All of these topics go fairly deep
- This will be at best be a high level introduction to these concepts

An Abstract Machine?

- C and C++ are specified in terms of an *Abstract Machine*
  - simplest imaginary computer that can execute a program in the source language
  - Talks about addressing, bytes, words, etc.

An Abstract Machine?

- Omits many details of real machines
  - Does not talk about things like caches, NUMA, etc.
- The C++ Standard states only that -
  - “Conforming implementations are required to emulate the observable behavior of the abstract machine”

Fundamentals - Alignment

- The view of memory, presented by the C++ *abstract machine* is access at byte-granularity
  - e.g. char c = foo[17]
- Intel CPUs impose a slight penalty for unaligned accesses
- Many others (RISC) architectures, simply abort execution on unaligned access
  - Compilers for these architectures must generate extra code to give the “illusion” of byte-granularity access.

Fundamentals - Alignment

- Compilers align data on some multiple of the underlying hardware’s word size
  - typically 4 or 8 bytes
- operator new() also returns aligned data
  - by default, on Linux, pointers are aligned on 16-byte boundaries

Fundamentals - Alignment

- CPU’s view of data access is at cache-line granularity
  - 64B on Intel
- Depending on where data falls with respect to the cache line boundary, 1-2 references to the next level in the memory hierarchy may be required
- If two cores end up mapping the same 64B to a cache line, modifying potentially different data structures, we have what’s known as “false sharing”

Fundamentals - Alignment

- Safe to assume 64B cache line size for current generation Intel Hardware
  - Other CPUs types may have 32B, 64B, or 128B line sizes

Fundamentals - Alignment

- C++17 provides two new *compile-time determined* constants to make this more portable -
  - std::hardware_destructive_interference_size
    - Minimum offset to prevent false line sharing
  - std::hardware_constructive_interference_size
    - Maximum width to promote true line sharing

Fundamentals - Alignment

```
struct keep_apart {
    alignas(std::hardware_destructive_interference_size) std::atomic<int> foo;
    alignas(std::hardware_destructive_interference_size) std::atomic<int> bar;
};

struct keep_together {
    std::atomic<int> foo;
    std::array<char, 16> bar;
};

static_assert(sizeof(keep_together) <=     std::hardware_constructive_interference_size);
```

Fundamentals - Alignment

- Compilers will correctly respect alignas() for stack locals
- But what about dynamic allocations?
  - Can use posix_memalign() to specify a different alignment
  - C++17 introduced std::aligned_alloc() as a portable alternative
- Unfortunately these options only deal with returning aligned blocks of memory, not objects

Fundamentals - Alignment

```
template<typename T, typename... Args>
unique_ptr<T> make_aligned(Args&&… args) {
    if (auto p = std::aligned_alloc(
                        std::hardware_destructive_interference_size, sizeof(T))) {

            return new (p) T(std::forward<Args>(args)...);

    }
    throw std::bad_alloc();
}
```

Data Structure Choice

- Simple exercise -
  - Load up a std::map<> with a largish amount of randomly generated data
  - Load up a std::vector<> with the same randomly generated data, sort the vector by the same key the map is ordered by
- Both look ups have the same Ɵlg*n* time complexity
- Time 1 million random lookups in each structure
- Which one is faster in terms of absolute wall clock time?

Data Structure Choice

- The vector implementation is **always** faster.

Why?

- std::map<K, V>
  - Typically implemented as a Red-Black tree
  - Nodes are not guaranteed to be adjacent in memory
  - At least 5 words of data per node (parent, left, right pointers, key, and value)
    - The “color” value is usually implemented by stealing a bit from the parent pointer

Why?

- std::vector<T>
  - Typically implemented as three pointers
    - front, back, end-of-allocated storage
  - Data is laid out contiguously
  - Versus std::map<> a sorted vector need only store the Key and Value

Prefer Array-Shaped Data

- If modifications are infrequent, a sorted vector<> will always outperform map<> or set<>
  - Even inserts into a sorted vector<> are not as expensive as you might expect
- Array-Shaped types are much more friendly to the CPU’s pre-fetcher and exploit the relatively high bandwidth of DRAM vs. DRAM’s random access latency
- Node based types (set<>, map<>, list<>, etc.) only make sense if you need their iterator guarantees
  - Deletes do not invalidate outstanding iterators

Valid C++

```
#include <vector>
#include <cstdlib>

int compare(void const* a, void const* b) {
    return ( *(int*)a - *(int*)b );
}

…
   vector<int> ints;
    // load ints up with some data...

   qsort((void*) ints.data(), ints.size(),
                         sizeof(int), compare);

…
```

Valid C++

```
#include <vector>
#include <algorithm>

…
   vector<int> ints;
    // load ints up with some data...

   sort(std::begin(ints), std::end(ints));

…
```

This one is always faster

```
#include <vector>
#include <algorithm>

…
   vector<int> ints;
    // load ints up with some data...

   sort(std::begin(ints), std::end(ints));

…
```

Why?

- qsort type erases the element type to void*
  - std::sort preserves type information
- qsort takes the comparison function by function pointer
  - std::sort uses the definition of < and == for the supplied type
  - std::sort almost always inlines comparisons

Don’t throw away type information

- Use the algorithms defined in <algorithm> and <numeric>
  - Generally optimal and cover a broad range of functionality
  - Preserve type information and are more conducive to inlining than similar C-style APIs, leading to better overall optimization
  - Learning these algorithms is time well spent
- See also - Sean Parent’s ‘C++ Seasoning talk’
  - https://channel9.msdn.com/Events/GoingNative/2013/Cpp-Seasoning

Call optimization

- In general, function pointers are “poison” to an optimizer
  - The compiler must invoke the function and cannot inline it
  - The call to qsort’s comparison function is the primary performance bottleneck
- Same is true, in general, of virtual methods and inheritance
  - Unless, the compiler can statically prove there’s only ever one concrete derivation, aka *de-virtualization*

Call optimization

- Functions or methods defined in separate translation units are not generally inlined either
  - The typical pattern of .h/.cpp per class proliferates method declarations which are not generally considered for inlining

Use Link-Time Optimization

- LTO (-flto) allows cross-translation unit optimizations, including -
  - Method/function inlining
  - De-virtualization
  - Cross function data-flow analysis

Common Idiom for calling C APIs

```
int some_api_function(int fd, char* buf, int* buf_size);

string get_some_value(int fd) {
    string res;
    int sz = 0;
    auto rc = some_api_function(fd, nullptr, &sz);
    if (rc < 0) {
            if (errno != E2BIG)
          throw system_error(errno, system_category());
            res.resize(sz);
        rc = some_api_function(fd, &res.front(), &sz);
    }

    if (rc < 0)
        throw system_error(errno, system_category());
    return res;
}
```

Common Idiom for calling C APIs

- Does the string::resize() call in the example allocate?

Common Idiom for calling C APIs

- Does the string::resize() call in the example allocate?
  - Maybe

SSO

- All common implementations of C++'s *string* implement the “Small String Optimization”

- More generally the “Small Space Optimization”

SSO

- Comes from the following observations
  - Strings are a commonly used type and frequent allocations are expensive
  - Many strings are short
  - We can, at a minimum, use the space normally occupied by the string’s data pointer to hold a string whose length is <= sizeof(char*) - 1
  - Embedding space for the common case provides better data locality

SSO

- Most Standard Library implementations set aside a larger buffer
- Unfortunately -
  - The exact size of this this is implementation defined
  - Not exposed in any portable way
  - Only safe to assume it’s sizeof(char*) - 1

SSO for the rest of us

- Can we get more control over this small space optimization?

SSO for the rest of us

```
#include <boost/container/small_vector.hpp>
using namespace boost::container;

constexpr auto small_buf_size = 256;
using buffer_t = small_vector<char, small_buf_size>;

string_view get_some_value2(int fd, buffer_t& buf) {
    int sz = small_buf_size;
    auto rc = some_api_function(fd, &buf.front(), &sz);
    if (rc < 0) {
        buf.resize(sz);
        rc = some_api_function(fd, &buf.front(), &sz);
    }

    if (rc < 0)
        throw system_error(errno, system_category());
    return string_view(buf.data(), buf.size());
}
```

small_vector

- Allows user to specify some expected small size before allocation occurs
- The most used data structure in clang
- Boost has a version
    - Also has static_vector, where you know up front the exact maximum size
- Under consideration for inclusion into C++20

Return by value

- Pre-C++ 17 most compilers implement the *Return Value Optimization*
  - On return from the called function, the result is already in the right place in the caller’s stack frame
  - Doing std::move(res) out of the called function is potentially a performance pessimization
    - Some work to move is more work than doing nothing

Return by value

- C++17 introduces guaranteed copy elision
  - Mandates RVO
  - Elides copies in the common case of passing a temporary by value
  - Elides copies when throwing *and* catching exceptions by value

Pass Small Types by Value

- If a type is small (typically <= 2-3 machine words in size), and is trivially copyable, pass by value
  - Compilers will pass the value in registers, rather than on the stack
  - Even if the argument is not enregistered, the resulting generated code is as if it had been passed by const& for trivial types

Pass Small Types by Value

```
price_t vwt_price(trade t)
{ return t.price * t.volume; }
```

```
// Dissasembly
vwt_price(trade): # @vwt_price(trade)
  mov eax, psi
  imul rax, rdi
  ret
```

Pass Small Types by Value

```
price_t vwt_price2(trade const& t)
{ return t.price * t.volume; }

// Dissasembly
vwt_price2(trade const&): # @vwt_price2(trade const&)
  mov eax, dword ptr [rdi + 8]
  imul rax, qword ptr [rdi]
  ret
```

Pass Small Types By Value

- True of many standard library types
  - Most iterators, string_view, etc.
- But, **not** shared_ptr<>
  - Copy must perform an atomic increment
  - ~100x more expensive than a non-atomic increment

Lock free isn't cheap

- C++ 11 (and C11) introduced a standard memory model
  - Sequentially Consistent for Data-Race Free Programs
  - Supporting library of portable atomic types and operations
- There is a **lot** of enthusiasm for lock-free algorithms
- However the C++ Standard does not ship any common lock free data-structures (yet)

Lock free isn’t cheap

- Several possible additions for C++20
  - Concurrent queues
  - Split counters
  - atomic_shared_ptr<>
  - RCU - Read Copy Update
  - Hazard Pointers

Lock free isn’t cheap

- On x86 all lock prefixed instructions generate cache coherency traffic
  - Instruction latency measured in 10’s - 100’s of clock cycles
- lock prefixed instructions preclude superscalar/out of order execution by the ALU
- lock prefixed instructions can manipulate at most 128 bits of data atomically
- lock prefixed instructions can manipulate at most a single memory location atomically

Lock free isn’t cheap

- Implementations are limited by underlying hardware to manipulating primitive types
  - You can’t just stick any old class into a lock_free::queue<T>

Lock free isn’t cheap

- Many lock-free algorithms are node based
  - e.g. ‘Michael & Scott’ lock free queues
    - basis for Boost’s lock free queues, as well as java.lang.ConcurrentQueue
- Require allocations
  - not guaranteed to be wait or pre-emption free
- Have poor locality of reference

How bad are mutexes, really?

- Common complaint is the cost of preemption by the operating system (milliseconds of latency)
  - naive Compare-and-Swap (CAS) spin lock will preclude this preemption
  - Generates substantial cache-coherency traffic without techniques already in use by most mutex implementations (e.g. exponential backoff)

std::mutex

- Acquisition is attempted via simple compare-and-swap operation to set some low-order bits in a pointer to the operating system’s lock structure
- Implementation will spin for a while in user space, on failed acquisition
  - Includes some form of backoff, to reduce cache-coherency traffic
  - minimizes chances of preemption on a lightly contended lock
- When all else fails, delegates to the host operating system

Other mutex types

- Boost includes both shared_mutex and upgrade_mutex
  - C++17 standardizes shared_mutex
- possibly useful in read-mostly scenarios where readers can access data race free, but need exclusion during updates
  - If updates are infrequent, may not result in preemption
- Somewhat un-intuitively, often more expensive than a plain mutex