What is faster: vec.emplace_back(x) or vec[x] ?

October 24, 2022C++ Performance, PerformanceLeave a Reply

*We at **Johnny's Software Lab LLC** are experts in performance. If performance is in any way concern in your software project, feel free to contact us.*

When we need to fill `std::vector` with values and the size of vector is known in advance, there are two possibilities: using `emplace_back()` or using `operator[]`. For the `emplace_back()` we should reserve the necessary amount of space with `reserve()` before emplacing into vector. This will avoid unnecessary vector regrow and benefit performance. Alternatively, if we want to use `operator[]`, we need to initialize all the elements of the vector before using it.

A common wisdom says that `emplace_back()` version should be faster. When using `emplace_back()`, we are accessing the elements of the vector only once, when inserting values into vector. On the other hand, when using `operator[]`, the CPU will need the run through the vector two times: first time when the vector is constructed, and the second time when we are actually filling the vector.

The Experiment

To test the behavior of these two approaches, we conduct an experiment. In the experiment, we are limiting ourselves to a vector of doubles with 256M doubles and 2 GB size. We have a following really simple code for the `emplace_back()` version:

```
std::vector<double> result;
result.reserve(in.size());
size_t size = in.size();
for (size_t i = 0; i < size; i++) {
    result.emplace_back(std::sqrt(in[i]));
}
```

And here is the corresponding code for the `operator[]` version:

```
std::vector<double> result(in.size());
size_t size = in.size();
for (size_t i = 0; i < size; i++) {
    result[i] = std::sqrt(in[i]);
}
```

We are measuring the runtime of the hot loop, as well as the total runtime (including vector initialization). Let's see if our assumptions about the runtime verify in tests.

First Results

Here are the first results[1]:

	emplace_back()	**operator[]**
Total	Runtime: 0.915 s	Runtime: 0.815 s
Hot loop	Runtime: 0.915 s	Runtime: 0.276 s

Unexpectedly, the total runtime for the `operator[]` version is about 10% faster. But the numbers look weird. There are a few interesting questions about these numbers: (1) why is the total runtime for the `operator[]` version faster? (2) why is the hot loop runtime for the `operator[]` 3.2 times faster than `emplace_back()` hot loop and (3) in `operator[]` why is the initialization so much slower than the hot loop (0.539 s vs 0.276 s)?

When your debugging effort comes to a standstill and you do not know what to measure, a good approach is to measure everything, then compare the results to try to figure out what's going on. It is also useful to start to answer the simplest question first and then build on top of that.

Question 1: Why is vector initialization so slow for operator[] ?

The `operator[]` hot loop runtime is 0.276 s, whereas the vector initialization runtime is 0.539 s. What this means is that initializing the vector was almost two times slower than the hot loop, but this doesn't make sense. Vector initialization is a simple `memset`, the simplest of all. In the hot loop we have `sqrt` which is not a cheap operation. So a natural expectation would be for the runtimes to be reversed – initialization faster and `sqrt` slower. To understand what is going on, we need to investigate further.

Let's look at the instruction count and the CPI (cycles-per-instruction) metric for emplace back:

	Instructions	**CPI**
Total	336.08 M	5.124
Hot Loop	335.55 M	2.718

These numbers don't make even more sense. The vector initialization was executing 99% of total instructions, yet it is faster than the 1 % of instructions executed for the vector initialization.

When things like this happen, we are essentially stuck. So, we measure everything. To do this, we are using LIKWID, a tool that can collect hardware counters. In addition, we are using LIKWID wrapper that can measure things like user mode runtime, system mode runtime, pagefaults, etc.

After the measurements, the most noticeable difference between the two codes is the time spent in system mode and the number of page faults. Here are the numbers:

	emplace_back()	**operator[]**
Total	User: 0.542 s System: 0.371 s Pagefaults: 524289	User: 0.366 s System: 0.454 s Pagefaults: 524289
Hot loop	User: 0.542 s System: 0.371 s Pagefaults: 524288	User: 0.278 s System: 0.000 s Pagefaults: 0

These numbers are very revealing. First access to the uninitialized memory allocated by the vector will result in many minor pagefaults. Minor pagefaults happen when a page in virtual memory is not yet backed up by a page in physical memory. When a minor pagefault happens, the kernel takes over to perform the allocation. Switching to kernel and allocating memory is relatively expensive process.

For the `emplace_back()` version, physical memory allocations happen inside the hot loop, because our test program is accessing memory for the first time inside the hot loop. For this reason we see many pagefaults and large amount of time spent in the system mode. **For the `operator[]` version, the physical memory allocation happens when the vector is constructed for the first time**. Therefore, the vector initialization takes so much time. In the hot loop there is no physical memory allocation for the `operator[]` version; that's why its hot loop is so much faster compared to `emplace_back()` hot loop.

We still don't understand why the hot loop in `emplace_back()` is slower. Is it only because of the page faults, or something else.

Question 2: Why is the hot loop in emplace_back() slower? Pagefaults or something else?

To discover this, we need to repeat the same test, but this time `emplace_back()` needs to work with virtual memory that has already been allocated in physical memory. To achieve this, we need to overwrite the memory allocator for `std::vector`.

We have created `malloc_wrapper` and passed it to `std::vector`. When memory is allocated through `malloc_wrapper`, it is also initialized to some default value. By doing this, pagefaults move from `emplace_back()` to `reserve()` method.

Here are the runtimes, total instruction count and CPI for the two hot loops:

	operator[]	emplace_back() with preinitialized memory
Runtime	0.276 s	0.496 s
Instructions	335.548 M	2952.794 M
CPI	2.72	0.606
Pagefaults	0	0
Systime	0 s	0 s

The `emplace_back()` version is slower about two times, but what surprises is the instruction count. The `operator[]` version executes almost ten times less instructions, albeit with much lower efficiency (CPI 2.72 vs 0.606).

We do expect that `emplace_back()` executes more instruction, because it needs to check for array boundaries in each iteration and regrow the array if necessary. But this doesn't account for 10 times more instructions.

The reason for this large difference in instruction count is typically vectorization: the compiler can emit special vector instructions, which process four doubles in a single instruction. In addition, the `operator[]` hot loop is without conditionals. We can verify this in two ways: (1) compiler optimization report or (2) looking at disassembly.

We are using CLANG for this experiment, and passing `-Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize` will provide us with the vectorization report. **The results are clear: hot loop in `operator[]` is vectorized; in contract hot loop in `emplace_back()` isn't**.

Question 3: Why is only the operator[] hot loop vectorized?

The `operator[]` hot loop is vectorized. The `emplace_back()` hot loop isn't. The reason, according to the CLANG vectorization report is `loop not vectorized: could not determine number of loop iterations`.

The vectorization report indicates that the number of iterations cannot be calculated before the loop starts, but this is not the case. The loop has exactly 256M iterations, so the report is misleading. The real reason is how `emplace_back()` is implemented!

Inside `emplace_back()`, two things happen. The code first checks if the size of vector is big enough to hold the next value to insert. If it is not, it makes another call to the system allocator to request a larger buffer, copies the values from the old buffer to the new buffer and destroys the old buffer. After this, the vector is large enough to hold the new value.

When we make a call to `reserve()`, we know for sure there will be no call to the system allocator, data copying and destruction when `emplace_back()` is called. But the compiler doesn't know this. So, the compiler must emit a check if the vector is large enough, and grow the vector if it isn't. **But just having a conditional call to a function, even if the call is never made, is a vectorization inhibitor. The compiler must emit the slower scalar version for calls to `push_back()` and `emplace_back()`, even if we know for sure that there will always be enough place in the vector.**

Question 4: What is the effect of pagefaults on runtime for the two hot loops?

To understand the effect of pagefaults on runtime, we need to compare both hot loops with and without pagefaults. For the `emplace_back()` version, we already have two implementations, one with page faults, one without. For the `operator[]` version, we only have implementation without pagefaults. We can emulate the implementation with pagefaults by using C style array instead of vector. Internally, vectors use C style arrays, so even if the code is not the same, the compiler emits very similar assembly.

Here is the relevant data:

	Without pagefaults	With pagefaults
Hot loop in `emplace_back()`	Runtime: 0.496 s Instructions: 2952.794 M Pagefaults: 0	Runtime: 0.920 s Instructions: 3221.755 M Pagefaults: 524288
Hot loop in `operator[]`	Runtime: 0.276 s Instructions: 335.55 M Pagefaults: 0	Runtime: 0.714 s Instructions: 336.07 M Pagefaults: 524288

If we look at these numbers, we see that **the slowdown due to pagefaults is constant**, 0.424 s for the `emplace_back()` and 0.438 s for the `operator[]`. This is the time needed to resolve 524288 pagefaults, which when multiplied by page size of 4 kB, results in exactly 2 GB of data. The time needed per pagefault is about 0.8 µs.

The slowdown due to pagefaults is more pronounced on the vectorized version of the code, since it is faster and more efficient.

Question 5: If the `operator[]` hot loop is not vectorized, which version is faster then?

Originally, we established that the `operator[]` version is slightly faster, and this is because its hot loop is vectorized. But what happens if compiler omits vectorization. Which version is faster than? Here are the numbers:

	emplace_back()	**operator[] without vectorization**
Total	Runtime: 0.920 s Instructions: 3221.76 M Pagefaults: 524289	Runtime: 1.031 s Instructions: 1611.15 M Pagefaults: 524289
Hot loop	Runtime: 0.920 s Instructions: 3221.75 M Pagefaults: 524288	Runtime: 0.491 s Instructions: 1610.61 M Pagefaults: 0

Without vectorization, the `emplace_back()` version is faster, even though it executes almost two times more instructions.

Discussion

We picked this example intentionally to show how deep the rabbit hole goes. What apparently seems to be a very trivial question, “which version is faster” translates into a much more complex question involving kernel space, compiler optimizations and hardware efficiency.

A question you might ask is what happens if the vector holds other types or it is running on other architectures. In principle, if the type stored in the vector is a simple type (not a class or a struct), the loop is often vectorizable and one could expect to see similar numbers as we have seen here. With larger classes, it is the opposite: vectorization doesn’t pay off because of the low memory efficiency, so the `operator[]` version will in principle slower. But, as always with software performance, the runtime numbers are the ones that give the final verdict.

If you are interested in this sort of performance optimizations, I deeply recommend to read about the essence of vectorization here, and about page faults here.