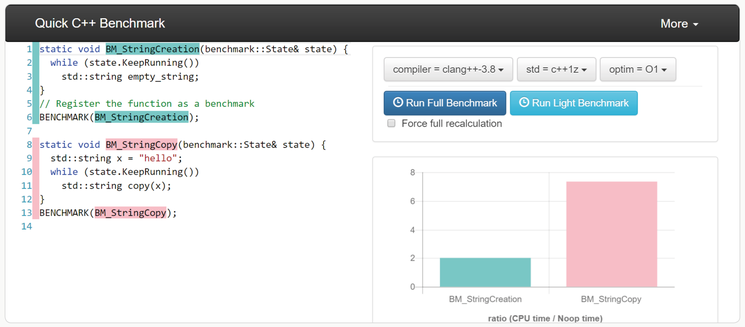
Simple C++ Microbenchmarks with quick-bench.com

What is the fastest between solution A and B? By which order of magnitude? What does “this operation is slow” really mean compared to the rest of the function? These are questions many C++ programmers ask themselves, but rarely actually measure.

Quick Bench is a free online tool to launch micro benchmarks on a whim and compare performances.

FREDERIC TINGAUD @FredTingaudDev

For a very sensitive piece of code, we will deploy benchmarks and non-regression tests. But in most cases, measuring would take more time than it is worth, so if we want to have programmers make educated choices instead of counting on cargo cult informations, we need a tool that is immediately available and reduces the boilerplate code as much as possible.

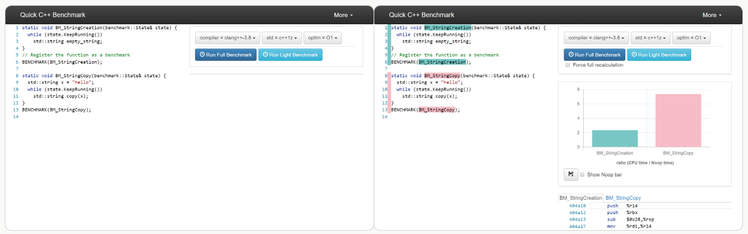


Usage

Using Quick Bench is very straight-forward. You write a benchmark using Google Benchmark API, or edit the one already there in the left-side editor. Then, you press “Run Full Benchmark”.

The results are visible in a chart and the disassembly of the generated functions is displayed under it.

It is also possible to set some compiler options using the compiler combo-boxes and to run the benchmark without disassembly probing by pressing “Run Light Benchmark”



Behind the Scenes

Quick Bench back-end runs on AWS, behind a load-balancer. The code is compiled and executed inside mono-threaded Docker containers on virtual machines.

Thus, it should exclusively be used to compare functions among the same benchmark, not to get absolute run times and hardware bound benchmarks.

This is why Quick Bench doesn’t display actual times, but ratios, so that users don’t get misled into thinking a benchmark would take the same time on their machine.

The annotated disassembly is obtained using the **`perf`** tool from the linux kernel.

Google Benchmark

Google Benchmark is a micro-benchmarking library that has the advantage of having little boilerplate, while being complete. To create a simple non parameterized benchmark, one simply does as follows:

```
static void StringCreation(benchmark::State& state) {
  while (state.KeepRunning())
    std::string empty_string;
}
// Register the function as a benchmark
BENCHMARK(StringCreation);
```

Variable initialization is done before the **`while(state.KeepRunning())`**. Only the content of this while loop will be repeatedly run and benchmarked.

It is also possible to run a parameterized benchmark by passing arguments to the registering macro:

```
static void CharCopy(benchmark::State& state) {
  char* src = new char[state.range(0)];
  char* dst = new char[state.range(0)];
  memset(src, 'x', state.range(0));
  while (state.KeepRunning())
    memcpy(dst, src, state.range(0));
  delete[] src;
  delete[] dst;
}
BENCHMARK(CharCopy)->Arg(8)->Arg(64)->Arg(512)->Arg(1<<10);
```

The library has many other options, but this is already enough for the vast majority of Quick Bench use cases.

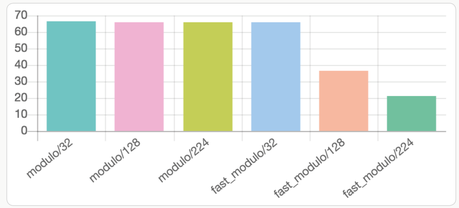
Sample Benchmarks

The following example benchmarks illustrate what kind of information Quick Bench can provide.

Fast modulo

Quick Bench was greatly inspired by a 2015 CppCon presentation by Chandler Carruth about Google Benchmark and perf for micro-benchmarking. In it, Chandler proposed the “fast modulo” algorithm.

The idea is simply to test and immediately return the value, if it is smaller than the modulo, instead of passing it to the modulo operator. The function was thought as a jest but does in fact seriously improve the performances!

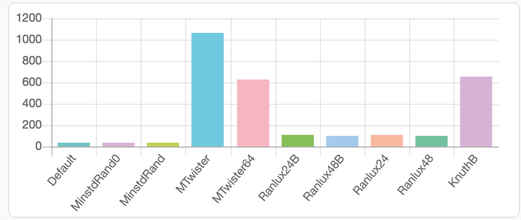


The first three bars are classic modulos of 32, 128 and 224, on values between 0 and 255. The following three are the same with the “fast modulo algorithm”.

Random Engines

C++11 introduced the **`<random>`** header, and with it a variety of random engines, whose quality and cost vary.

The following results show the relative time taken to construct each of the standard random engines.



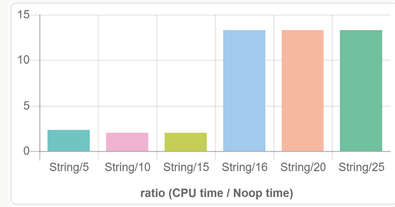
Locale Construction

When manipulating strings, using the correct locale instead of reasonning on plain ASCII is very important. Nevertheless, constructing a new **`std::locale`** is very costly.

The following chart compares creating an **`std::locale`** and using it to call **`std::toupper`** on “Hello World!” vs calling **`toupper`** with an already constructed **`locale`**.

Short String Optimization

Here is the creation time of six strings with different lengths. Guess where the short string optimization kicks in, for this implementation of libc++...



The Project

Quick Bench is an open source project that can be found on github. Please don’t hesitate to contribute and provide feedback including suggestions. It was created as a side project by Frederic Tingaud, who is a principal software engineer at Murex. @WORK_AT_MUREX