Introduction to Performance Analysis in C++

Written by:

- zhiayang

Last updated: 25 July 2022

Feedback form for Lecture 11 and 12

Feedback form for lecture 11 and 12

1 Introduction

References:

- Agner Fog - optimisation manuals
- Chandler Carruth - Going nowhere faster

In this lecture, we'll briefly delve into how to measure and tune the performance of your C++ application. Note that most of the details covered here are not really specific to C++, and can be applied across other programming languages.

2 Measuring performance

The first important thing to know is *how to measure* the performance of your application. Of course there are *many* other ways to measure performance, but we'll just cover some commonly used tools here.

2.1 Basic: `time` utility

The most basic method of measuring time is to use the built-in `time` utility of Unix, which, as its name might suggest, lets you measure the time that a command takes to execute. Here, we'll just measure the time taken for this simple program:

```cpp
int main() {
  std::vector<int> xs{};
  xs.resize(1 << 21);
  for (size_t i = 0; i < (1 << 21); i++) {
    xs[i] = rand();
  }

  std::sort(xs.begin(), xs.end());
}
```

```
$ time ./sort.out

real    0m0.338s
user    0m0.319s
sys     0m0.011s
```

Snippet 1: A simple program that sorts numbers

If you look at the output of `time`, there are 3 values: `real`, `user`, and `sys`. The first is what we call "wall-clock time", which is the *actual* amount of time that the program took to execute; if you used a stopwatch to time it, this is the number you would get (assuming you press quickly :D).

The next two: `user` is the amount of time that the program spends executing code, and `sys` is the amount of time spent in the kernel waiting for stuff to happen (eg. sleeping, blocked on IO, etc.).

Let's introduce a multithreaded sorter:

```cpp
int main() {
  std::vector<int> xs{};
  xs.resize(1 << 21);
  for (size_t i = 0; i < (1 << 21); i++) {
    xs[i] = rand();
  }

  std::vector<std::thread> ts{};
  for (int i = 0; i < 4; i++) {
    ts.emplace_back([xs]() mutable {  //
      std::sort(xs.begin(), xs.end());
    });
  }

  for (auto& t : ts) {
    t.join();
  }
}
```

```
$ time ./sort-multi.out

real    0m0.407s
user    0m1.286s
sys     0m0.032s
```

Snippet 2: A simple program that sorts numbers, but faster

Here, we see that the `user` time is greater than the `real` time — this means that the program was multithreaded! The `user` time is summed across all threads, which, if the program is doing work on all those threads, will be greater than the actual elapsed time.

2.2 Less basic: `hyperfine` utility

Next, we introduce the hyperfine utility, which is useful for benchmarking programs *against* each other, but can also be used to benchmark single programs. For example, let's look at a worse version of our original sort that *doesn't* reserve the elements upfront:

```cpp
std::vector<int> xs{};
constexpr int N = (1 << 22);
xs.reserve(N);
for (int i = 0; i < N; i++) {
  xs.push_back(rand());
}

std::cout << "xs[0] = "  //
          << xs[0] << "\n";
```

```cpp
std::vector<int> xs{};
constexpr int N = (1 << 22);
// xs.reserve(N);
for (int i = 0; i < N; i++) {
  xs.push_back(rand());
}

std::cout << "xs[0] = "  //
          << xs[0] << "\n";
```

```
$ hyperfine --warmup=1 --runs=3 ./vector-good.out ./vector-bad.out
Benchmark 1: ./vector-good.out
  Time (mean ± σ):      64.5 ms ±   0.4 ms    [User: 55.0 ms, System: 6.4 ms]
  Range (min … max):    63.8 ms …  65.8 ms    42 runs

Benchmark 2: ./vector-bad.out
  Time (mean ± σ):      79.2 ms ±   0.6 ms    [User: 62.4 ms, System: 13.1 ms]
  Range (min … max):    78.3 ms …  82.1 ms    35 runs

Summary
  './vector-good.out' ran
    1.23 ± 0.01 times faster than './vector-bad.out'
```

Snippet 3: Reallocations are slow!

We get some pretty detailed statistics from `hyperfine`, and it even tells us which program was the fastest, and by how much compared to the rest. When comparing various implementations, it's a useful tool to have.

2.3 Less basic: google benchmark

One other tool that we can use is google benchmark, which we'll use for micro-benchmarks — it is designed for running small pieces of code (hence *micro*) repeatedly to measure their performance.

The usage is quite simple:

```cpp
#include <benchmark/benchmark.h>
static void bench_list(benchmark::State& st) {
  // setup:
  std::list<int> xs{};
  for (size_t i = 0; i < 100000; i++) {
    xs.push_back(rand());
  }

  // always loop over `st`
  for (auto _ : st) {
    // actual code to benchmark
    for (auto x : xs) {
      benchmark::DoNotOptimize(x);
    }
  }
}
BENCHMARK(bench_vector);
BENCHMARK_MAIN();
```

Snippet 4: An example of using the google benchmark library

There's some things to note: first, each benchmark function should take in a `benchmark::State&` argument, and we use the `BENCHMARK(...)` macro to tell the library which functions to run as benchmarks. We then use `BENCHMARK_MAIN()` to define the main function, *instead of writing our own*.

Finally, in the benchmark function itself, we have 2 parts: the first part is outside the for-loop, and you can use that to setup various things that need to be used in the loop itself (here, we used it to fill the container). The second part is inside the loop itself — this is the part that is run repeatedly.

The neat thing is that the library automatically determines how many times to call the benchmark function to get a good result, as well as how many iterations of the inner loop to perform. If we run it:

```
$ ./bench.out 2> /dev/null
-------------------------------------------------------
Benchmark             Time             CPU   Iterations
-------------------------------------------------------
bench_vector      65263 ns        64082 ns        11428
bench_list       326960 ns       323113 ns         2081
```

Snippet 5: Example output of running google benchmark

The difference between the two functions is simply that we replaced `std::list` with `std::vector`, and we get much better performance.

2.4 Not basic: `perf` utility

This Linux utility basically gives you the most detailed statistics for your program, including some advanced metrics that you can use to see *what exactly* is causing slowdowns, including cache misses, branch mispredictions, page faults, etc.

It's actually quite a versatile utility, but for our purposes we'll just be using `perf stat`; if we look at our sort program from earlier:

```
$ perf stat -d -- ./sort.out
            229.10 msec task-clock:u
                 0      context-switches:u
                 0      cpu-migrations:u
             2,170      page-faults:u
       688,802,210      cycles:u
        31,433,908      stalled-cycles-frontend:u
        86,806,428      stalled-cycles-backend:u
       494,986,884      instructions:u

       110,804,798      branches:u
        18,495,265      branch-misses:u
       208,888,773      L1-dcache-loads:u
         1,887,374      L1-dcache-load-misses:u
   <not supported>      LLC-loads:u
   <not supported>      LLC-load-misses:u

       0.233489083 seconds time elapsed

       0.225088000 seconds user
       0.003368000 seconds sys
```

Snippet 6: An example of running `perf stat` with detailed (`-d`) output

We see quite a lot of metrics, which we can use to figure out where our program is being bottlenecked; we'll talk about those numbers in more detail later. Note that we used `-d` to tell perf to output more detailed statistics; you can specify it up to 3 times (eg. `perf stat -d -d -d ...`) to get even more numbers.

3 Compiler optimisations

Now that we know how to measure the performance of our programs, we can start talking about how to increase their performance. The first, and usually most taken-for-granted area, is the fact that the *compiler* performs optimisations.

By default, compilers *do not* optimise your program — it's equivalent to passing `-O0`. To enable optimisations, we can use `-O1`, `-O2`, or `-O3` to enable increasingly powerful optimisations.

Basically, the summary from this section is that compilers are *very smart* nowadays — probably smarter than you.

3.1 Loop optimisations

The first class of optimisations we'll talk about are loop optimisations; many programs have tight loops that can benefit a lot from these kind of optimisations.

3.1.1 Loop unrolling

If the compiler has some information about the number of iterations of your loop, it can choose to *unroll* it — either partially or fully. This eliminates (some of) the conditional branch(es), and can save you some instructions.

For example, if we had a loop like this, the compiler, knowing the loop bounds completely (since they are constants), might fully unroll the loop into this:

```
for (int i = 0; i < 5; i++)
  foo(i);
```

```
foo(0);
foo(1);
foo(2);
foo(3);
foo(4);
```

Snippet 7: An example of loop unrolling

If we look at the assembly, we find that it is indeed the case:

```
$ make -B loop.out
clang++ -g -O3 -std=c++20 -pthread -Wall -Wextra -Wconversion -fomit-frame-pointe
r    loop.cpp   -o loop.out
$ objdump --no-addresses -d loop.out | grep _Z1av -A15
<_Z1av>:
  50                      push   %rax
  31 ff                   xor    %edi,%edi
  e8 e8 ff ff ff          call   <_Z3fooi>
  bf 01 00 00 00          mov    $0x1,%edi
  e8 de ff ff ff          call   <_Z3fooi>
  bf 02 00 00 00          mov    $0x2,%edi
  e8 d4 ff ff ff          call   <_Z3fooi>
  bf 03 00 00 00          mov    $0x3,%edi
  e8 ca ff ff ff          call   <_Z3fooi>
  bf 04 00 00 00          mov    $0x4,%edi
  e8 c0 ff ff ff          call   <_Z3fooi>
  31 c0                   xor    %eax,%eax
  59                      pop    %rcx
  c3                      ret
```

Snippet 8: The disassembly showing that the loop was unrolled

Even if the loop bounds are not fully known, the compiler can *still* optimise it; for instance, by emitting code that does something like this:

```
int iters = ...;
while(true) {
  if (iters > 8) {
    body(); body(); body(); body();
    body(); body(); body(); body();
    iters -= 8;
  } else if (iters > 4) {
    body(); body(); body(); body();
    iters -= 4;
  } else {
    body();
    iters--;
  }
}
```

Snippet 9: An example of partial loop unrolling

Of course the real mechanisms are more sophisticated (there might be cost analysis on the body to see if unrolling is even worth it, perhaps due to increased code size, etc.), but this gives the general idea.

3.1.2 Loop hoisting (loop-invariant code motion)

If computations performed in the loop body do not depend on any loop variables, they are said to be *loop-invariant* — in these cases, the compiler can move their computation outside the loop, so that it's only done once.

For instance, given something like the code on the left, the compiler can trivially transform it into the code on the right:

```
// global p
for (int i = 0; i < v.size(); i++) {
  int x = 5 * p;
  int y = x * 420;
  v[i] = y;
}
```

```
// global p
int x = 5 * p;
int y = x * 420;
for (int i = 0; i < v.size(); i++) {
  v[i] = y;
}
```

Snippet 10: Here, `x` and `y` are loop-invariant and their computations are moved out of the loop

Of course, the compiler needs to be sure that the code is actually loop-invariant; if we had used (for instance) `v.size()` or some other *possibly* loop-variant code in the body, then this optimisation might not be able to be performed.

Furthermore, in this particular case, the compiler doesn't need keep loading `p` in the body even though it's a global, since it assumes that no data races occur (since they are UB) — nobody else should be writing to the variable.

Finally, while this seems like pretty trivial computations to optimise, if they appear in a tight loop with many iterations, even saving a couple of arithmetic instructions might be worth it.

3.1.3 Vectorisation

The last loop-related class of optimisations we'll talk about is vectorisation. First, we should understand *what* vectorised code is — it refers code that operates on entire arrays of values at once, rather than just a single value. We might know this as SIMD instructions — single instruction, multiple data.

On x86, we have SSE2/3/4 and AVX instructions, and on ARM we have NEON. These are instructions that load anywhere from 128 to 512 bits of data from memory into similarly-sized registers, and perform operations on all of them *simultaneously*. As you might imagine, judicious use of SIMD can lead to increased program performance.

While you can manually write these vector instructions using SIMD intrinsics, compilers are also able to do this for us — this is known as *automatic vectorisation*.

If we look at this code, which simply fills a `std::array` with a random value:

Snippet 11: Filling an array with a random value

First, let's try compiling this with full optimisations, but SSE *disabled* (using the `-mno-sse` flag):

```
$ make -B loop.no-sse.out
clang++ -g -O3 -std=c++20 -pthread -Wall -Wextra -Wconversion -fomit-frame-pointe
r    loop.cpp   -mno-sse -o loop.no-sse.out
$ objdump --no-addresses -d loop.no-sse.out
<__Z1bv>:
  53                          push   %rbx
  50                          push   %rax
  48 89 fb                    mov    %rdi,%rbx
  48 c7 47 78 00 00 00 00     movq   $0x0,0x78(%rdi)
  48 c7 47 70 00 00 00 00     movq   $0x0,0x70(%rdi)
  48 c7 47 68 00 00 00 00     movq   $0x0,0x68(%rdi)
  <truncated>
  e8 0f 01 00 00              call   <_main+0x3c>   # rand()
  89 03                       mov    %eax,(%rbx)
  89 43 04                    mov    %eax,0x4(%rbx)
  89 43 08                    mov    %eax,0x8(%rbx)
  89 43 0c                    mov    %eax,0xc(%rbx)
  <truncated>
  d1 23                       shll   (%rbx)
  d1 63 04                    shll   0x4(%rbx)
  d1 63 08                    shll   0x8(%rbx)
  d1 63 0c                    shll   0xc(%rbx)
  <truncated>
  48 89 d8                    mov    %rbx,%rax
  48 83 c4 08                 add    $0x8,%rsp
  5b                          pop    %rbx
  c3                          ret
```

Snippet 12: The loop program without SSE instructions

The truncated part is just the same stuff repeated many times — 32 to be exact — one per element in the array. Now, if we remove the artificial limitation[1] and compile *without*, we see some SSE instructions appearing:

```
$ make -B loop.out
clang++ -g -O3 -std=c++20 -pthread -Wall -Wextra -Wconversion -fomit-frame-pointe
r    loop.cpp   -o loop.out
$ objdump --no-addresses -d loop.out | grep _Z1bv -A26
<_Z1bv>:
  53                      push   %rbx
  48 89 fb                mov    %rdi,%rbx
  0f 57 c0                xorps  %xmm0,%xmm0
  0f 11 47 70             movups %xmm0,0x70(%rdi)
  0f 11 47 60             movups %xmm0,0x60(%rdi)
  0f 11 47 50             movups %xmm0,0x50(%rdi)
  0f 11 47 40             movups %xmm0,0x40(%rdi)
  0f 11 47 30             movups %xmm0,0x30(%rdi)
  0f 11 47 20             movups %xmm0,0x20(%rdi)
  0f 11 47 10             movups %xmm0,0x10(%rdi)
  0f 11 07                movups %xmm0,(%rdi)
  e8 25 fe ff ff          call   <rand@plt>
  66 0f 6e c0             movd   %eax,%xmm0
  66 0f fe c0             paddd  %xmm0,%xmm0
  66 0f 70 c0 00          pshufd $0x0,%xmm0,%xmm0
  f3 0f 7f 03             movdqu %xmm0,(%rbx)
  f3 0f 7f 43 10          movdqu %xmm0,0x10(%rbx)
  f3 0f 7f 43 20          movdqu %xmm0,0x20(%rbx)
  f3 0f 7f 43 30          movdqu %xmm0,0x30(%rbx)
  f3 0f 7f 43 40          movdqu %xmm0,0x40(%rbx)
  f3 0f 7f 43 50          movdqu %xmm0,0x50(%rbx)
  f3 0f 7f 43 60          movdqu %xmm0,0x60(%rbx)
  f3 0f 7f 43 70          movdqu %xmm0,0x70(%rbx)
  48 89 d8                mov    %rbx,%rax
  5b                      pop    %rbx
  c3                      ret
```

Snippet 13: The loop program with SSE instructions

It's now short enough that we didn't need to truncate it! Of course the compiler *also* decided to unroll the loop, but we see far fewer instructions — only sets of 8 — because each instruction actually operates on *four* `int`s at a time.

The specifics on how exactly these SSE instructions work are out of the scope of this lecture (they're *very* complicated), but just know that in this case, `xmm0` is a 128-bit register, which can hold 4 32-bit integers.

Finally, we should *actually* tell the compiler to use the full capabilities of our CPU. Assuming that we only want to run our compiled program on the CPU that we're compiling for, we should use `-march=native`, which tells the compiler to generate code that takes full advantage of all the features of the CPU that we're compiling on.

If we take one last look at the assembly:

```
$ make -B loop.march-native.out
clang++ -g -O3 -std=c++20 -pthread -Wall -Wextra -Wconversion -fomit-frame-pointe
r    loop.cpp   -march=native -o loop.march-native.out
$ objdump --no-addresses -d loop.march-native.out | grep _Z1bv -A20
<_Z1bv>:
  53                      push   %rbx
  48 89 fb                mov    %rdi,%rbx
  c5 f8 57 c0             vxorps %xmm0,%xmm0,%xmm0
  c5 fc 11 47 60          vmovups %ymm0,0x60(%rdi)
  c5 fc 11 47 40          vmovups %ymm0,0x40(%rdi)
  c5 fc 11 47 20          vmovups %ymm0,0x20(%rdi)
  c5 fc 11 07             vmovups %ymm0,(%rdi)
  c5 f8 77                vzeroupper
  e8 2d fe ff ff          call   <rand@plt>
  c5 f9 6e c0             vmovd  %eax,%xmm0
  c5 f9 fe c0             vpaddd %xmm0,%xmm0,%xmm0
  c4 e2 7d 58 c0          vpbroadcastd %xmm0,%ymm0
  c5 fe 7f 03             vmovdqu %ymm0,(%rbx)
  c5 fe 7f 43 20          vmovdqu %ymm0,0x20(%rbx)
  c5 fe 7f 43 40          vmovdqu %ymm0,0x40(%rbx)
  c5 fe 7f 43 60          vmovdqu %ymm0,0x60(%rbx)
  48 89 d8                mov    %rbx,%rax
  5b                      pop    %rbx
  c5 f8 77                vzeroupper
  c3                      ret
```

Snippet 14: The loop program with AVX instructions

We see that it got *even* shorter! Now, the compiler is using AVX instructions — which are not available on all CPUs — which can operate on 256 bits at a time. Thus, we only need 4 instructions to cover our 64 integers, which is exactly what we see here.

One thing to note is that you can also target a specific CPU microarchitecture, for example `znver3` for AMD Zen 3 CPUs, or `alderlake` for Intel Alder Lake CPUs.

▼ The difference between `-march` and `-mtune`

If you look at the documentation for GCC and Clang, you'll realise that there are two very similar flags, `-mtune` and `-march`.

The key point is that `-march` lets the compiler assume the microarchitecture, which means that generated code might use instructions that *do not exist* on earlier CPUs. For instance, if we tried to run a program compiled with `-march=alderlake` on an older CPU that did not support AVX instructions (eg. an Athlon 64), then it would crash.

On the other hand, `-mtune` just tells the compiler to *tune* the program (eg. by arranging instructions, assuming things about the micro-op cache, number of execution units, etc.) for the specified microarchitecture, but *not* to use instructions that do not exist on the microarchitecture specified by `-march` (or the oldest one, if not specified); this means that it can run on older CPUs.

3.2 Inlining

The next most important optimisation to understand is *function inlining*, which is where the compiler essentially pastes the body of a function into the caller, instead of emitting a call instruction. This saves two branches (the call and return), and also lets the compiler optimise certain calling-convention things (eg. not needing to save certain registers).

For the most part, the compiler knows best — they have a bunch of heuristics that determine whether a function should be inlined or not. If we look at our loop function again:

```
int a() {
  for (int i = 0; i < 5; i++)
    foo(i);

  return 0;
}

int main() {
  a();
}
```

```
$ objdump -d loop2.out | grep "main>:" -A16
<_main>:
    push   %rax
    xor    %edi,%edi
    call   100003ef0 <foo(int)>
    mov    $0x1,%edi
    call   100003ef0 <foo(int)>
    mov    $0x2,%edi
    call   100003ef0 <foo(int)>
    mov    $0x3,%edi
    call   100003ef0 <foo(int)>
    mov    $0x4,%edi
    call   100003ef0 <foo(int)>
    xor    %eax,%eax
    pop    %rcx
    ret
```

Snippet 15: An example of code inlining (and loop unrolling)

Here, we see that the compiler inlined `a`, and while it was doing that, also unrolled the loop within. We did *not* need to mark `a` as `inline`; as mentioned in the first lecture, the `inline` nowadays has no bearing on whether a compiler inlines a function or not, and is only used to fiddle with multiple definitions[2].

3.2.1 Controlling function inlining

While compilers might be Very Smart, you *might* be Even Smarter, so there *are* mechanisms to let you control inlining of functions. In fact, it was already used in this example, just… hidden. If we zoom out a little bit more:

```
[[gnu::noinline]] void foo(int i) {
  k += i;
}
int a() {
  for (int i = 0; i < 5; i++)
    foo(i);

  return 0;
}
```

Snippet 16: An example of a function attribute that disables inlining

Here, we used the `gnu::noinline` attribute, which, as its name might suggest, prevents inlining of the function. Since this is not a standard attribute but a compiler-specific one, we prefix it with `gnu::`.

On the other hand, if we wanted to *force* a function to be inlined, we can use `[[gnu::always_inline]]`, which does exactly what it says.

3.2.2 Inlining considerations

Some things to consider when manually specifying whether or not to inline a function:

1. how often the function is called (hot or cold); if it's only called in a rare code path, then it's probably not worth it to inline
2. how large the function is; small functions have the greatest benefit from inlining, and large functions can also fill up the instruction cache
3. how complex the function is; complex functions tend to use more registers, which increases the register pressure[3] on the caller function if it gets inlined

Most of the time, you should leave this up to the compiler.

3.3 Other optimisations

Other optimisations are not so important to talk about, because they're more obvious. For instance, constant propagation just means that the compiler tries hard to simplify expressions involving constants as far as possible:

```
int x = 69;
int y = x + 420;
int z = y * 50;

printf("%d\n", z);
```

```
$ objdump -d misc.out | grep "main>:" -A9
<_main>:
    push   %rbp
    mov    %rsp,%rbp
    lea    0x33(%rip),%rdi
    mov    $0x5f82,%esi
    xor    %eax,%eax
    call   100003f8c <_main+0x1c>
    xor    %eax,%eax
    pop    %rbp
    ret
```

Snippet 17: An example of constant propagation

As expected, it simplified the expression to `0x5f82`, which is exactly 24450 in hex. One way to *help* the compiler perform these kinds of optimisations is to make variables `const` (or better yet, `constexpr`) whenever possible.

As for dead code elimination, it does exactly that — eliminate code that the compiler can prove will *never* be called; this includes branches that are always true or false, or assignments to variables that are "dead" (ie. will always be overwritten by another store).

Finally, we can talk a little about strength reduction; it's the optimisation that reduces something like `a * 2` to `a + a`, or to `a << 1`. Some old-school programmers might manually write `a << 1` instead of just `a * 2`, but just trust the compiler — they're very smart these days.

4 Memory architecture overview

In order to know how to make programs fast, we must understand how memory is organised; this is prerequisite knowledge, but in short there are (usually) 3 layers of cache — L1, L2, and L3 — followed by main memory, in increasing latency and size.

For reference, the latency for cache *hits* are: ~1ns for L1, ~4ns for L2, ~20ns for L3, ~100ns for main memory; it stands to reason that we want things to stay in the cache as much as possible

4.1 Cache locality

There are two kinds of locality to note when talking about caches — spatial locality and temporal locality. The former talks about objects being close together in memory in terms of *addresses*, and the latter about access to memory (a specific address, or set of addresses) happening together *in time*.

When trying to optimise cache hits, we should generally try to improve both kinds of locality for our data, and we'll cover more about this below.

4.2 Cache sharing

One important thing to note is that caches are organised into *cache lines* — on most modern processors, they are 64 bytes large. Note that every memory read always first goes through the cache[4]. As a corollary, this means that every memory read is done in chunks of at least 64 bytes; this plays a part in spatial locality.

If your objects are smaller than the cache line size, then some sharing will happen.

▼ **Why do I see some people saying that cache lines are 128 bytes?**

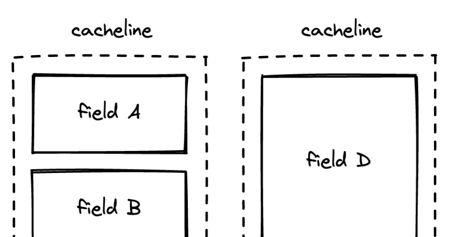
This is due to a specific quirk of a specific CPU manufacturer; on some Intel CPUs, the prefetcher reads cache lines in *pairs*, so they read 128 bytes instead of 64.

It's pretty much black magic though, so don't worry too much about it — either 64 or 128 will work, and if you really need the performance for whatever reason, just benchmark it.

Some further reading: stackoverflow.

4.2.1 True sharing

The constructive version of sharing is *true* sharing, which is when related objects end up in the same cache line. For example, when reading a struct from memory, adjacent fields have a higher chance of ending up in the same cache line, so that reading certain fields is faster due to it already being in cache.



An illustration of cache lines

In this example, if we read any of the fields A, B, or C, then all of them will be loaded into cache, and subsequent accesses will be a lot faster. So, it's generally a good idea to group fields that are frequently accessed together into the same cache line.

In C++, we can use `std::hardware_constructive_interference_size` to get the maximum size of contiguous memory that promotes true sharing — ie. the size of a cache line.

4.2.2 False sharing

The destructive version of sharing is *false* sharing, which is when objects that don't belong together end up in the same cache line. This is really only a problem in multithreaded programs, on architectures with coherent caches (eg. x86).

Cache coherency is a completely separate topic, but the gist of it is that on multi-core systems, memory writes by a core need to be propagated to the caches of other cores on the system, so that they do not read stale data from their outdated caches.

Suppose that we have two threads running on different cores:

```
struct Foo {
  int a;
  int b;
};

auto f = new Foo();
auto t1 = std::thread([f]() {
  for (int i = 0; i < 10'000'000; i++)
    f->a++;
});
auto t2 = std::thread([f]() {
  for (int i = 0; i < 10'000'000; i++)
    f->b++;
});

t1.join();
t2.join();
```

Snippet 18: A simple test program to illustrate false sharing

They operate on different fields in the struct, but what if the fields were in the same cache line? Then when one thread writes to `a` (or `b`), it forces the other thread to evict that cache line, which must then be reloaded because it needs to read from it to increment; this forces the other thread's cache to be evicted…

Evidently, false sharing is bad for performance, since it keeps forcing the memory that we want cached out of the cache. A solution to this problem is to add padding between the two fields so that they will go in separate cache lines, like so:

```
struct Foo {
  alignas(std::hardware_destructive_interference_size) int a;
  alignas(std::hardware_destructive_interference_size) int b;
};

auto f = new Foo();
auto t1 = std::thread([f]() {
  for (int i = 0; i < 10'000'000; i++)
    f->a++;
});
auto t2 = std::thread([f]() {
  for (int i = 0; i < 10'000'000; i++)
    f->b++;
});

t1.join();
t2.join();
```

Snippet 19: Using `alignas` to fix false sharing

Of course this makes the struct larger, but there's a price to pay for everything. We can compare the performance of these two programs using our old friend hyperfine:

```
$ hyperfine --warmup 1 ./false.out ./false2.out
Benchmark 1: ./false.out
  Time (mean ± σ):      49.9 ms ±   5.6 ms    [User: 89.9 ms, System: 1.0 ms]
  Range (min … max):    40.4 ms …  63.2 ms    57 runs

Benchmark 2: ./false2.out
  Time (mean ± σ):      12.1 ms ±   0.5 ms    [User: 19.4 ms, System: 0.7 ms]
  Range (min … max):    11.4 ms …  14.7 ms    169 runs

Summary
  './false2.out' ran
    4.13 ± 0.49 times faster than './false.out'
```

Snippet 20: Benchmarking the two programs

As expected, the program that split the two variables into separate cache lines performed better.

4.3 Instruction cache

As a small aside, it should be obvious that the instruction bytes also need to be loaded from memory, so it makes sense for there to also be a cache for instructions. On most CPUs, L2 and L3 caches are unified (ie. hold both data and code), but L1 cache is usually split into L1d (data) and L1i (instruction).

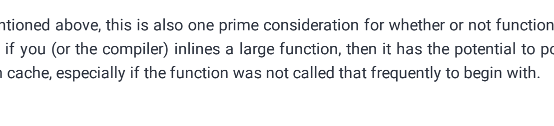
Just as how we would optimise data locality to keep our data cache fresh, we should also optimise code locality to keep the instruction cache fresh. For example, complex code that jumps all over the place can perform worse, since there's a higher chance that the target function is not in the instruction cache due to poor spatial locality.

The good thing is that for the most part, the compiler performs most of this for you; it can classify functions into "hot" and "cold", and tends to put all the hot functions in one group and the cold functions in another (address wise) to get good spatial locality.

As we mentioned above, this is also one prime consideration for whether or not functions should be inlined; if you (or the compiler) inlines a large function, then it has the potential to pollute the instruction cache, especially if the function was not called that frequently to begin with.

4.4 Prefetching & locality

Another factor to consider is the CPU prefetcher, which pre-fetches data that it thinks will be accessed next, based on past access patterns and data locality. For instance, consider sequential array access:



An example of prefetching for sequential access

When we access the first few elements, the CPU can prefetch subsequent elements for us (load them into cache), so that the next access will be faster. Note that this is distinct from cache lines; the prefetcher may load more than one cache line at a time.

Of course, this means that the reverse — non-sequential access — can suffer a penalty, since the prefetcher probably doesn't know what to do. Let's compare two programs, one that accesses sequentially, and one that accesses randomly:

```
std::array<size_t, N> idx{};
std::generate(idx.begin(),  //
              idx.end(),    //
              []() {        //
                return rand() % N;
              });
```

```
size_t i = 0;
std::array<size_t, N> idx{};
std::generate(idx.begin(),  //
              idx.end(),    //
              [&i]() {      //
                return i++;
              });
```

```
std::vector<int> elems{};
std::generate_n(std::back_inserter(elems), N, &rand);

for (auto _ : st) {
  int sum = 0;
  for (auto i : idx) {
    sum += elems[i];
  }
  std::cerr << "sum = " << sum << "\n";
}
```

Snippet 21: Accessing a vector sequentially and randomly

If we run this with google benchmark, we see the following results:

```
$ ./prefetch.out 2> /dev/null
------------------------------------------------------------
Benchmark                 Time             CPU   Iterations
------------------------------------------------------------
bench_random        1887808 ns      1884839 ns          367
bench_sequential     622632 ns       622044 ns         1035
```

Snippet 22: The benchmarking results

Clearly, the sequential access is faster. We can also look at `perf stat` to check cache hits and misses:

```
$ perf stat -d -e task-clock,   \
    l2_request_g1.rd_blk_l,     \
    l2_cache_req_stat.ls_rd_blk_c -- ./prefetch-rng.out

 Performance counter stats for './prefetch-rng.out':

           991.93 msec task-clock:u
     1,061,182,148      L1-dcache-loads:u              # L1 loads
       561,462,773      L1-dcache-load-misses:u        # L2 misses: 52.91%


$ perf stat -d -e task-clock,   \
    l2_request_g1.rd_blk_l,     \
    l2_cache_req_stat.ls_rd_blk_c -- ./prefetch-seq.out

 Performance counter stats for './prefetch-seq.out':

           337.57 msec task-clock:u
     1,032,832,563      L1-dcache-loads:u              # L1 loads
        96,916,467      L1-dcache-load-misses:u        # L1 misses: 9.38%
```

Snippet 23: Examining cache hit rate with `perf stat`

As might be expected, we see that random access has a very high cache miss rate, whereas sequential access has a much lower miss rate.

4.4.1 Row-major vs column-major array access

Back in the first lecture, we mentioned that C++ (and C) use row-major ordering for arrays, looking like this:

Row major array ordering

Now that we know about caches, it should be clear why accessing arrays by rows would be faster than columns:

```
// array-row: accessing by rows
constexpr size_t N = 4096;
auto array = new int[N][N];

int sum = 0;
for (size_t r = 0; r < N; r++) {
  for (size_t c = 0; c < N; c++) {
    sum += array[r][c];
  }
}
```

```
// array-col: accessing by columns
constexpr size_t N = 4096;
auto array = new int[N][N];

int sum = 0;
for (size_t c = 0; c < N; c++) {
  for (size_t r = 0; r < N; r++) {
    sum += array[r][c];
  }
}
```

```
$ hyperfine ./array-row.out ./array-col.out
Benchmark 1: ./array-row.out
  Time (mean ± σ):      31.3 ms ±   1.0 ms  [User: 12.3 ms, System: 16.1 ms]
  Range (min … max):    29.6 ms …  34.7 ms  81 runs

Benchmark 2: ./array-col.out
  Time (mean ± σ):     172.0 ms ±   3.4 ms  [User: 150.3 ms, System: 17.9 ms]
  Range (min … max):   168.0 ms … 179.5 ms  16 runs

Summary
  './array-row.out' ran
    5.49 ± 0.21 times faster than './array-col.out'
```

Snippet 24: Comparing the performance of row-major access versus column-major access

Again, this result should not be a surprise. The prefetcher might be contributing to the performance difference, but one other factor might be cache eviction; since we are accessing elements in the same cache line in the row-major case, access is usually fast.

For the column-major access, consecutive elements probably won't be in the same cache line, so for a large array, we might end up evicting entries from the cache, so that by the time we get back to the first row, it has already been evicted from cache.

Yet another factor is automatic vectorisation; since the array bounds are known at compile-time, the compiler is able to better unroll the loop. And better yet, since in the row-major iteration, consecutive elements are adjacent in memory, SSE/AVX instructions can be used to sum up all of them in one go.

If we slightly modify the example to compile without SSE instructions (`-mno-sse`) and shrink the array so that it fits in L1 cache (~48KiB on my machine):

```
// array-row: accessing by rows
constexpr size_t N = 96;
auto array = new short[N][N];
int sum = 0;
while (k-- > 0) {
  for (size_t r = 0; r < N; r++) {
    for (size_t c = 0; c < N; c++) {
      sum += array[r][c];
    }
  }
}
```

```
// array-col: accessing by columns
constexpr size_t N = 96;
auto array = new short[N][N];
int sum = 0;
while (k-- > 0) {
  for (size_t c = 0; c < N; c++) {
    for (size_t r = 0; r < N; r++) {
      sum += array[r][c];
    }
  }
}
```

```
$ hyperfine ./array-row2-nosse.out ./array-col2-nosse.out
Benchmark 1: ./array-col2-nosse.out
  Time (mean ± σ):      49.8 ms ±   0.4 ms  [User: 47.2 ms, System: 0.6 ms]
  Range (min … max):    49.1 ms …  51.4 ms  53 runs

Benchmark 2: ./array-row2-nosse.out
  Time (mean ± σ):      50.3 ms ±   0.8 ms  [User: 47.3 ms, System: 0.9 ms]
  Range (min … max):    48.8 ms …  52.0 ms  53 runs

Summary
  './array-col2-nosse.out' ran
    1.01 ± 0.02 times faster than './array-row2-nosse.out'
```

Snippet 25: Disabling SSE and making everything fit in cache makes this a fair fight

Then the two access patterns are comparable in performance.

5 Execution architecture overview

Other than the memory architecture, we should also understand the *execution* architecture of the CPU, to know how our code is being run. This is quite a complicated topic, and there's really not much we can influence from a software perspective, so this won't be a particularly long chapter.

5.1 Superscalar and out-of-order execution

Most modern processors are superscalar and out-of-order, which means that multiple instructions can execute at the same time, and also execute in a different order than they were written in the original program.

The instruction scheduler in the CPU figures out the dependencies between instructions, and schedules them appropriately to the execution units to be executed. For example, given the following assembly:

```
add %rax, %rdi
add %rbx, %rsi
sub %rdi, %rsi
```

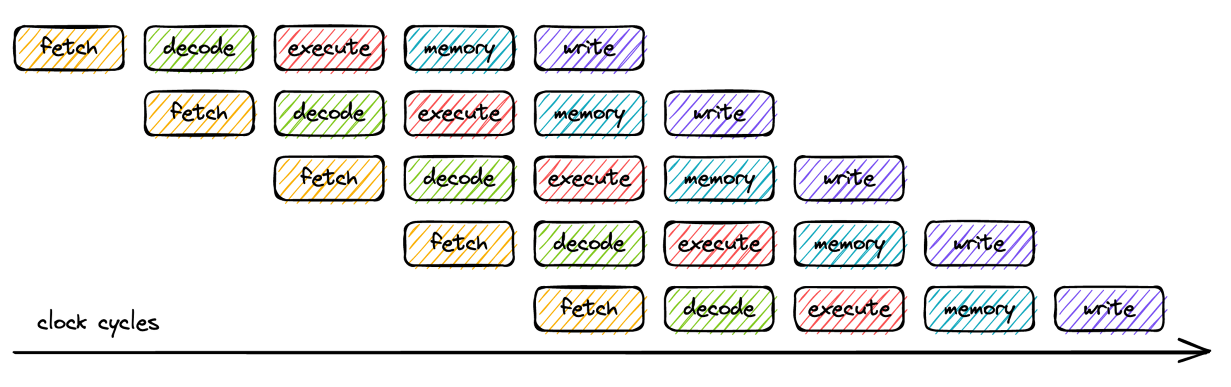
The CPU can execute the two add instructions simultaneously, because they do not have dependencies between each other. Of course, this assumes that the CPU has enough execution units free to do this (in particular, arithmetic units). After the results from both are available, the CPU can then execute the last instruction.

One point to note about execution units is that for CPUs with simultaneous multithreading (aka Intel hyperthreading), the two "threads" on a core usually share the core's execution units.

Lastly, at least on x86, the processors also speculatively execute instructions, and either make the results visible if the instruction was *actually* executed, or rollback the changes if not. Speculative execution is one of the leading causes of CPU vulnerabilities nowadays.

5.2 Pipelined execution

More importantly though, modern CPUs have very deep pipelines, on the order of 10-20 stages. As a quick recap, a pipelined CPU splits instruction execution into stages; this diagram should be familiar to you already:



The classic 5-stage pipeline

In order to target 100% utilisation (avoid stalling the pipeline), the CPU must predict whether a conditional branch will be taken or not, and speculatively start to fetch and decode the instructions at the predicted site. If the prediction is wrong, then the pipeline must be flushed.

5.3 Branching and branch prediction

This is the problem that we're interested in as a result of CPUs having deep pipelines — the penalty for misprediction becomes very high since *a lot* of work will need to be discarded.

While we can't directly control the branch predictor, we can make its job easier by making our code very *predictable*. Let's look at an example to show that having predictable code is valuable:

```cpp
constexpr int N = 1048576;
std::mt19937_64 rng{420};
uniform_int_distribution d{-N, N};

std::vector<int> xs{};
xs.reserve(2097152);
std::generate_n(  //
    std::back_inserter(xs),
    2 * N,
    [&]() { return d(rng); });
```

```cpp
constexpr int N = 1048576;
std::mt19937_64 rng{420};
uniform_int_distribution d{0, N};

std::vector<int> xs{};
xs.reserve(2097152);
size_t k = 0;
std::generate_n(
    //
    std::back_inserter(xs),
    2 * N,
    [&]() {
      return d(rng) *  //
             (k++ > N ? -1 : 1);
    });
```

```cpp
[[gnu::noinline]] static void add() {
  final++;
}

[[gnu::noinline]] static void sub() {
  final--;
}
```

```cpp
for (int i = 0; i < iters; i++) {
  for (auto i : xs) {
    if (i > 0)
      add();
    else
      sub();
  }
}
```

Snippet 26: Comparing the performance between predictable branches and unpredictable ones

On the left setup, we just have an array of random numbers that are uniformly distributed between large negative and positive bounds. On the right setup, we ensure that the first half of the array is positive, and the second half is negative.

The loop body simply calls `add()` if the number is greater than zero, and `sub()` otherwise.

Apart from the arrangement of elements in the array, everything else is exactly the same, and the amount of work done is also exactly the same — so we would expect these programs to perform similarly, right? Well, since we've already gone through all this setup, the answer should be obvious :D.

```
$ hyperfine --warmup 1 ./predict1.out ./predict2.out
Benchmark 1: ./predict1.out
  Time (mean ± σ):     427.4 ms ±   1.2 ms  [User: 419.5 ms, System: 4.4 ms]
  Range (min … max):   425.7 ms … 429.5 ms  10 runs

Benchmark 2: ./predict2.out
  Time (mean ± σ):     241.8 ms ±   1.8 ms  [User: 234.6 ms, System: 4.3 ms]
  Range (min … max):   239.4 ms … 244.2 ms  12 runs

Summary
  './predict2.out' ran
    1.77 ± 0.01 times faster than './predict1.out'
```

```
$ perf stat -d -- ./predict1.out

 Performance counter stats for './predict1.out':

            214.46 msec task-clock:u
       147,507,770      branches:u        #
        16,697,337      branch-misses:u   # 11.32%

$ perf stat -d -- ./predict2.out

 Performance counter stats for './predict2.out':

            115.49 msec task-clock:u
       144,901,045      branches:u        #
            40,445      branch-misses:u   # 0.03%
```

Snippet 27: Using hyperfine and perf stat to analyse the performance

Again, after all the setup above, this should not be surprising; we indeed see that `predict2` has far fewer branch misses (~500x) than `predict1`.

Note that we had to make `add` and `sub` `noinline`, otherwise the compiler would completely eliminate the branch.

5.3.1 `[[likely]]` and `[[unlikely]]`

You might have noticed that there are the `[[likely]]` and `[[unlikely]]` attributes that can be used like so:

```cpp
void foo(int x) {
  if (x > 0) [[likely]] {
    // stuff
  } else [[unlikely]] {
    // other stuff
  }
}
```

Snippet 28: An example of using the likely and unlikely attributes

While this might seem cool, it doesn't actually *directly* influence the branch predictor (at least on most architectures). Rather, it lets the compiler make more informed decisions about how to arrange the code. For instance, it might know that on a certain CPU architecture, forward branches are usually predicted true, so it might put the likely block after the branch.

5.4 Branch-free code

Reference: Andrei Alexandrescu - Speed is found in the minds of the people

Since branching mispredictions are so costly, why don't we just write code that doesn't branch? You might think that code that doesn't branch can't be very useful, since we can't check conditions. Thanks to the implicit conversions in C++, we can convert conditions (boolean values) to either 1 or 0, and perform arithmetic with those values.

Let's look at a simple example that counts the number of non-negative integers in an array; we'll reuse the setup code from before, and just focus on the loop bodies:

```cpp
auto xs = make_vec();
for (int i = 0; i < iters; i++) {
  for (auto i : xs) {
    if (i > 0) total++;
  }
}
```

```cpp
auto xs = make_vec();
for (int i = 0; i < iters; i++) {
  for (auto i : xs) {
    total += (i > 0);
  }
}
```

```
$ hyperfine --warmup 1 ./branch.out ./branchless.out
Benchmark 1: ./branch.out
  Time (mean ± σ):     179.6 ms ±  12.3 ms  [User: 171.7 ms, System: 5.1 ms]
  Range (min … max):   164.1 ms … 200.4 ms  15 runs

Benchmark 2: ./branchless.out
  Time (mean ± σ):      38.7 ms ±  12.7 ms  [User: 34.4 ms, System: 3.1 ms]
  Range (min … max):    27.0 ms …  72.2 ms  76 runs

Summary
  './branchless.out' ran
    4.64 ± 1.56 times faster than './branch.out'
```

```
$ perf stat -d -- ./branch.out

 Performance counter stats for './branch.out':

            170.51 msec task-clock:u
       606,903,600      cycles:u
       108,651,794      branches:u
        19,682,655      branch-misses:u

$ perf stat -d -- ./branchless.out

 Performance counter stats for './branchless.out':

             40.95 msec task-clock:u
       112,072,746      cycles:u
        44,924,112      branches:u
            11,702      branch-misses:u
```

Snippet 29: Comparing branching and branchless implementations

We can clearly see that the number of branches in the branchless version is (obviously) a lot fewer than that of the branched code. How the code works is that the condition `i > 0` is implicitly converted from a boolean to an integer, which is well-defined to give either 1 or 0 (true or false), which we then perform arithmetic with.

5.4.1 Beating the branch predictor

In this case, we used a uniform distribution of positive and negative integers; if we instead used one that is skewed towards a certain choice so that our branches are more predictable, then the performance margin disappears:

```cpp
constexpr int N = 1000000;
constexpr int M = std::numeric_limits<int>::max();
std::mt19937_64 rng{420};
std::uniform_int_distribution d{-M, 10};
```

```
$ hyperfine --warmup 1 ./branch2.out ./branchless2.out
Benchmark 1: ./branch2.out
  Time (mean ± σ):      74.8 ms ±   0.7 ms  [User: 69.7 ms, System: 2.6 ms]
  Range (min … max):    74.0 ms …  77.4 ms  37 runs

Benchmark 2: ./branchless2.out
  Time (mean ± σ):      75.4 ms ±   0.4 ms  [User: 70.4 ms, System: 2.5 ms]
  Range (min … max):    74.6 ms …  76.2 ms  37 runs

Summary
  './branch2.out' ran
    1.01 ± 0.01 times faster than './branchless2.out'
```

Snippet 30: With predictable data, we can't beat the branch predictor

5.4.2 Conditional moves

We won't cover this too much, but x86 CPUs have instructions that perform operations conditionally — `set` and `cmov`. The former sets the destination to 0 or 1 depending on whether the condition (eg. `setge` checks if the status flags correspond to `>=`) is true, while the latter performs a move conditionally based on the condition (eg. `cmovge` only moves if `>=`).

While this might initially seem superior to branches, it's not always the case because of speculative execution in modern processors. In a tight loop, the CPU executes further iterations of the loop speculatively by predicting the branch, but it cannot do a prediction for a conditional move.

For a more in-depth explanation, watch Chandler's CppCon talk — from 30 minutes onwards.

6 Optimising C++ programs

Now we'll give a more high-level approach for how to optimise C++ programs without going into micro-optimisations like some of the ones that we went through above.

6.1 Reducing indirections

The first tip is to reduce indirections whenever possible; one example of this is passing arguments by value if they are small enough, instead of using a const reference. We can write a simple benchmark here:

```cpp
[[gnu::noinline]] size_t take_value(std::string_view sv) {
  return sv.size() * 2;
}

[[gnu::noinline]] size_t take_ref(const std::string_view& sv) {
  return sv.size() * 2;
}

[[gnu::noinline]] size_t take_value(size_t n) {
  return n / 2;
}

[[gnu::noinline]] size_t take_ref(const size_t& n) {
  return n / 2;
}
```

```cpp
for (auto _ : st) {
  for (auto i : idx) {
    n += take_value(  //
        take_value(strings[i]));
  }
}
```

```cpp
for (auto _ : st) {
  for (auto i : idx) {
    n += take_ref(  //
        take_ref(strings[i]));
  }
}
```

Snippet 31: Comparing taking by reference vs value

Running it, we confirm our hypothesis that passing small types by value is indeed faster, by a measurable amount:

```
$ ./pass.out 2> /dev/null
--------------------------------------------------------------
Benchmark                    Time             CPU   Iterations
--------------------------------------------------------------
bench_pass_by_value       3151 ns         3136 ns       223317
bench_pass_by_ref         3767 ns         3718 ns       190583
```

Snippet 32: Passing by value is a little faster

If we look at the generated assembly, we can easily see why:

```
take_ref(const size_t&):
   mov    (%rdi),%rax
   shr    %rax
   ret

take_ref(const std::string_view&):
   mov    0x8(%rdi),%rax
   add    %rax,%rax
   ret
```

```
take_value(size_t):
   mov    %rdi,%rax
   shr    %rax
   ret

take_value(std::string_view):
   lea    (%rsi,%rsi,1),%rax
   ret
```

Snippet 33: The generated assembly for our 4 functions

For the pass-by-reference implementations, we see an indirect load — `(%rdi)`, which dereferences the pointer stored in `rdi` — in both functions (the first one also adds an offset of 8, to get the `size` field directly).

For the pass-by-value implementations though, we don't see such a thing — for the `size_t` function, we see that it just moves `rdi` to `rax` and shifts it right (divide by 2, a strength reduction). For the `string_view` one, it uses the `lea` instruction to essentially perform `rsi + rsi` and returns it.

From that, we can deduce that the size of the `string_view` is just passed directly in `rsi`! If we perform more deduction, we can see that `string_view`'s layout is probably something like this:

```cpp
struct string_view {
  const char* buf;
  size_t length;
};
```

Snippet 34: The likely layout of `std::string_view`

That is, we could pass the struct entirely in registers[5], saving an extra indirection!

6.2 Reducing dynamic allocations

The next thing to talk about is reducing dynamic allocations; the heap is a complex beast, and while it is very well optimised on modern platforms, it's still slower than just decrementing the stack pointer to make some stack space.

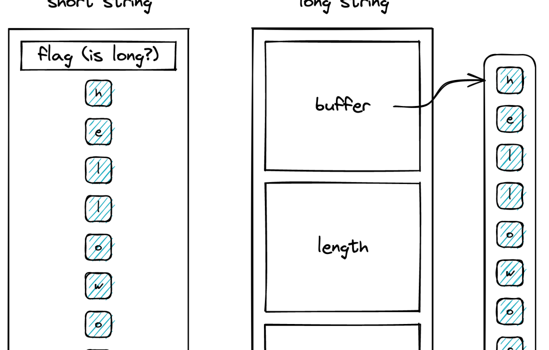
A key point to note is that heap memory is not inherently any faster or slower than stack memory — at the end of the day, they are just memory regions, and will be cached and evicted in similar ways. The difference lies in the *work required* to get an allocation (usable piece of memory) from one of those regions.

We'll talk about a few ways here.

6.3 Small string optimisation (SSO)

Most production C++ standard libraries implement what's known as the *small string optimisation*, which does what it says — it optimises for small strings. It does this by storing the contents of the string inside the string struct itself, instead of through a pointer in the heap.

How does this work? Well, you can think of a `std::string` just as a container of `char`, so it needs to have a pointer to a buffer, a size, and a capacity — on 64 bit platforms, that's already 24 bytes! There are many workloads where strings are often smaller than 24 bytes, so SSO can be very helpful.



The layout of a string with short-string-optimisation compared to one without (long string)

6.3.1 Evaluating SSO performance

To see the impact of SSO, let's bring back our `SimpleString` class from earlier, which *doesn't* implement the short string optimisation. We won't cover the code again, so let's go straight to the benchmarks:

```cpp
// bench_simple_short
for (auto _ : st) {
  SimpleString str{"hello, world!"};
  benchmark::DoNotOptimize(str);
}
```

```cpp
// bench_sso_short
for (auto _ : st) {
  std::string str{"hello, world!"};
  benchmark::DoNotOptimize(str);
}
```

```
$ ./sso.out
-------------------------------------------------------------
Benchmark                   Time             CPU   Iterations
-------------------------------------------------------------
bench_simple_short       79.9 ns         79.9 ns      8646031
bench_sso_short          1.00 ns         1.00 ns    693976286
```

Snippet 35: Comparing implementations with and without short string optimisation

As we might expect, the implementation with SSO (`std::string`) is *a lot* faster than our SimpleString. We're not performing any reallocations or appends here, so this is pretty much on SSO. If we look at an example that uses a large string (that is too big to fit in the SSO buffer), we can see we get comparable performance:

```cpp
// bench_simple_long
for (auto _ : st) {
  SimpleString str{
      "hello, world! this is quite "
      "a long string, so please "
      "forgive me."};
  benchmark::DoNotOptimize(str);
}
```

```cpp
// bench_sso_long
for (auto _ : st) {
  std::string str{
      "hello, world! this is quite "
      "a long string, so please "
      "forgive me."};
  benchmark::DoNotOptimize(str);
}
```

```
$ ./sso.out
-------------------------------------------------------------
Benchmark                   Time             CPU   Iterations
-------------------------------------------------------------
bench_simple_long        79.2 ns         79.2 ns      8935182
bench_sso_long           79.6 ns         79.5 ns      8841175
```

Snippet 36: Without SSO, the two implementations performs similarly

6.3.2 Short-string optimisation implementation

Reference: Joel Laity - libc++'s implementation of std::string

There are a few ways to implement SSO, but we'll look at libc++'s implementation here. You can think of its layout as something like this:

```cpp
class string {
  struct short_t {
    uint8_t size;
    char buffer[23];
  };

  struct long_t {
    size_t capacity;
    size_t length;
    char* buffer;
  };

  union {
    long_t __long_str;
    short_t __short_str;
  };
};
```

```cpp
char* __get_ptr() {
  return (__short_str.size & 1)  //
             ? __short_str.buffer
             : __long_str.buffer;
}

size_t __get_size() const {
  return (__short_str.size & 1)  //
             ? __short_str.size >> 1
             : __long_str.length;
}

template <size_t N>
string(const char (&s)[N]) {
  if (N <= 22) {
    __short_str.size = N << 1;
    std::copy(s, s + N - 1, __short_str.buffer);
    __short_str.buffer[N] = 0;
  } else {
    // normal long string stuff
  }
}
```

Snippet 37: Possible implementations of SSO

The long string is exactly as we expect. In the short string, the least significant bit of `size` is used as a flag for whether it's a short or long string. We can use this bit because (as the library writers!) we know that `new` (or whatever) will return even addresses.

When we call the constructor with a sufficiently short string, we can use the short version, and it works as one might expect (taking care to still null-terminate it). The other helper methods of string aren't too interesting, so we'll skip those.

There are quite a few possible implementations — libstdc++'s `std::string` is 32 bytes large, and it does SSO by storing a pointer to itself, and the Folly library's `fbstring` can store a 23 byte string (excluding the null terminator) by being a little more clever.

6.4 String interning

Reference: SerenityOS AK library FlyString: header, source

One last string-related optimisation that we'll talk about is string interning, which deduplicates strings by storing them in a global data structure, and passing around handles to the unique, deduplicated string.

This is very useful in compilers, where we might have a small set of identifiers (`hash`, `a`, `copy`, …) that get reused *all the time*, and where the main operation performed is equality (`operator==`).

We can implement something like this by looking up a map of existing strings before allocating a copy of one, and in the case of copy construction and assignment, we can cheaply copy the handle to the global string without doing any allocations.

This allows us to use the handles themselves to check if any two strings are equal. Since we always consult this global map first, we never create two different handles to the same string.

Let's see how this might be implemented.

```cpp
struct InternString {
  // Handle consists of a pointer to the string in the string_pool
  const std::string* ptr;

private:
  // Global pool of all unique strings
  static inline std::unordered_set<std::string> string_pool;

  // ...

};
```

Snippet 38: Struct layout of `InternString`

To create new `InternString`s, we simply have to insert it into the global pool, and then return the pointer to the string in that global pool.

If the string already exists in the global pool, `std::unordered_set` will already deduplicate it for us, and not allocate a new `std::string`. Instead, it'll simply return an iterator to the existing one.

```cpp
  InternString(std::string&& str) {
    auto res = string_pool.insert(std::move(str));
    ptr = &*res.first;
  }

  InternString(const std::string& str) {
    auto res = string_pool.emplace(str);
    ptr = &*res.first;
  }

  InternString(const char* str) : InternString(std::string(str)) {}
```

Snippet 39: Constructors of `InternString`

It's now very easy to create the comparison functions that we needed.

```cpp
  friend bool operator==(const InternString& lhs,
                         const InternString& rhs) {
    return lhs.ptr == rhs.ptr;
  }
  friend auto operator<=>(const InternString& lhs,
                          const InternString& rhs) {
    return *lhs.ptr <=> *rhs.ptr;
  }
```

Snippet 40: Comparison operators of `InternString`

We might as well throw in a hashing function, so that we can create `std::unordered_map`s of `InternString`, for example.

```cpp
template <>
struct std::hash<InternString> {
  std::size_t operator()(InternString s) {
    return reinterpret_cast<std::size_t>(s.ptr);
  }
};
```

Snippet 41: `std::hash` specialization for `InternString`

To benchmark it, we'll create a `std::vector` of 1000000 `InternString`s or `std::string`s, each string being one of "0", "1", … "99". After that initial setup phase, we'll see how long it takes to count the number of "0"s, to copy the vector, and to sum the hashes of every string.

```
$ ./intern.out --benchmark_filter='bench'
-------------------------------------------------------------
Benchmark                   Time             CPU   Iterations
-------------------------------------------------------------
bench_intern_count     952879 ns       950769 ns          769
bench_string_count    4465814 ns      4457378 ns          157
bench_intern_copy     1330165 ns      1325640 ns          561
bench_string_copy    14802987 ns     14772064 ns           47
bench_intern_hash      997506 ns       995541 ns          713
bench_string_hash     9775822 ns      9760627 ns           72
```

As expected, `InternString` beats `std::string` on every benchmark :P

However, be aware that `InternString` is not applicable for all use cases. For example, if you create lots of distinct strings only once, or you're doing lots of string manipulation, then it's often better to just use `std::string`. We can see this by measuring how long it takes to generate the vector of strings we used in the benchmark.

```
$ ./intern.out --benchmark_filter='generate'
--------------------------------------------------------------
Benchmark                    Time             CPU   Iterations
--------------------------------------------------------------
generate_intern_vec   65877576 ns     65781658 ns           10
generate_string_vec   29067194 ns     29017604 ns           24
```

6.5 Arena allocators

In the previous lecture we've already seen how to use C++ allocators, specifically the polymorphic allocators, to build monotonic allocators. More generally speaking, these are a form of "arena" allocator.

Essentially, the point of an arena allocator is to make allocations very very cheap, at the cost of not being able to deallocate individual blocks at any time — you usually can only deallocate the whole arena at once.

6.5.1 Implementing a simple arena allocator

Instead of using the PMR library again, we'll just implement our own (slightly janky) version of an arena allocator. One thing to note is that our allocator must respect alignment requirements, if not we will run into undefined behaviour.

```cpp
struct Arena {
  Arena(size_t capacity) : m_buffer(new char[capacity]), m_used(0) {}
  ~Arena() {
    delete[] m_buffer;
  }

  Arena(const Arena&) = delete;
  Arena& operator=(const Arena&) = delete;

  template <typename T, typename... Args>
  T* create(Args&&... args) {
    // ...?
  }

private:
  char* m_buffer;
  size_t m_used;
};
```

Snippet 42: The skeleton of our Arena

The boilerplate is quite simple: we just specify a capacity for the arena, and it allocates that amount of memory at one go, and deletes it on destruction. We also make it non-copyable for obvious reasons.

Of course, we've left out the real meat of the arena :D. Let's look at its implementation now:

```cpp
template <typename T, typename... Args>
T* create(Args&&... args) {
  // ...?
  constexpr size_t align = alignof(T);
  constexpr uintptr_t mask = static_cast<uintptr_t>(~(align - 1));

  // calculate where the usable space starts
  auto current = m_buffer + m_used;

  // get the next aligned pointer starting from `current`
  auto ptr = reinterpret_cast<char*>(
      (reinterpret_cast<uintptr_t>(current + (align - 1))) & mask);

  // calculate the real size (padding for alignment + actual size of T)
  auto real_size = static_cast<size_t>((ptr + sizeof(T)) - current);
  m_used += real_size;

  // construct the type
  return new (ptr) T(std::forward<Args>(args)...);
}
```

Snippet 43: The implementation of `Arena::create`

Let's break it down step by step:

1. get the alignment of our `T` type
2. figure out where we can start allocating space, after the previous allocations
3. do some pointer math to align the pointer to the next alignment boundary (of `align`)
4. get the size we need to increment by (the `real_size`)
5. use placement new and `std::forward` to construct the new object in-place our buffer

It's quite simple, really. Let's benchmark the performance versus just `new`:

```cpp
for (auto _ : st) {
  constexpr size_t N = 10'000;
  std::vector<Big*> bigs{};
  bigs.reserve(N);

  for (size_t i = 0; i < N; i++)
    bigs.push_back(new Big());
}
```

```cpp
for (auto _ : st) {
  constexpr size_t N = 10'000;
  auto arena = Arena(1 << 22);
  std::vector<Big*> bigs{};
  bigs.reserve(N);

  for (size_t i = 0; i < N; i++)
    bigs.push_back(arena.create<Big>());
}
```

```
$ ./arena.out
-----------------------------------------------------
Benchmark            Time             CPU   Iterations
-----------------------------------------------------
bench_new      1010849 ns      1005330 ns          706
bench_arena      60247 ns        60113 ns        11536
```

Snippet 44: Benchmarking `new` vs our arena

We get a sizable performance improvement here, so that's a win.

6.5.2 Arena considerations

Unfortunately, arena allocators are not suitable for all kinds of programs; where they perform best is when there are a lot of small objects that are created, and then at some point, they can *all* be deallocated. An example of this is in the parser for a compiler; once the parsing is finished (and converted to an intermediate representation, perhaps), the parse tree and all its nodes can just be deleted all at once.

Another point to note is that we never call the destructor for the objects, so their lifetime never ends! While this is not *technically* undefined behaviour, it does mean that types that have non-trivial destructors might not work as expected.

This limitation is due to the simplicity of the arena — we have no additional bookkeeping mechanism to track where each allocation starts and ends, and we don't even know the type of the objects once they're gone. One way around this is to use a *pool allocator* instead, and make it templated; now, each block is exactly the same, and we know the type.

6.6 Struct layout (aka "data oriented design")

We talked about accessing 2D arrays in row-major and column-major order above; for a less academic example, suppose we are writing some kind of game that has some kind of entity system:

```cpp
// bench_AoS
struct Enemy {
  int age;
  double health;
  double damage;
};

std::array<Enemy, 256> enemies{};
```

Snippet 45: The array-of-struct implementation

While `age` seems like a weird contrived example, you can think of it as a position, or something else that needs to be touched basically all the time. Every game tick, we update the ages of all the enemies — seems trivial enough.

This implementation is usually called the "Array of Struct" version, because we have an array… of structs. Let's look at the "transposed" version, which is "Struct of Array":

```cpp
// bench_SoA
struct Enemies {
  std::array<int, 256> ages;
  std::array<double, 256> healths;
  std::array<double, 256> damages;
};

Enemies enemies{};
```

Snippet 46: The struct-of-array implementation

Here, we make one struct for all the enemies, and have separate arrays for each of the fields of the enemy. The updating code is quite similar in both cases:

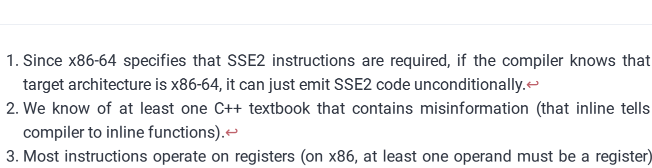
```cpp
for (auto _ : st) {
  for (int t = 0; t < 1000; t++) {
    for (auto& enemy : enemies) {
      enemy.age++;
    }
  }
}
```

```cpp
for (auto _ : st) {
  for (int t = 0; t < 1000; t++) {
    for (auto& age : enemies.ages) {
      age++;
    }
  }
}
```

However, if we benchmark them, we can see the stark contrast in performance:

```
$ ./entity.out
-----------------------------------------------------
Benchmark           Time             CPU   Iterations
-----------------------------------------------------
bench_aos       84158 ns        83975 ns         8296
bench_soa       22086 ns        22038 ns        31708
```

Of course, this should also *not* be a surprise, because this is really just row-major and column-major access in disguise!



Snippet 47: Everything is a matrix

---

1. Since x86-64 specifies that SSE2 instructions are required, if the compiler knows that the target architecture is x86-64, it can just emit SSE2 code unconditionally.↩
2. We know of at least one C++ textbook that contains misinformation (that inline tells the compiler to inline functions).↩
3. Most instructions operate on registers (on x86, at least one operand must be a register), so the compiler needs to perform *register allocation* to allocate variables to registers. By inlining a function, the number of variables increases, so allocation becomes more complex.↩
4. Unless you're performing non-temporal memory accesses.↩
5. This is limited by ABI; if you're using an inferior platform like Windows, its ABI limitations prevent passing structs by value like this.↩