Microbenchmarking

Some performance problems and questions can only be resolved by manual measurement. You might be settling a bet about whether it's worthwhile to use a `StringBuilder`, measuring the performance of a third party library, optimizing a delicate algorithm by unrolling inner loops, or helping the JIT place commonly used data into registers by trial and error—and you might not be willing to use a profiler to do the performance measurements for you because the profiler is either too slow, too expensive, or too cumbersome. Though often perilous, microbenchmarking is still very popular. If you do it, we want to make sure that you do it right.

Poor Microbenchmark Example

We start with an example of a poorly designed microbenchmark and improve it until the results it provides are meaningful and correlate well with factual knowledge about the problem domain. The purpose is to determine which is faster—using the `is` keyword and then casting to the desired type, or using the `as` keyword and relying on the result.

```
//Test class
class Employee {
  public void Work() {}
}
//Fragment 1 – casting safely and then checking for null
static void Fragment1(object obj) {
  Employee emp = obj as Employee;
  if (emp ! = null) {
   emp.Work();
  }
}
//Fragment 2 – first checking the type and then casting
static void Fragment2(object obj) {
  if (obj is Employee) {
   Employee emp = obj as Employee;
   emp.Work();
  }
}
```

A rudimentary benchmarking framework might go along the following lines:

```
static void Main() {
  object obj = new Employee();
  Stopwatch sw = Stopwatch.StartNew();
  for (int i = 0; i < 500; i++) {
   Fragment1(obj);
  }
  Console.WriteLine(sw.ElapsedTicks);
  sw = Stopwatch.StartNew();
  for (int i = 0; i < 500; i++) {
   Fragment2(obj);
  }
  Console.WriteLine(sw.ElapsedTicks);
}
```

This is *not* a convincing microbenchmark, although the results are fairly reproducible. More often than not, the output is 4 ticks for the first loop and 200-400 ticks for the second loop. This might lead to the conclusion that the first fragment is 50-100 times faster. However, there are significant errors in this measurement and the conclusion that stems from it:

- The loop runs only once and 500 iterations are not enough to reach any meaningful conclusions—it takes a negligible amount of time to run the whole benchmark, so it can be affected by many environmental factors.
- No effort was made to prevent optimization, so the JIT compiler may have inlined and discarded both measurement loops completely.
- The `Fragment1` and `Fragment2` methods measure not only the cost of the `is` and `as` keywords, but also the cost of a method invocation (to the `Fragment` *N* method itself!). It may be the cast that invoking the method is significantly more expensive than the rest of the work.

Improving upon these problems, the following microbenchmark more closely depicts the actual costs of both operations:

```
class Employee {
  //Prevent the JIT compiler from inlining this method (optimizing it away)
  [MethodImpl(MethodImplOptions.NoInlining)]
  public void Work() {}
}
static void Measure(object obj) {
  const int OUTER_ITERATIONS = 10;
  const int INNER_ITERATIONS = 100000000;
  //The outer loop is repeated many times to make sure we get reliable results
  for (int i = 0; i < OUTER_ITERATIONS; ++i) {
   Stopwatch sw = Stopwatch.StartNew();
   //The inner measurement loop is repeated many times to make sure we are measuring ar
   //operation of significant duration
   for (int j = 0; j < INNER_ITERATIONS; ++j) {
   Employee emp = obj as Employee;
   if (emp ! = null)
   emp.Work();
   }
   Console.WriteLine("As - {0}ms", sw.ElapsedMilliseconds);
  }
  for (int i = 0; i < OUTER_ITERATIONS; ++i) {
   Stopwatch sw = Stopwatch.StartNew();
   for (int j = 0; j < INNER_ITERATIONS; ++j) {
   if (obj is Employee) {
   Employee emp = obj as Employee;
   emp.Work();
   }
   }
   Console.WriteLine("Is Then As - {0}ms", sw.ElapsedMilliseconds);
  }
}
```

The results on one of the author's test machines (after discarding the first iteration) were around 410ms for the first loop and 440ms for the second loop, a reliable and reproducible performance difference, which might have you convinced that indeed, it's more efficient to use just the `as` keyword for casts and checks.

However, the riddles aren't over yet. If we add the `virtual` modifier to the `Work` method, the performance difference disappears completely, not even if we increase the number of iterations. This cannot be explained by the virtues or maladies of our microbenchmarking framework—it is a result from the problem domain. There is no way to understand this behavior without going to the assembly language level and inspecting the loop generated by the JIT compiler in both cases. First, before the `virtual` modifier:

```
; Disassembled loop body - the first loop
mov edx,ebx
mov ecx,163780h (MT: Employee)
call clr!JIT_IsInstanceOfClass (705ecfaa)
test eax,eax
je WRONG_TYPE
mov ecx,eax
call dword ptr ds:[163774h] (Employee.Work(), mdToken: 06000001)
WRONG_TYPE:
; Disassembled loop body - the second loop
mov edx,ebx
mov ecx,163780h (MT: Employee)
call clr!JIT_IsInstanceOfClass (705ecfaa)
test eax,eax
je WRONG_TYPE
mov ecx,ebx
cmp dword ptr [ecx],ecx
call dword ptr ds:[163774h] (Employee.Work(), mdToken: 06000001)
WRONG_TYPE:
```

In Chapter 3 we'll discuss in great depth the instruction sequence emitted by the JIT compiler to call a non-virtual method and to call a virtual method. When calling a non-virtual method, the JIT compiler must emit an instruction that makes sure we are not making a method call on a null reference. The `CMP` instruction in the second loop serves that task. In the first loop, the JIT compiler is smart enough to optimize this check away, because immediately prior to the call, there is a null reference check on the result of the cast (`if (emp ! = null) . . .`). In the second loop, the JIT compiler's optimization heuristics are not sufficient to optimize the check away (although it would have been just as safe), and this extra instruction is responsible for the extra 7-8% of performance overhead.

After adding the `virtual` modifier, however, the JIT compiler generates exactly the same code in both loop bodies:

```
; Disassembled loop body – both cases
mov edx,ebx
mov ecx,1A3794h (MT: Employee)
call clr!JIT_IsInstanceOfClass (6b24cfaa)
test eax,eax
je WRONG_TYPE
mov ecx,eax
mov eax,dword ptr [ecx]
mov eax,dword ptr [eax + 28h]
call dword ptr [eax + 10h]
WRONG_TYPE:
```

The reason is that when invoking a virtual method, there's no need to perform a null reference check explicitly—it is inherent in the method dispatch sequence (as we will see in Chapter 3). When the loop bodies are identical, so are the timing results.

Microbenchmarking Guidelines

For successful microbenchmarks you have to make sure that what you decided to measure adheres to the following guidelines:

- Your test code's environment is representative of the real environment for which it is being developed. For example, you should not run a method on in-memory data collections if it is designed to operate on database tables.
- Your test code's inputs are representative of the real input for which it is being developed. For example, you should not measure how a sorting method fares on three-element lists if it is designed to operate on collections with several million elements.
- The supporting code used to set up the environment should be negligible compared to the actual test code you are measuring. If this is impossible, then the setup should happen once and the test code should be repeated many times.
- The test code should run for sufficiently long so as to be considerable and reliable in the face of hardware and software fluctuations. For example, if you are testing the overhead of a boxing operation on value types, a single boxing operation is likely to be too fast to produce significant results, and will require many iterations of the same test to become substantial.
- The test code should not be optimized away by the language compiler or the JIT compiler. This happens often in Release mode when trying to measure simple operations. (We will return to this point later.)

When you have ascertained that your test code is sufficiently robust and measures the precise effect that you intended to measure, you should invest some time in setting up the benchmarking environment:

- When the benchmark is running, no other processes should be allowed to run on the target system. Networking, file I/O, and other types of external activity should be minimized (for example, by disabling the network card and shutting down unnecessary services).
- Benchmarks that allocate many objects should be wary of garbage collection effects. It's advisable to force a garbage collection before and after significant benchmark iterations to minimize their effects on each other.
- The hardware on the test system should be similar to the hardware to be used in the production environment. For example, benchmarks that involve intensive random disk seeks will run much faster on a solid state hard drive than a mechanical drive with a rotating head. (The same applies to graphics cards, specific processor features such as SIMD instructions, memory architecture, and other hardware traits.)

Finally, you should focus on the measurement itself. These are the things to keep in mind when designing benchmarking code:

- Discard the first measurement result—it is often influenced by the JIT compiler and other application startup costs. Moreover, during the first measurement data and instructions are unlikely to be in the processor's cache. (There are some benchmarks that measure cache effects, and should not heed this advice.)
- Repeat measurements many times and use more than just the average—the standard deviation (which represents the variance of the results) and the fluctuations between consecutive measurements are interesting as well.
- Subtract the measurement loop's overhead from the benchmarked code—this requires measuring the overhead of an empty loop, which isn't trivial because the JIT compiler is likely to optimize away empty loops. (Writing the loop by hand in assembly language is one way of countering this risk.)
- Subtract the time measurement overhead from the timing results, and use the least expensive and most accurate time measurement approach available—this is usually `System.Diagnostics.Stopwatch`.
- Know the resolution, precision, and accuracy of your measurement mechanism—for example, `Environment.TickCount`'s precision is usually only 10-15ms, although its resolution appears to be 1ms.

Note Resolution is the fineness of the measurement mechanism. If it reports results in integer multiples of 100ns, then its resolution is 100ns. However, its precision might be far less—for a physical time interval of 500ns it might report 2 × 100ns on one occasion and 7 × 100ns on another. We might be able to place an upper bound on the precision at 300ns, in this case. Finally, accuracy is the degree of correctness of the measurement mechanism. If it reliably and repeatedly reports a 5000ns physical time interval as 5400ns with a precision of 100ns, we might say its accuracy is +8% of the truth.

The unfortunate example in the beginning of this section should not dissuade you from ever writing your own microbenchmarks. However, you should mind the advice given here and design meaningful benchmarks whose results you can trust. The worst performance optimization is one that is based on incorrect measurements; unfortunately, manual benchmarking often leads into this trap.