Writing Faster Managed Code: Know What Things Cost

249 out of 261 rated this helpful

Jan Gray
Microsoft CLR Performance Team

June 2003

Applies to:
  Microsoft® .NET Framework

Summary: This article presents a low-level cost model for managed code execution time, based upon measured operation times, so that developers may make better informed coding decisions and write faster code. (30 printed pages)

Download the CLR Profiler. (330KB)

Contents

Introduction (and Pledge)

There are myriad ways to implement a computation, and some are far better than others: simpler, cleaner, easier to maintain. Some ways are blazingly fast and some are astonishingly slow.

Don't perpetrate slow and fat code on the world. Don't you despise such code? Code that runs in fits and starts? Code that locks up the UI for seconds at time? Code that pegs the CPU or thrashes the disk?

Don't do it. Instead, stand up and pledge along with me:

> "I promise I will not ship slow code. Speed is a feature I care about. Every day I will pay attention to the performance of my code. I will regularly and methodically *measure* its speed and size. I will learn, build, or buy the tools I need to do this. It's my responsibility."

(Really.) So did you promise? Good for you.

So how *do* you write the fastest, tightest code day in and day out? It is a matter of consciously choosing the frugal way in preference to the extravagant, bloated way, again and again, and a matter of thinking through the consequences. Any given page of code captures dozens of such small decisions.

But you can't make smart choices among alternatives if you don't know what things cost: *you can't write efficient code if you don't know what things cost.*

It was easier in the good old days. Good C programmers knew. Each operator and operation in C, be it assignment, integer or floating-point math, dereference, or function call, mapped more or less one-to-one to a single primitive machine operation. True, sometimes several machine instructions were required to put the right operands in the right registers, and sometimes a single instruction could capture several C operations (famously *dest++ = *src++;), but you could usually write (or read) a line of C code and know where the time was going. For both code and data, the C compiler was WYWIWYG—"what you write is what you get". (The exception was, and is, function calls. If you don't know what the function costs, you don't know diddly.)

In the 1990s, to enjoy the many software engineering and productivity benefits of data abstraction, object-oriented programming, and code reuse, the PC software industry made a transition from C to C++.

C++ is a superset of C, and is "pay as you go"—the new features cost nothing if you don't use them—so C programming expertise, including one's internalized cost model, is directly applicable. If you take some working C code and recompile it for C++, the execution time and space overhead shouldn't change much.

On the other hand, C++ introduces many new language features, including constructors, destructors, new, delete, single, multiple and virtual inheritance, casts, member functions, virtual functions, overloaded operators, pointers to members, object arrays, exception handling, and compositions of same, which incur non-trivial hidden costs. For example, virtual functions cost two extra indirections per call, and add a hidden vtable pointer field to each instance. Or consider that this innocuous-looking code:

```
{ complex a, b, c, d; … a = b + c * d; }
```

compiles into approximately thirteen implicit member function calls *(hopefully inlined)*.

Nine years ago we explored this subject in my article C++: Under the Hood. I wrote:

> "It is important to understand how your programming language is implemented. Such knowledge dispels the fear and wonder of "What on earth is the compiler doing here?"; imparts confidence to use the new features; and provides insight when debugging and learning other language features. It also gives a feel for the relative costs of different coding choices that is necessary to write the most efficient code day to day."

Now we're going to take a similar look at managed code. This article explores the *low-level* time and space costs of managed execution, so we *can* make smarter tradeoffs in our day to day coding.

And keep our promises.

Why Managed Code?

For the vast majority of native code developers, managed code is a better, more productive platform to run their software. It removes whole categories of bugs, such as heap corruptions and array-index-out-of-bound errors that so often lead to frustrating late-night debugging sessions. It supports modern requirements such as safe mobile code (via code access security) and XML Web services, and compared to the aging Win32/COM/ATL/MFC/VB, the .NET Framework is a refreshing clean slate design, where you can get more done with less effort.

For your user community, managed code enables richer, more robust applications—better living through better software.

What Is the Secret to Writing Faster Managed Code?

Just because you can get more done with less effort is not a license to abdicate your responsibility to code wisely. First, you must admit it to yourself: "I'm a newbie." You're a newbie. *I'm a newbie too.* We're all babes in managed code land. We're all still learning the ropes—including what things cost.

When it comes to the rich and convenient .NET Framework, it's like we're kids in the candy store. "Wow, I don't have to do all that tedious strncpy stuff, I can just '+' strings together! Wow, I can load a megabyte of XML in a couple of lines of code! Whoo-hoo!"

It's all so easy. *So easy, indeed.* So easy to burn megabytes of RAM parsing XML infosets just to pull a few elements out of them. In C or C++ it was so painful you'd think twice, maybe you'd build a state machine on some SAX-like API. With the .NET Framework, you just load the whole infoset in one gulp. Maybe you even do it over and over. Then maybe your application doesn't seem so fast anymore. Maybe it has a working set of many megabytes. Maybe you should have thought twice about what those easy methods cost...

Unfortunately, in my opinion, the current .NET Framework documentation does not adequately detail the performance implications of Framework types and methods—it doesn't even specify which methods might create new objects. Performance modeling is not an easy subject to cover or document; but still, the "not knowing" makes it that much harder for us to make informed decisions.

Since we're all newbies here, and since we don't know what anything costs, and since the costs are not clearly documented, what are we to do?

Measure it. The secret is to *measure it* and to *be vigilant*. We're all going to have to get into the habit of measuring the cost of things. If we go to the trouble of measuring what things cost, then we won't be the ones inadvertently calling a whizzy new method that costs ten times what we *assumed* it costs.

(By the way, to gain deeper insight into the performance underpinnings of the BCL (base class library) or the CLR itself, consider taking a look at the Shared Source CLI, a.k.a. Rotor. Rotor code shares a bloodline with the .NET Framework and the CLR. It's not the same code throughout, but even so, I promise you that a thoughtful study of Rotor will give you new insights into the goings on under the hood of the CLR. But be sure to review the SSCLI license first!)

The Knowledge

If you aspire to be a cab driver in London, you first must earn The Knowledge. Students study for many months to memorize the thousands of little streets in London and learn the best routes from place to place. And they go out every day on scooters to scout around and reinforce their book learning.

Similarly, if you want to be a high performance managed code developer, you have to acquire *The Managed Code Knowledge*. You have to learn what each low-level operation costs. You have to learn what features like delegates and code access security cost. You have to learn the costs of the types and methods you're using, and the ones you're writing. And it doesn't hurt to discover which methods may be too costly for your application—and so avoid them.

The Knowledge isn't in any book, alas. You have to get out on *your* scooter and explore—that is, crank up csc, ildasm, the VS.NET debugger, the CLR Profiler, your profiler, some perf timers, and so forth, and see what your code costs in time and space.

Towards a Cost Model for Managed Code

Preliminaries aside, let's consider a cost model for managed code. That way you'll be able to look at a leaf method and tell at a glance which expressions and statements are more costly; and you'll be able to make smarter choices as you write new code.

(This will not address the transitive costs of calling your methods or methods of the .NET Framework. That will have to wait for another article on another day.)

Earlier I stated that most of the C cost model still applies in C++ scenarios. *Similarly, much of the C/C++ cost model still applies to managed code.*

How can that be? You know the CLR execution model. You write your code in one of several languages. You compile it to CIL (Common Intermediate Language) format, packaged into assemblies. You run the main application assembly, and it starts executing the CIL. But isn't that an order of magnitude slower, like the bytecode interpreters of old?

The Just-in-Time compiler

No, it's not. The CLR uses a JIT (just-in-time) compiler to compile each method in CIL into native x86 code and then runs the native code. Although there is a small delay for JIT compilation of each method as it is first called, every method called runs pure native code with no interpretive overhead.

Unlike a traditional off-line C++ compilation process, the time spent in the JIT compiler is a "wall clock time" delay, in each user's face, so the JIT compiler does not have the luxury of exhaustive optimization passes. Even so, the list of optimizations the JIT compiler performs is impressive:

- Constant folding
- Constant and copy propagation
- Common subexpression elimination
- Code motion of loop invariants
- Dead store and dead code elimination
- Register allocation
- Method inlining
- Loop unrolling (small loops with small bodies)

The result is comparable to traditional native code—*at least in the same ballpark*.

As for data, you will use a mix of value types or reference types. Value types, including integral types, floating point types, enums, and structs, typically live on the stack. They are as just as small and fast as locals and structs are in C/C++. As with C/C++, you should probably avoid passing large structs as method arguments or return values, because the copying overhead can be prohibitively expensive.

Reference types and boxed value types live in the heap. They are addressed by object references, which are simply machine pointers just like object pointers in C/C++.

So jitted managed code can be fast. With a few exceptions that we discuss below, if you have a gut feel for the cost of some expression in native C code, you won't go far wrong modeling its cost as equivalent in managed code.

I should also mention NGEN, a tool which "ahead-of-time" compiles the CIL into native code assemblies. While NGEN'ing your assemblies does not currently have a substantial impact (good or bad) on execution time, it can reduce total working set for shared assemblies that are loaded into many AppDomains and processes. (The OS can share one copy of the NGEN'd code across all clients; whereas jitted code is typically not currently shared across AppDomains or processes. But see also LoaderOptimizationAttribute.MultiDomain.)

Automatic Memory Management

Managed code's most significant departure (from native) is automatic memory management. You allocate new objects, but the CLR garbage collector (GC) automatically frees them for you when they become unreachable. GC runs now and again, often imperceptibly, generally stopping your application for just a millisecond or two—occasionally longer.

Several other articles discuss the performance implications of the garbage collector and we won't recapitulate them here. If your application follows the recommendations in these other articles, the overall cost of garbage collection can be insignificant, a few percent of execution time, competitive with or superior to traditional C++ object new and delete. The amortized cost of creating and later automatically reclaiming an object is sufficiently low that you can create many tens of millions of small objects per second.

But object allocation is still not *free*. Objects take up space. Rampant object allocation leads to more frequent garbage collection cycles.

Far worse, unnecessarily retaining references to useless object graphs keeps them alive. We sometimes see modest programs with lamentable 100+ MB working sets, whose authors deny their culpability and instead attribute their poor performance to some mysterious, unidentified (and hence intractable) issue with managed code itself. It's tragic. But then an hour's study with the CLR Profiler and changes to a few lines of code cuts their heap usage by a factor of ten or more. If you're facing a large working set problem, the first step is to look in the mirror.

So do not create objects unnecessarily. Just because automatic memory management dispels the many complexities, hassles, and bugs of object allocation and freeing, because it is so fast and so convenient, we naturally tend to create more and more objects, as if they grow on trees. If you want to write really fast managed code, create objects thoughtfully and appropriately.

This also applies to API design. It is possible to design a type and its methods so they *require* clients to create new objects with wild abandon. Don't do that.

What Things Cost in Managed Code

Now let us consider the time cost of various low-level managed code operations.

Table 1 presents the approximate cost of a variety of low-level managed code operations, in nanoseconds, on a quiescent 1.1 GHz Pentium-III PC running Windows XP and .NET Framework v1.1 ("Everett"), gathered with a set of simple timing loops.

The test driver calls each test method, specifying a number of iterations to perform, automatically scaled to iterate between 2^{18} and 2^{30} iterations, as necessary to perform each test for at least 50 ms. Generally speaking, this is long enough to observe several cycles of generation 0 garbage collection in a test which does intense object allocation. The table shows results averaged over 10 trials, as well as the best (minimum time) trial for each test subject.

Each test loop is unrolled 4 to 64 times as necessary to diminish the test loop overhead. I inspected the native code generated for each test to ensure the JIT compiler was not optimizing the test away—for example, in several cases I modified the test to keep intermediate results live during and after the test loop. Similarly I made changes to preclude common subexpression elimination in several tests.

Table 1 Primitive Times (average and minimum) (ns)

Avg	Min	Primitive	Avg	Min	Primitive	Avg	Min	Primitive
0.0	0.0	Control	2.6	2.6	new valtype L1	0.8	0.8	isinst up 1
1.0	1.0	Int add	4.6	4.6	new valtype L2	0.8	0.8	isinst down 0
1.0	1.0	Int sub	6.4	6.4	new valtype L3	6.3	6.3	isinst down 1
2.7	2.7	Int mul	8.0	8.0	new valtype L4	10.7	10.6	isinst (up 2) down 1
35.9	35.7	Int div	23.0	22.9	new valtype L5	6.4	6.4	isinst down 2
2.1	2.1	Int shift	22.0	20.3	new reftype L1	6.1	6.1	isinst down 3
2.1	2.1	long add	26.1	23.9	new reftype L2	1.0	1.0	get field
2.1	2.1	long sub	30.2	27.5	new reftype L3	1.2	1.2	get prop

34.2	34.1	long mul	34.1	30.8	new reftype L4	1.2	1.2	set field
50.1	50.0	long div	39.1	34.4	new reftype L5	1.2	1.2	set prop
5.1	5.1	long shift	22.3	20.3	new reftype empty ctor L1	0.9	0.9	get this field
1.3	1.3	float add	26.5	23.9	new reftype empty ctor L2	0.9	0.9	get this prop
1.4	1.4	float sub	38.1	34.7	new reftype empty ctor L3	1.2	1.2	set this field
2.0	2.0	float mul	34.7	30.7	new reftype empty ctor L4	1.2	1.2	set this prop
27.7	27.6	float div	38.5	34.3	new reftype empty ctor L5	6.4	6.3	get virtual prop
1.5	1.5	double add	22.9	20.7	new reftype ctor L1	6.4	6.3	set virtual prop
1.5	1.5	double sub	27.8	25.4	new reftype ctor L2	6.4	6.4	write barrier
2.1	2.0	double mul	32.7	29.9	new reftype ctor L3	1.9	1.9	load int array elem
27.7	27.6	double div	37.7	34.1	new reftype ctor L4	1.9	1.9	store int array elem
0.2	0.2	inlined static call	43.2	39.1	new reftype ctor L5	2.5	2.5	load obj array elem
6.1	6.1	static call	28.6	26.7	new reftype ctor no-inl L1	16.0	16.0	store obj array elem
1.1	1.0	inlined instance call	38.9	36.5	new reftype ctor no-inl L2	29.0	21.6	box int
6.8	6.8	instance call	50.6	47.7	new reftype ctor no-inl L3	3.0	3.0	unbox int
0.2	0.2	inlined this inst call	61.8	58.2	new reftype ctor no-inl L4	41.1	40.9	delegate invoke
6.2	6.2	this instance call	72.6	68.5	new reftype ctor no-inl L5	2.7	2.7	sum array 1000
5.4	5.4	virtual call	0.4	0.4	cast up 1	2.8	2.8	sum array 10000
5.4	5.4	this virtual call	0.3	0.3	cast down 0	2.9	2.8	sum array 100000
6.6	6.5	interface call	8.9	8.8	cast down 1	5.6	5.6	sum array 1000000
1.1	1.0	inst itf instance call	9.8	9.7	cast (up 2) down 1	3.5	3.5	sum list 1000
0.2	0.2	this itf instance call	8.9	8.8	cast down 2	6.1	6.1	sum list 10000
5.4	5.4	inst itf virtual call	8.7	8.6	cast down 3	22.0	22.0	sum list 100000
5.4	5.4	this itf virtual call				21.5	21.4	sum list 1000000

A disclaimer: please do not take this data too literally. Time testing is fraught with the peril of unexpected second order effects. A chance happenstance might place the jitted code, or some crucial data, so that it spans cache lines, interferes with something else, or what have you. It's a bit like the Uncertainty Principle: times and time differences of 1 nanosecond or so are at the limits of the observable.

Another disclaimer: this data is only pertinent for small code and data scenarios that fit entirely in cache. If the "hot" parts of your application do not fit in on-chip cache, you may well have a different set of performance challenges. We have much more to say about caches near the end of the paper.

And yet another disclaimer: one of the sublime benefits of shipping your components and applications as assemblies of CIL is that your program can automatically get faster every second, and get faster every year—"faster every second" because the runtime can (in theory) retune the JIT compiled code as your program runs; and "faster ever year" because with each new release of the runtime, better, smarter, faster algorithms can take a fresh stab at optimizing your code. So if a few of these timings seem less than optimal in .NET 1.1, take heart that they should improve in subsequent releases of the product. *It follows that any given code native code sequence reported in this article may change in future releases of the .NET Framework.*

Disclaimers aside, the data does provide a reasonable gut feel for the current performance of various primitives. The numbers makes sense, and they substantiate my assertion that most jitted managed code runs "close to the machine" just like compiled native code does. The primitive integer and floating operations are fast, method calls of various kinds less so, but (trust me) still comparable to native C/C++; and yet we also see that some operations which are usually cheap in native code (casts, array and field stores, function pointers (delegates)) are now more expensive. Why? Let's see.

Arithmetic Operations

Table 2 Arithmetic Operation Times (ns)

Avg	Min	Primitive	Avg	Min	Primitive
1.0	1.0	int add	1.3	1.3	float add
1.0	1.0	int sub	1.4	1.4	float sub
2.7	2.7	int mul	2.0	2.0	float mul
35.9	35.7	int div	27.7	27.6	float div
2.1	2.1	int shift			
2.1	2.1	long add	1.5	1.5	double add
2.1	2.1	long sub	1.5	1.5	double sub
34.2	34.1	long mul	2.1	2.0	double mul
50.1	50.0	long div	27.7	27.6	double div
5.1	5.1	long shift			

In the old days, floating-point math was perhaps an order of magnitude slower than integer math. As Table 2 shows, with modern pipelined floating-point units, it appears there is little or no difference. It is amazing to think an average notebook PC is a now gigaflop class machine (for problems that fit in cache).

Let's take a look at a line of jitted code from the integer and floating point add tests:

Disassembly 1 Int add and float add

```
int add              a = a + b + c + d + e + f + g + h + i;
0000004c 8B 54 24 10      mov        edx,dword ptr [esp+10h]
00000050 03 54 24 14      add        edx,dword ptr [esp+14h]
00000054 03 54 24 18      add        edx,dword ptr [esp+18h]
00000058 03 54 24 1C      add        edx,dword ptr [esp+1Ch]
0000005c 03 54 24 20      add        edx,dword ptr [esp+20h]
00000060 03 D5            add        edx,ebp
00000062 03 D6            add        edx,esi
00000064 03 D3            add        edx,ebx
00000066 03 D7            add        edx,edi
00000068 89 54 24 10      mov        dword ptr [esp+10h],edx

float add            i += a + b + c + d + e + f + g + h;
00000016 D9 05 38 61 3E 00 fld        dword ptr ds:[003E6138h]
0000001c D8 05 3C 61 3E 00 fadd       dword ptr ds:[003E613Ch]
00000022 D8 05 40 61 3E 00 fadd       dword ptr ds:[003E6140h]
```

```
00000028 D8 05 44 61 3E 00 fadd        dword ptr ds:[003E6144h]
0000002e D8 05 48 61 3E 00 fadd        dword ptr ds:[003E6148h]
00000034 D8 05 4C 61 3E 00 fadd        dword ptr ds:[003E614Ch]
0000003a D8 05 50 61 3E 00 fadd        dword ptr ds:[003E6150h]
00000040 D8 05 54 61 3E 00 fadd        dword ptr ds:[003E6154h]
00000046 D8 05 58 61 3E 00 fadd        dword ptr ds:[003E6158h]
0000004c D9 1D 58 61 3E 00 fstp        dword ptr ds:[003E6158h]
```

Here we see the jitted code is close to optimal. In the int add case, the compiler even enregistered five of the local variables. In the float add case, I was obliged to make variables a through h class statics to defeat common subexpression elimination.

Method Calls

In this section we examine the costs and implementations of method calls. The test subject is a class T implementing interface I, with various sorts of methods. See Listing 1.

Listing 1 Method call test methods

```
interface I { void itf1();… void itf5();… }
public class T : I {
   static bool falsePred = false;
   static void dummy(int a, int b, int c, …, int p) { }

   static void inl_s1() { } …    static void s1()     { if (falsePred) dummy(1, 2, 3, …, 16); } …    void inl_i1()       { } …    void i1()
```

Consider Table 3. It *appears*, to a first approximation, a method is either inlined (the abstraction costs nothing) or not (the abstraction costs >5X an integer operation). There does not appear to be a significant difference in the raw cost of a static call, instance call, virtual call, or interface call.

Table 3 Method Call Times (ns)

Avg	Min	Primitive	Callee	Avg	Min	Primitive	Callee
0.2	0.2	inlined static call	inl_s1	5.4	5.4	virtual call	v1
6.1	6.1	static call	s1	5.4	5.4	this virtual call	v1
1.1	1.0	inlined instance call	inl_i1	6.6	6.5	interface call	itf1
6.8	6.8	instance call	i1	1.1	1.0	inst itf instance call	itf1
0.2	0.2	inlined this inst call	inl_i1	0.2	0.2	this itf instance call	itf1
6.2	6.2	this instance call	i1	5.4	5.4	inst itf virtual call	itf5
				5.4	5.4	this itf virtual call	itf5

However, these results are unrepresentative *best cases*, the effect of running tight timing loops millions of times. In these test cases, the virtual and interface method call sites are monomorphic (e.g. per call site, the target method does not change over time), so the combination of caching the virtual method and interface method dispatch mechanisms (the method table and interface map pointers and entries) and spectacularly provident branch prediction enables the processor to do an unrealistically effective job calling through these otherwise difficult-to-predict, data-dependent branches. In practice, a data cache miss on any of the dispatch mechanism data, or a branch misprediction (be it a compulsory capacity miss or a polymorphic call site), can and will slow down virtual and interface calls by dozens of cycles.

Let's take a closer look at each of these method call times.

In the first case, *inlined static call,* we call a series of empty static methods s1_inl() etc. Since the compiler completely inlines away all the calls, we end up timing an empty loop.

In order to measure the approximate cost of a *static method call*, we make the static methods s1() etc. so large that they are unprofitable to inline into the caller.

Observe we even have to use an explicit false predicate variable falsePred. If we wrote

```
static void s1() { if (false) dummy(1, 2, 3, …, 16); }
```

the JIT compiler would eliminate the dead call to dummy and inline the whole (now empty) method body as before. By the way, here some of the 6.1 ns of call time must be attributed to the (false) predicate test and jump within the called static method s1. (By the way, a better way to disable inlining is the CompilerServices.MethodImpl(MethodImplOptions.NoInlining) attribute.)

The same approach was used for the inlined instance call and regular instance call timing. However, since the C# language specification ensures that any call on a null object reference throws a NullReferenceException, every call site must ensure the instance is not null. This is done by dereferencing the instance reference; if it *is* null it will generate a fault that is turned into this exception.

In Disassembly 2 we use a static variable t as the instance, because when we used a local variable

```
  T t = new T();
```

the compiler hoisted the null instance check out of the loop.

Disassembly 2   Instance method call site with null instance "check"

```
            t.i1();
00000012 8B 0D 30 21 A4 05 mov         ecx,dword ptr ds:[05A42130h]
00000018 39 09             cmp         dword ptr [ecx],ecx
0000001a E8 C1 DE FF FF    call        FFFFDEE0
```

The cases of the *inlined this instance call* and *this instance call* are the same, except the instance is this; here the null check has been elided.

Disassembly 3   This instance method call site

```
            this.i1();
00000012 8B CE             mov         ecx,esi
00000014 E8 AF FE FF FF    call        FFFFFEC8
```

Virtual method calls work just like in traditional C++ implementations. The address of each newly introduced virtual method is stored within a new slot in the type's method table. Each derived type's method table conforms with and extends that of its base type, and any virtual method override replaces the base type's virtual method address with the derived type's virtual method address in the corresponding slot in the derived type's method table.

At the call site, a virtual method call incurs two additional loads compared to an instance call, one to fetch the method table address (always found at *(this+0)), and another to fetch the appropriate virtual method address from the method table and call it. See Disassembly 4.

Disassembly 4   Virtual method call site

```
              this.v1();
00000012 8B CE           mov        ecx,esi
00000014 8B 01           mov        eax,dword ptr [ecx] ; fetch method table address
00000016 FF 50 38        call       dword ptr [eax+38h] ; fetch/call method address
```

Finally, we come to *interface method calls* (Disassembly 5). These have no exact equivalent in C++. Any given type may implement any number of interfaces, and each interface logically requires its own method table. To dispatch on an interface method, we lookup the method table, its interface map, the interface's entry in that map, and then call indirect through appropriate entry in the interface's section of the method table.

Disassembly 5   Interface method call site

```
              i.itf1();
00000012 8B 0D 34 21 A4 05 mov        ecx,dword ptr ds:[05A42134h]; instance address
00000018 8B 01             mov        eax,dword ptr [ecx]         ; method table addr
0000001a 8B 40 0C          mov        eax,dword ptr [eax+0Ch]     ; interface map addr
0000001d 8B 40 7C          mov        eax,dword ptr [eax+7Ch]     ; itf method table addr
00000020 FF 10             call       dword ptr [eax]             ; fetch/call meth addr
```

The remainder of the primitive timings, *inst itf instance call*, *this itf instance call*, *inst itf virtual call*, *this itf virtual call* highlight the idea that whenever a derived type's method implements an interface method, it remains callable via an instance method call site.

For example, for the test *this itf instance call*, a call on an interface method implementation via an instance (not interface) reference, the interface method is successfully inlined and the cost goes to 0 ns. Even an interface method implementation is potentially inlineable when you call it as an instance method.

Calls to Methods Yet to be Jitted

For static and instance method calls (but not virtual and interface method calls), the JIT compiler currently generates different method call sequences depending upon whether the target method has already been jitted by the time its call site is being jitted.

If the callee (target method) has not yet been jitted, the compiler emits a call indirect through a pointer which is first initialized with a "prejit stub". The first call upon the target method arrives at the stub, which triggers JIT compilation of the method, generating native code, and updating the pointer to address the new native code.

If the callee has already been jitted, its native code address is known so the compiler emits a direct call to it.

New Object Creation

New object creation consists of two phases: object allocation and object initialization.

For reference types, objects are allocated on the garbage collected heap. For value types, whether stack-resident or embedded within another reference or value type, the value type object is found at some constant offset from the enclosing structure—no allocation required.

For typical small reference type objects, heap allocation is very fast. After each garbage collection, except in the presence of pinned objects, live objects from the generation 0 heap are compacted and promoted to generation 1, and so the memory allocator has a nice large contiguous free memory arena to work with. Most object allocations incur only a pointer increment and bounds check, which is cheaper than the typical C/C++ free list allocator (malloc/operator new). The garbage collector even takes into account your machine's cache size to try to keep the gen 0 objects in the fast sweet spot of the cache/memory hierarchy.

Since the preferred managed code style is to allocate most objects with short lifetimes, and reclaim them quickly, we also include (in the time cost) the amortized cost of the garbage collection of these new objects.

Note the garbage collector spends no time mourning dead objects. If an object is dead, GC doesn't see it, doesn't walk it, doesn't give it a nanosecond's thought. GC is concerned only with the welfare of the living.

(Exception: finalizable dead objects are a special case. GC tracks those, and specially promotes dead finalizable objects to the next generation pending finalization. This is expensive, and in the worst case can transitively promote large dead object graphs. Therefore, don't make objects finalizable unless strictly necessary; and if you must, consider using the *Dispose Pattern*, calling GC.SuppressFinalizer when possible.) Unless required by your Finalize method, don't hold references from your finalizable object to other objects.

Of course, the amortized GC cost of a large short-lived object is greater than the cost of a small short-lived object. Each object allocation brings us that much closer to the next garbage collection cycle; larger objects do so that much sooner that small ones. Sooner (or later), the moment of reckoning will come. GC cycles, particularly generation 0 collections, are very fast, but are not free, even if the vast majority of new objects are dead: to find (mark) the live objects, it is first necessary to pause threads and then walk stacks and other data structures to collect root object references into the heap.

(Perhaps more significantly, fewer larger objects fit in the same amount of cache as smaller objects do. Cache miss effects can easily dominate code path length effects.)

Once space for the object is allocated, it remains to initialize it (construct it). The CLR guarantees that all object references are preinitialized to null, and all primitive scalar types are initialized to 0, 0.0, false, etc. (Therefore it is unnecessary to redundantly do so in your user-defined constructors. Feel free, of course. But be aware that the JIT compiler currently does not necessarily optimize away your redundant stores.)

In addition to zeroing out instance fields, the CLR initializes (reference types only) the object's internal implementation fields: the method table pointer and the object header word, which precedes the method table pointer. Arrays also get a Length field, and object arrays get Length and element type fields.

Then the CLR calls the object's constructor, if any. Each type's constructor, whether user-defined or compiler generated, first calls its base type's constructor, then runs user-defined initialization, if any.

In theory this could be expensive for deep inheritance scenarios. If E extends D extends C extends B extends A (extends System.Object) then initializing an E would always incur five method calls. In practice, things aren't so bad, because the compiler inlines away (into nothingness) calls upon empty base type constructors.

Referring to the first column of Table 4, observe we can create and initialize a struct D with four int fields in about 8 int-add-times. Disassembly 6 is the generated code from three different timing loops, creating A's, C's, and E's. (Within each loop we modify each new instance, which keeps the JIT compiler from optimizing everything away.)

Table 4 Value and Reference Type Object Creation Times (ns)

Avg	Min	Primitive	Avg	Min	Primitive	Avg	Min	Primitive
2.6	2.6	new valtype L1	22.0	20.3	new reftype L1	22.9	20.7	new rt ctor L1
4.6	4.6	new valtype L2	26.1	23.9	new reftype L2	27.8	25.4	new rt ctor L2
6.4	6.4	new valtype L3	30.2	27.5	new reftype L3	32.7	29.9	new rt ctor L3
8.0	8.0	new valtype L4	34.1	30.8	new reftype L4	37.7	34.1	new rt ctor L4
23.0	22.9	new valtype L5	39.1	34.4	new reftype L5	43.2	39.1	new rt ctor L5
			22.3	20.3	new rt empty ctor L1	28.6	26.7	new rt no-inl L1
			26.5	23.9	new rt empty ctor L2	38.9	36.5	new rt no-inl L2
			38.1	34.7	new rt empty ctor L3	50.6	47.7	new rt no-inl L3
			34.7	30.7	new rt empty ctor L4	61.8	58.2	new rt no-inl L4
			38.5	34.3	new rt empty ctor L5	72.6	68.5	new rt no-inl L5

Disassembly 6 Value type object construction

```
            A a1 = new A(); ++a1.a;
00000020 C7 45 FC 00 00 00 00 mov     dword ptr [ebp-4],0
00000027 FF 45 FC         inc         dword ptr [ebp-4]

            C c1 = new C(); ++c1.c;
00000024 8D 7D F4         lea         edi,[ebp-0Ch]
00000027 33 C0            xor         eax,eax
00000029 AB               stos        dword ptr [edi]
0000002a AB               stos        dword ptr [edi]
0000002b AB               stos        dword ptr [edi]
0000002c FF 45 FC         inc         dword ptr [ebp-4]

            E e1 = new E(); ++e1.e;
00000026 8D 7D EC         lea         edi,[ebp-14h]
00000029 33 C0            xor         eax,eax
0000002b 8D 48 05         lea         ecx,[eax+5]
0000002e F3 AB            rep stos    dword ptr [edi]
00000030 FF 45 FC         inc         dword ptr [ebp-4]
```

The next five timings (*new reftype L1, … new reftype L5*) are for five inheritance levels of reference types A, …, E, sans user-defined constructors:

```
    public class A     { int a; }
    public class B : A { int b; }
    public class C : B { int c; }
    public class D : C { int d; }
    public class E : D { int e; }
```

Comparing the reference type times to the value type times, we see that the amortized allocation and freeing cost of each instance is approximately 20 ns (20X int add time) on the test machine. That's fast—allocating, initializing, and reclaiming about 50 million short-lived objects per second, sustained. For objects as small as five fields, allocation and collection accounts for only half of the object creation time. See Disassembly 7.

Disassembly 7   Reference type object construction

```
            new A();
0000000f B9 D0 72 3E 00   mov         ecx,3E72D0h
00000014 E8 9F CC 6C F9   call        F96CCCB8

            new C();
0000000f B9 B0 73 3E 00   mov         ecx,3E73B0h
00000014 E8 A7 CB 6C F9   call        F96CCBC0

            new E();
0000000f B9 90 74 3E 00   mov         ecx,3E7490h
00000014 E8 AF CA 6C F9   call        F96CCAC8
```

The last three sets of five timings present variations on this inherited class construction scenario.

1. *New rt empty ctor L1, …, new rt empty ctor L5:* Each type A, …, E has an empty user-defined constructor. These are all inlined away and the generated code is the same as the above.
2. *New rt ctor L1, …, new rt ctor L5:* Each type A, …, E has a user-defined constructor that sets its instance variable to 1:

```
public class A     { int a; public A() { a = 1; } }
public class B : A { int b; public B() { b = 1; } }
public class C : B { int c; public C() { c = 1; } }
public class D : C { int d; public D() { d = 1; } }
public class E : D { int e; public E() { e = 1; } }
```

The compiler inlines each set of nested base class constructor calls into the new site. (Disassembly 8).

Disassembly 8   Deeply inlined inherited constructors

```
               new A();
00000012 B9 A0 77 3E 00   mov         ecx,3E77A0h
00000017 E8 C4 C7 6C F9   call        F96CC7E0
0000001c C7 40 04 01 00 00 00 mov     dword ptr [eax+4],1

               new C();
00000012 B9 80 78 3E 00   mov         ecx,3E7880h
00000017 E8 14 C6 6C F9   call        F96CC630
0000001c C7 40 04 01 00 00 00 mov     dword ptr [eax+4],1
00000023 C7 40 08 01 00 00 00 mov     dword ptr [eax+8],1
0000002a C7 40 0C 01 00 00 00 mov     dword ptr [eax+0Ch],1

               new E();
00000012 B9 60 79 3E 00   mov         ecx,3E7960h
00000017 E8 84 C3 6C F9   call        F96CC3A0
0000001c C7 40 04 01 00 00 00 mov     dword ptr [eax+4],1
00000023 C7 40 08 01 00 00 00 mov     dword ptr [eax+8],1
0000002a C7 40 0C 01 00 00 00 mov     dword ptr [eax+0Ch],1
00000031 C7 40 10 01 00 00 00 mov     dword ptr [eax+10h],1
00000038 C7 40 14 01 00 00 00 mov     dword ptr [eax+14h],1
```

1. *New rt no-inl L1, …, new rt no-inl L5:* Each type A, …, E has a user-defined constructor that is has been intentionally written to be too expensive to inline. This scenario simulates the cost of creating complex objects with deep inheritance hierarchies and largish constructors.

```
public class A     { int a; public A() { a = 1; if (falsePred) dummy(…); } }
public class B : A { int b; public B() { b = 1; if (falsePred) dummy(…); } }
public class C : B { int c; public C() { c = 1; if (falsePred) dummy(…); } }
public class D : C { int d; public D() { d = 1; if (falsePred) dummy(…); } }
public class E : D { int e; public E() { e = 1; if (falsePred) dummy(…); } }
```

The last five timings in Table 4 show the additional overhead of calling the nested base constructors.

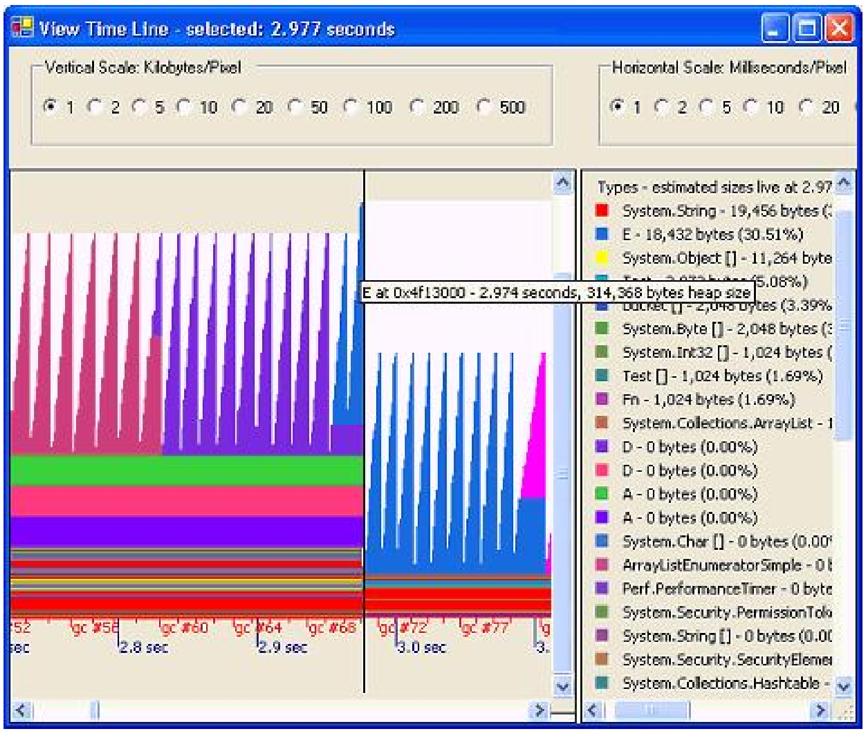
Interlude: CLR Profiler Demo

Now for a quick demo of the CLR Profiler. The CLR Profiler, formerly known as the Allocation Profiler, uses the CLR Profiling APIs to gather event data, particularly call, return, and object allocation and garbage collection events, as your application runs. (The CLR Profiler is an "invasive" profiler, meaning it unfortunately slows the profiled application substantially.) After events have been collected, you use CLR Profiler to explore the memory allocation and GC behavior of you application, including the interaction between your hierarchical call graph and your memory allocation patterns.

CLR Profiler is worth learning because for many "performance-challenged" managed code applications, understanding your data allocation profile provides the critical insight necessary to reduce your working set and so deliver fast and frugal components and applications.

The CLR Profiler can also reveal which methods allocate more storage than you expected, and can uncover cases where you inadvertently keep references to useless object graphs that otherwise could be reclaimed by GC. (A common problem design pattern is a software cache or lookup table of items that are no longer needed or are safe to reconstitute later. It is tragic when a cache keeps object graphs alive past their useful life. Instead, be sure to null out references to objects you no longer need.)

Figure 1 is a timeline view of the heap during execution of the timing test driver. The sawtooth pattern indicates allocation of many thousands of instances of objects C (magenta), D (purple), and E (blue). Every few milliseconds, we chew up another ~150 KB of RAM in the new object (generation 0) heap, and the garbage collector runs briefly to recycle it and promote any live objects to gen 1. It is remarkable that even under this invasive (slow) profiling environment, in the interval of 100 ms (2.8 s through 2.9s), we undergo ~8 generation 0 GC cycles. Then at 2.977 s, making room for another E instance, the garbage collector does a generation 1 garbage collection, which collects and compacts the gen 1 heap—and so the sawtooth continues, from a lower starting address.


View Time Line - selected: 2.977 seconds

Vertical Scale: Kilobytes/Pixel: [✓ 1] 2 5 10 20 50 100 200 500

Horizontal Scale: Milliseconds/Pixel: [✓ 1] 2 5 10 20

E at 0x4f13000 - 2.974 seconds, 314,368 bytes heap size

Types - estimated sizes live at 2.97
- 🟥 System.String - 19,456 bytes (
- 🟦 E - 18,432 bytes (30.51%)
- 🟨 System.Object [] - 11,264 byte
- 🟦 Test - 3,072 bytes (5.08%)
- 🟦 Bucket [] - 2,048 bytes (3.39%
- 🟩 System.Byte [] - 2,048 bytes (
- 🟩 System.Int32 [] - 1,024 bytes (
- 🟦 Test [] - 1,024 bytes (1.69%)
- 🟪 Fn - 1,024 bytes (1.69%)
- 🟫 System.Collections.ArrayList - 1
- 🟪 D - 0 bytes (0.00%)
- 🟥 D - 0 bytes (0.00%)
- 🟩 A - 0 bytes (0.00%)
- 🟪 A - 0 bytes (0.00%)
- 🟦 System.Char [] - 0 bytes (0.00
- 🟪 ArrayListEnumeratorSimple - 0 b
- 🟪 Perf.PerformanceTimer - 0 byte
- 🟩 System.Security.PermissionToke
- 🟪 System.String [] - 0 bytes (0.00
- 🟫 System.Security.SecurityElemen
- 🟦 System.Collections.Hashtable -

Figure1 CLR Profiler Time Line view

Notice that the larger the object (E larger than D larger than C), the faster the gen 0 heap fills up and the more frequent the GC cycle.

Casts and Instance Type Checks

The bedrock foundation of safe, secure, *verifiable* managed code is type safety. Were it possible to cast an object to a type that it is not, it would be straightforward to compromise the integrity of the CLR and so have it at the mercy of untrusted code.

Table 5 Cast and isinst Times (ns)

Avg	Min	Primitive	Avg	Min	Primitive
0.4	0.4	cast up 1	0.8	0.8	isinst up 1
0.3	0.3	cast down 0	0.8	0.8	isinst down 0
8.9	8.8	cast down 1	6.3	6.3	isinst down 1
9.8	9.7	cast (up 2) down 1	10.7	10.6	isinst (up 2) down 1
8.9	8.8	cast down 2	6.4	6.4	isinst down 2
8.7	8.6	cast down 3	6.1	6.1	isinst down 3

Table 5 shows the overhead of these mandatory type checks. A cast from a derived type to a base type is always safe—and free; whereas a cast from a base type to a derived type must be type-checked.

A (checked) cast converts the object reference to the target type, or throws InvalidCastException.

In contrast, the isinst CIL instruction is used to implement the C# as keyword:

```
bac = ac as B;
```

If ac is not B or derived from B, the result is null, not an exception.

Listing 2 demonstrates one of the cast timing loops, and Disassembly 9 shows the generated code for one cast down to a derived type. To perform the cast, the compiler emits a direct call to a helper routine.

Listing 2   Loop to test cast timing

```
public static void castUp2Down1(int n) {
   A ac = c; B bd = d; C ce = e; D df = f;
   B bac = null; C cbd = null; D dce = null; E edf = null;
   for (n /= 8; --n >= 0; ) {
      bac = (B)ac; cbd = (C)bd; dce = (D)ce; edf = (E)df;
      bac = (B)ac; cbd = (C)bd; dce = (D)ce; edf = (E)df;
   }
}
```

Disassembly 9   Down cast

```
            bac = (B)ac;
0000002e 8B D5            mov         edx,ebp
00000030 B9 40 73 3E 00   mov         ecx,3E7340h
00000035 E8 32 A7 4E 72   call        724EA76C
```

Properties

In managed code, a property is a pair of methods, a property getter, and a property setter, that act like a field of an object. The **get_** method fetches the property; the **set_** method updates the property to a new value.

Other than that, properties behave, and cost, just like regular instance methods and virtual methods do. If you are using a property to simply fetch or store an instance field, it is usually inlined, as with any small method.

Table 6 shows the time needed to fetch (and add), and to store, a set of integer instance fields and properties. The cost of getting or setting a property is indeed identical to direct access to the underlying field, *unless* the property is declared virtual, in which case the cost is approximately that of a virtual method call. No surprise there.

Table 6   Field and Property Times (ns)

Avg	**Min**	**Primitive**
1.0	1.0	get field
1.2	1.2	get prop
1.2	1.2	set field
1.2	1.2	set prop

6.4	6.3	get virtual prop
6.4	6.3	set virtual prop

Write Barriers

The CLR garbage collector takes good advantage of the "generational hypothesis"—*most new objects die young*—to minimize collection overhead.

The heap is logically partitioned into generations. The newest objects live in generation 0 (gen 0). These objects have not yet survived a collection. During a gen 0 collection, GC determines which, if any, gen 0 objects are reachable from the GC root set, which includes object references in machine registers, on the stack, class static field object references, etc. Transitively reachable objects are "live" and promoted (copied) to generation 1.

Because the total heap size may be hundreds of MB, while the gen 0 heap size might only be 256 KB, limiting the extent of the GC's object graph tracing to the gen 0 heap is an optimization essential to achieving the CLR's very brief collection pause times.

However, it is possible to store a reference to a gen 0 object in an object reference field of a gen 1 or gen 2 object. Since we don't scan gen 1 or gen 2 objects during a gen 0 collection, if that is the only reference to the given gen 0 object, that object could be erroneously reclaimed by GC. We can't let that happen!

Instead, all stores to all object reference fields in the heap incur a *write barrier*. This is bookkeeping code that efficiently notes stores of new generation object references into fields of older generation objects. Such old object reference fields are added to the GC root set of subsequent GC(s).

The per-object-reference-field-store write barrier overhead is comparable to the cost of a simple method call (Table 7). It is a new expense that is not present in native C/C++ code, but it is usually a small price to pay for super fast object allocation and GC, and the many productivity benefits of automatic memory management.

Table 7 Write Barrier Time (ns)

Avg	**Min**	**Primitive**
6.4	6.4	write barrier

Write barriers can be costly in tight inner loops. But in years to come we can look forward to advanced compilation techniques that reduce the number of write barriers taken and total amortized cost.

You might think write barriers are only necessary on stores to object reference fields of reference types. However, within a value type method, stores to its object reference fields (if any) are also protected by write barriers. This is necessary because the value type itself may sometimes be embedded within a reference type residing in the heap.

Array Element Access

To diagnose and preclude array-out-of-bounds errors and heap corruptions, and to protect the integrity of the CLR itself, array element loads and stores are bounds checked, ensuring the index is within the interval [0,array.Length-1] inclusive or throwing IndexOutOfRangeException.

Our tests measure the time to load or store elements of an int[] array and an A[] array. (Table 8).

Table 8 Array Access Times (ns)

Avg	**Min**	**Primitive**
1.9	1.9	load int array elem
1.9	1.9	store int array elem
2.5	2.5	load obj array elem
16.0	16.0	store obj array elem

The bounds check requires comparing the array index to the implicit array.Length field. As Disassembly 10 shows, in just two instructions we check the index is neither less than 0 nor greater than or equal to array.Length—if it is, we branch to an out of line sequence that throws the exception. The same holds true for loads of object array elements, and for stores into arrays of ints and other simple value types. (*Load obj array elem* time is (insignificantly) slower due to a slight difference in its inner loop.)

Disassembly 10   Load int array element

```
                         ; i in ecx, a in edx, sum in edi
                 sum += a[i];
00000024 3B 4A 04        cmp         ecx,dword ptr [edx+4] ; compare i and array.Length
00000027 73 19           jae         00000042
00000029 03 7C 8A 08     add         edi,dword ptr [edx+ecx*4+8]
…                        ; throw IndexOutOfRangeException
00000042 33 C9           xor         ecx,ecx
00000044 E8 52 78 52 72  call        7252789B
```

Through its code quality optimizations, the JIT compiler often eliminates redundant bounds checks.

Recalling previous sections, we can expect *object array element stores* to be considerably more expensive. To store an object reference into an array of object references, the runtime must:

1. check array index is in bounds;
2. check object is an instance of the array element type;
3. perform a write barrier (noting any intergenerational object reference from the array to the object).

This code sequence is rather long. Rather than emit it at every object array store site, the compiler emits a call to a shared helper function, as shown in Disassembly 11. This call, plus these three actions accounts for the additional time needed in this case.

Disassembly 11   Store object array element

```
                         ; objarray in edi
                         ; obj      in ebx
                 objarray[1] = obj;
00000027 53              push        ebx
00000028 8B CF           mov         ecx,edi
0000002a BA 01 00 00 00  mov         edx,1
0000002f E8 A3 A0 4A 72  call        724AA0D7   ; store object array element helper
```

Boxing and Unboxing

A partnership between .NET compilers and the CLR enables value types, including primitive types like int (System.Int32), to participate as if they were reference types—to be addressed as object references. This affordance—this syntactic sugar—allows value types to be passed to methods as objects, stored in collections as objects, etc.

To "box" a value type is to create a reference type object that holds a copy of its value type. This is conceptually the same as creating a class with an unnamed instance field of the same type as the value type.

To "unbox" a boxed value type is to copy the value, from the object, into a new instance of the value type.

As Table 9 shows (in comparison with Table 4), the amortized time needed to box a int, and later to garbage collect it, is comparable to the time needed to instantiate a small class with one int field.

Table 9   Box and Unbox int Times (ns)

Avg	**Min**	**Primitive**
29.0	21.6	box int

3.0	3.0	unbox int

To unbox a boxed int object requires an explicit cast to int. This compiles into a comparison of the object's type (represented by its method table address) and the boxed int method table address. If they are equal, the value is copied out of the object. Otherwise an exception is thrown. See Disassembly 12.

Disassembly 12   Box and unbox int

```
box            object o = 0;
0000001a B9 08 07 B9 79   mov        ecx,79B90708h
0000001f E8 E4 A5 6C F9   call       F96CA608
00000024 8B D0            mov        edx,eax
00000026 C7 42 04 00 00 00 00 mov        dword ptr [edx+4],0

unbox            sum += (int)o;
00000041 81 3E 08 07 B9 79 cmp        dword ptr [esi],79B90708h ; "type == typeof(int)"?
00000047 74 0C            je         00000055
00000049 8B D6            mov        edx,esi
0000004b B9 08 07 B9 79   mov        ecx,79B90708h
00000050 E8 A9 BB 4E 72   call       724EBBFE                ; no, throw exception
00000055 8D 46 04         lea        eax,[esi+4]
00000058 3B 08            cmp        ecx,dword ptr [eax]
0000005a 03 38            add        edi,dword ptr [eax]      ; yes, fetch int field
```

Delegates

In C, a pointer to function is a primitive data type that literally stores the address of the function.

C++ adds pointers to member functions. A pointer to member function (PMF) represents a deferred member function invocation. The address of a non-virtual member function may be a simple code address, but the address of a virtual member function must embody a particular virtual member function call—the dereference of such a PMF *is* a virtual function call.

To dereference a C++ PMF, you must supply an instance:

```
A* pa = new A;
void (A::*pmf)() = &A::af;
(pa->*pmf)();
```

Years ago, on the Visual C++ compiler development team, we used to ask ourselves, what kind of beastie is the naked expression pa->*pmf (sans function call operator)? We called it a *bound pointer to member function* but *latent member function call* is just as apt.

Returning to managed code land, a delegate object is just that—a latent method call. A delegate object represents both the method to call and the instance to call it upon—or for a delegate to a static method, just the static method to call.

(As our documentation states: A delegate declaration defines a reference type that can be used to encapsulate a method with a specific signature. A delegate instance encapsulates a static or an instance method. Delegates are roughly similar to function pointers in C++; however, delegates are type-safe and secure.)

Delegate types in C# are derived types of MulticastDelegate. This type provides rich semantics including the ability to build an invocation list of (object,method) pairs to be invoked when you invoke the delegate.

Delegates also provide a facility for asynchronous method invocation. After you define a delegate type and instantiate one, initialized with a latent method call, you may invoke it synchronously (method call syntax) or asynchronously, via BeginInvoke. If BeginInvoke is called, the runtime queues the call and returns immediately to the caller. The target method is called later, on a thread pool thread.

All of these rich semantics are not inexpensive. Comparing Table 10 and Table 3, note that delegate invoke is approximately eight times slower than a method call. Expect that to improve over time.

Table 10 Delegate invoke time (ns)

Avg	Min	Primitive
41.1	40.9	delegate invoke

Of Cache Misses, Page Faults, and Computer Architecture

Back in the "good old days", circa 1983, processors were slow (~.5 million instructions/s), and relatively speaking, RAM was fast enough but small (~300 ns access times on 256 KB of DRAM), and disks were slow and large (~25 ms access times on 10 MB disks). PC microprocessors were scalar CISCs, most floating point was in software, and there were no caches.

After twenty more years of Moore's Law, circa 2003, processors are *fast* (issuing up to three operations per cycle at 3 GHz), RAM is relatively very slow (~100 ns access times on 512 MB of DRAM), and disks are *glacially* slow and *enormous* (~10 ms access times on 100 GB disks). PC microprocessors are now out-of-order dataflow superscalar hyperthreading trace-cache RISCs (running decoded CISC instructions) and there are several layers of caches—for example, a certain server-oriented microprocessor has 32 KB level 1 data cache (perhaps 2 cycles of latency), 512 KB L2 data cache, and 2 MB L3 data cache (perhaps a dozen cycles of latency), all on chip.

In the good old days, you could, and sometimes did, count the bytes of code you wrote, and count the number of cycles that code needed to run. A load or store took about the same number of cycles as an add. The modern processor uses branch prediction, speculation, and out-of-order (dataflow) execution across multiple function units to find instruction level parallelism and so make progress on several fronts at once.

Now our fastest PCs can issue up to ~9000 operations per microsecond, but in that same microsecond, only load or store to DRAM ~10 cache lines. In computer architecture circles this is known as *hitting the memory wall*. Caches hide the memory latency, but only to a point. If code or data does not fit in cache, and/or exhibits poor locality of reference, our 9000 operation-per-microsecond supersonic jet degenerates to a 10 load-per-microsecond tricycle.

And *(don't let this happen to you)* should the working set of a program exceed available physical RAM, and the program starts taking hard page faults, then in each 10,000-microsecond page fault service (disk access), we miss the opportunity to bring the user up to *90 million* operations closer to their answer. That is just so horrible that I trust that you will from this day forward take care to measure your working set (vadump) and use tools like CLR Profiler to eliminate unnecessary allocations and inadvertent object graph retentions.

But what does all of this have to do with knowing the cost of managed code primitives? **Everything**.

Recalling Table 1, the omnibus list of managed code primitive times, measured on a 1.1 GHz P-III, observe that every single time, even the amortized cost of allocating, initializing, and reclaiming a five field object with five levels of explicit constructor calls, is *faster* than a single DRAM access. Just one load that misses all levels of on-chip cache can take longer to service than almost any single managed code operation.

So if you are passionate about the speed of your code, it is imperative that you consider *and measure* the cache/memory hierarchy as you design and implement your algorithms and data structures.

Time for a simple demonstration: Is it faster to sum an array of ints, or sum an equivalent linked list of ints? Which, how much so, and why?

Think about it for a minute. For small items such as ints, the memory footprint per array element is one-fourth that of the linked list. (Each linked list node has two words of object overhead and two words of fields (next link and int item).) That's going to hurt cache utilization. Score one for the array approach.

But the array traversal might incur an array bounds check per item. You've just seen that the bounds check takes a bit of time. Perhaps that tips the scales in favor the linked list?

Disassembly 13 Sum int array versus sum int linked list

```
sum int array:            sum += a[i];
00000024 3B 4A 04         cmp         ecx,dword ptr [edx+4]       ; bounds check
00000027 73 19            jae         00000042
00000029 03 7C 8A 08      add         edi,dword ptr [edx+ecx*4+8] ; load array elem
            for (int i = 0; i < m; i++)
0000002d 41               inc         ecx
0000002e 3B CE            cmp         ecx,esi
```

```
00000030 7C F2          jl         00000024


sum int linked list:        sum += l.item; l = l.next;
0000002a 03 70 08        add        esi,dword ptr [eax+8]
0000002d 8B 40 04        mov        eax,dword ptr [eax+4]
            sum += l.item; l = l.next;
00000030 03 70 08        add        esi,dword ptr [eax+8]
00000033 8B 40 04        mov        eax,dword ptr [eax+4]
            sum += l.item; l = l.next;
00000036 03 70 08        add        esi,dword ptr [eax+8]
00000039 8B 40 04        mov        eax,dword ptr [eax+4]
            sum += l.item; l = l.next;
0000003c 03 70 08        add        esi,dword ptr [eax+8]
0000003f 8B 40 04        mov        eax,dword ptr [eax+4]
            for (m /= 4; --m >= 0; ) {
00000042 49             dec        ecx
00000043 85 C9          test       ecx,ecx
00000045 79 E3          jns        0000002A
```

Referring to Disassembly 13, I have stacked the deck in favor of the linked list traversal, unrolling it four times, even removing the usual null pointer end-of-list check. Each item in the array loop requires six instructions, whereas each item in the linked list loop needs only 11/4 = 2.75 instructions. Now which do you suppose is faster?

Test conditions: first, create an array of one million ints, and a simple, traditional linked list of one million ints (1 M list nodes). Then time how long, per item, it takes to add up the first 1,000, 10,000, 100,000, and 1,000,000 items. Repeat each loop many times, to measure the most flattering cache behavior for each case.

Which is faster? After you guess, refer to the answers: the last eight entries in Table 1.

Interesting! The times get substantially slower as the referenced data grows larger than successive cache sizes. The array version is always faster than the linked list version, even though it executes twice as many instructions; for 100,000 items, the array version is seven times faster!

Why is this so? First, fewer linked list items fit in any given level of cache. All those object headers and links waste space. Second, our modern out-of-order dataflow processor can potentially zoom ahead and make progress on several items in the array at the same time. In contrast, with the linked list, until the current list node is in cache, the processor can't get started fetching the next link to the node after that.

In the 100,000 items case, the processor is spending (on average) approximately (22-3.5)/22 = 84% of its time twiddling its thumbs waiting for some list node's cache line to be read from DRAM. That sounds bad, but things could be *much* worse. Because the linked list items are small, many of them fit on a cache line. Since we traverse the list in allocation order, and since the garbage collector preserves allocation order even as it compacts dead objects out of the heap, it is likely, after fetching one node on a cache line, that the next several nodes are now also in cache. If the nodes were larger, or if the list nodes were in a random address order, then every node visited might well be a full cache miss. Adding 16 bytes to each list node doubles the traversal time per item to 43 ns; +32 bytes, 67 ns/item; and adding 64 bytes doubles it again, to 146 ns/item, probably the average DRAM latency on the test machine.

So what is the takeaway lesson here? Avoid linked lists of 100,000 nodes? *No*. The lesson is that cache effects can dominate any consideration of low level efficiency of managed code versus native code. *If* you are writing performance-critical managed code, particularly code managing large data structures, keep cache effects in mind, think through your data structure access patterns, and strive for smaller data footprints and good locality of reference.

By the way, the trend is that the memory wall, the ratio of DRAM access time divided by CPU operation time, will continue to grow worse over time.

Here are some "cache-conscious design" rules of thumb:

- Experiment with, and measure, your scenarios because it is hard to predict second order effects and because rules of thumb aren't worth the paper they're printed on.
- Some data structures, exemplified by arrays, make use of *implicit adjacency* to represent a relation between data. Others, exemplified by linked lists, make use of *explicit pointers (references)* to represent the relation. Implicit adjacency is generally preferable—"implicitness" saves space compared to pointers; and adjacency provides stable locality of reference, and may allow the processor to commence more work before chasing down the next pointer.
- Some usage patterns favor hybrid structures—lists of small arrays, arrays of arrays, or B-trees.

- Perhaps disk-access-sensitive scheduling algorithms, designed back when disk accesses cost only 50,000 CPU instructions, should be recycled now that DRAM accesses can take thousands of CPU operations.
- Since the CLR mark-and-compact garbage collector preserves the relative order of objects, *objects allocated together in time (and on the same thread) tend to remain together in space*. You may be able to use this phenomenon to thoughtfully collocate cliquish data on common cache line(s).
- You may wish to partition your data into hot parts that are frequently traversed and must fit in cache, and cold parts that are infrequently used and can be "cached out".

Do-It-Yourself Time Experiments

For the timing measurements in this paper, I used the Win32 high-resolution performance counter QueryPerformanceCounter (and QueryPerformanceFrequency).

They are easily called via P/Invoke:

```
[System.Runtime.InteropServices.DllImport("KERNEL32")]
private static extern bool QueryPerformanceCounter(
   ref long lpPerformanceCount);

[System.Runtime.InteropServices.DllImport("KERNEL32")]
private static extern bool QueryPerformanceFrequency(
   ref long lpFrequency);
```

You call QueryPerformanceCounter just before and just after your timing loop, subtract counts, multiply by 1.0e9, divide by frequency, divide by number of iterations, and that's your approximate time per iteration in ns.

Due to space and time restrictions, we did not cover locking, exception handling, or the code access security system. *Consider it an exercise for the reader.*

By the way, I produced the disassemblies in this article using the Disassembly Window in VS.NET 2003. There is a trick to it, however. If you run your application in the VS.NET debugger, even as an optimized executable built in Release mode, it will be run in "debug mode" in which optimizations such as inlining are disabled. The only way I found to get a peek at the optimized native code the JIT compiler emits was to launch my test application *outside* the debugger, and then attach to it using Debug.Processes.Attach.

A Space Cost Model?

Ironically, space considerations preclude a thorough discussion of space. A few brief paragraphs, then.

Low-level considerations (several being C# (default TypeAttributes.SequentialLayout) and x86 specific):

- The size of a value type is generally the total size of its fields, with 4-byte or smaller fields aligned to their natural boundaries.
- It is possible to use [StructLayout(LayoutKind.Explicit)] and [FieldOffset(n)] attributes to implement unions.
- The size of a reference type is 8 bytes plus the total size of its fields, rounded up to the next 4-byte boundary, and with 4-byte or smaller fields aligned to their natural boundaries.
- In C#, enum declarations may specify an arbitrary integral base type (except char)—so it is possible to define 8-bit, 16-bit, 32-bit, and 64-bit enums.
- As in C/C++, you can often shave a few tens of percent of space off a larger object by sizing your integral fields appropriately.
- You can inspect the size of an allocated reference type with the CLR Profiler.
- Large objects (many dozens of KB or more) are managed in a separate large object heap, to preclude expensive copying.
- Finalizable objects take an additional GC generation to reclaim—use them sparingly and consider using the Dispose Pattern.

Big picture considerations:

- Each AppDomain currently incurs a substantial space overhead. Many runtime and Framework structures are not shared across AppDomains.
- Within a process, jitted code is typically not shared across AppDomains. If the runtime is specifically hosted, it is possible to override this behavior. See the documentation for CorBindToRuntimeEx and the STARTUP_LOADER_OPTIMIZATION_MULTI_DOMAIN flag.
- In any event, jitted code is not shared across processes. If you have a component that will be loaded into many processes, consider precompiling with NGEN to share the native code.

Reflection

It has been said that "if you have to ask what Reflection costs, you can't afford it". If you've read this far you know how important it is to ask what things cost, and to measure those costs.

Reflection is useful and powerful, but compared to jitted native code, it is neither fast nor small. You've been warned. Measure it for yourself.

Conclusion

Now you know (more or less) what managed code costs at the lowest level. You now have the basic understanding necessary to make smarter implementation trade-offs and write faster managed code.

We have seen that jitted managed code can be as "pedal to the metal" as native code. Your challenge is to code wisely and to chose wisely among the many rich and easy to use facilities in the Framework

There are settings where performance doesn't matter, and settings where it is the most important feature of a product. Premature optimization *is* the root of all evil. But so is careless inattention to efficiency. You're a professional, an artist, a craftsman. So be sure you know the cost of things. If you don't know or even if you think you do—measure it—regularly.

As for the CLR team, we continue to work to provide a platform that is substantially *more productive than native code* and yet is *faster than native code.* Expect things to get better and better. Stay tuned.

Remember your promise.

Resources

- David Stutz et al, *Shared Source CLI Essentials*. O'Reilly and Assoc., 2003. ISBN 059600351X.
- Jan Gray, C++: Under the Hood.
- Gregor Noriskin, Writing High-Performance Managed Applications : A Primer, MSDN.
- Rico Mariani, Garbage Collector Basics and Performance Hints, MSDN.
- Emmanuel Schanzer, Performance Tips and Tricks in .NET Applications, MSDN.
- Emmanuel Schanzer, Performance Considerations for Run-Time Technologies in the .NET Framework, MSDN.
- vadump (Platform SDK Tools), MSDN.
- .NET Show, [Managed] Code Optimization, Sept. 10, 2002, MSDN.