JIT Compiler Optimizations

Earlier in this book we have seen the importance of some optimizations performed by the JIT compiler. Specifically, in Chapter 3 we have discussed inlining in some detail when investigating the method dispatch sequences for virtual and non-virtual methods. In this section we summarize the primary optimizations performed by the JIT compiler, and how to make sure your code does not thwart the JIT compiler's ability to carry out these optimizations. JIT optimizations mostly affect the performance of CPU-bound applications, but are relevant, to some extent, to other types of programs as well.

To inspect these optimizations you must attach the debugger to a running process—the JIT compiler will not perform optimizations when it detects the presence of a debugger. Specifically, when you attach to the process you must make sure that the methods you want to inspect have already been called and compiled.

If for some reason you would like to disable JIT optimizations—for example, to make debugging easier in face of inlining and tail calls (discussed later)—you don't have to modify the code or use a Debug build. Instead, you can create an .ini file that has the same name as your application executable (for example, MyApp.ini) and put the following three lines in it. When placed next to your executable on the next launch, it will inhibit the JIT compiler from performing any optimizations.

```
[.NET Framework Debugging Control]
GenerateTrackingInfo = 1
AllowOptimize = 0
```

Standard Optimizations

There are some standard optimizations performed by every optimizing compiler, even simple ones. For example, the JIT is able to reduce the following C# code to only a couple of x86 machine code instructions:

```
//Original C# code:
static int Add(int i, int j) {
  return i + j;
}
static void Main() {
  int i = 4;
  int j = 3*i + 11;
  Console.WriteLine(Add(i, j));
}
; Optimized x86 assembly code
call 682789a0       ; System.Console.get_Out()
mov ecx,eax
mov edx,1Bh       ; Add(i,j) was folded into its result, 27 (0x1B)
mov eax,dword ptr [ecx]       ; the rest is standard invocation sequence for the
mov eax,dword ptr [eax + 38h]       ; TextWriter.WriteLine virtual method
call dword ptr [eax + 14h]
```

Note It is not the C# compiler that performs this optimization. If you inspect the generated IL code, the local variables are there, as is the call to the `Add` method. The JIT compiler is responsible for all optimizations.

This optimization is called *constant folding*, and there are many similar simple optimizations, such as *common subexpression reduction* (in a statement like `a + (b * a) - (a * b * c)`, the value `a * b` need only be calculated once). The JIT compiler performs these standard optimizations, but often fares considerably worse compared to optimizing compilers such as the Microsoft Visual C++ compiler. The reason is that the JIT compiler has a very constrained execution environment, and must be able to compile methods quickly to prevent significant delays when they are first called.

Method Inlining

This optimization often reduces code size and almost always decreases execution time by replacing method call sites with the callee's body. As we have seen in Chapter 3, virtual methods are not inlined by the JIT compiler (even if a sealed method is called on a derived type); interface methods undergo partial inlining with speculative execution; only static and non-virtual methods can always be inlined. For performance critical code, such as simple properties and methods on very commonly accessed infrastructure classes, make sure to avoid virtual methods and interface implementations.

The exact criteria used by the JIT compiler to determine which methods to inline are not publicly available. Some heuristics can be discovered by experimentation:

- Methods with complex call graphs (e.g. loops) are not inlined.
- Methods with exception handling are not inlined.
- Recursive methods are not inlined.
- Methods that have non-primitive value type parameters, local variables, or return values are not inlined.
- Methods with bodies larger than 32 bytes of IL are not inlined. (The `MethodImplOptions.AggressiveInlining` value of the `[MethodImpl]` attribute overrules this limitation.)

In recent versions of the CLR, some artificial limitations on inlining were removed. For example, as of .NET 3.5 SP1, the 32-bit JIT compiler is able to inline methods that accept *some* non-primitive value type parameters, like Chapter 3's `Point2D`. The changes made in that release replace operations on value types by equivalent operations on primitive types under certain conditions (`Point2D` is transformed to two `int`s), and allow better optimization of struct-related code in general, such as copy propagation, redundant assignment elimination, and others. For example, consider the following trivial code:

```
private static void MethodThatTakesAPoint(Point2D pt) {
  pt.Y = pt.X ^ pt.Y;
  Console.WriteLine(pt.Y);
}
Point2D pt;
pt.X = 3;
pt.Y = 5;
MethodThatTakesAPoint(pt);
```

Using the CLR 4.5 JIT compiler, this entire section of code is compiled to the moral equivalent of `Console.WriteLine(6)`, which is the result of `3 ^ 5`. The JIT compiler is able to use inlining and constant propagation on the custom value type. When using the CLR 2.0 JIT compiler, an actual call to the method is emitted at the call site, and inside the method there is no visible optimization:

```
; calling code
mov eax,3
lea edx,[eax + 2]
push edx
push eax
call dword ptr ds:[1F3350h] (Program.MethodThatTakesAPoint(Point2D), mdToken: 06000003)

; method code
push ebp
mov ebp,esp
mov eax,dword ptr [ebp + 8]
xor dword ptr [ebp + 0Ch],eax
call mscorlib_ni + 0x22d400 (715ed400) (System.Console.get_Out(), mdToken: 06000773)
mov ecx,eax
mov edx,dword ptr [ebp + 0Ch]
mov eax,dword ptr [ecx]
call dword ptr [eax + 0BCh]
pop ebp
ret 8
```

Although there is no way to *force* inlining if the JIT compiler is not willing to perform it, there is a way to turn off inlining. The `MethodImplOptions.NoInlining` value of the `[MethodImpl]` attribute disables inlining of the specific method on which it is placed—incidentally, this is quite useful for micro-benchmarking, discussed in Chapter 2.

Range-Check Elimination

When accessing array elements, the CLR must make sure that the index used to access the array is within the bounds of the array. If this check was not made, memory safety would be compromised; you could initialize a `byte[]` object and index it with negative and positive indices to read/write any location in memory. Although absolutely necessary, this range check has a performance cost of a few instructions. Below is the code emitted by the JIT compiler for a standard array access:

```
//Original C# code:
uint[] array = new uint[100];
array[4] = 0xBADC0FFE;
; Emitted x86 assembly instructions
mov ecx,offset 67fa33aa             ; type of array element
mov edx,64h             ; array size
call 0036215c             ; creates a new array (CORINFO_HELP_NEWARR_1_VC)
cmp dword ptr [eax + 4],4              ; eax + 4 contains the array length, 4 is the index
jbe NOT_IN_RANGE              ; if the length is less than or equal the index, jump away
mov dword ptr [eax + 18h],0BADC0FFEh             ; the offset was calculated at JIT time (0x18 = 8 + 4*4
; Rest of the program's code, jumping over the NOT_IN_RANGE label
NOT_IN_RANGE:
call clr!JIT_RngChkFail             ; throws an exception
```

There is a specific case in which the JIT compiler can eliminate the range-check for accessing array elements—an indexed `for` loop that visits every array element. Without this optimization, accessing arrays would always be slower than in unmanaged code, which is an unacceptable performance hit for scientific applications and memory-bound work. For the following loop, the JIT compiler will eliminate the range check:

```
//Original C# code:
for (int k = 0; k < array.Length; ++k) {
   array[k] = (uint)k;
}
; Emitted x86 assembly instructions (optimized)
xor edx,edx             ; edx = k = 0
mov eax,dword ptr [esi + 4]              ; esi = array, eax = array.Length
test eax,eax             ; if the array is empty,
jle END_LOOP             ; skip the loop
NEXT_ITERATION:
mov dword ptr [esi + edx*4 +8],edx             ; array[k] = k
inc edx             ; ++k
```

```
cmp eax,edx              ; as long as array.Length > k,
jg NEXT_ITERATION              ; jump to the next iteration
END_LOOP:
```

There is only a single check during the loop, and it's the check that makes sure the loop terminates. However, the array access inside the loop is *not* checked—the highlighted line writes to the k-th element in the array without making sure (again) that k is within the array bounds.

Unfortunately, it is also fairly easy to impede this optimization. Some changes to the loop, which appear quite innocent, may have the adverse effect of forcing a range check when accessing the array:

```
//The range-check elimination occurs
for (int k = 0; k < array.Length - 1; ++k) {
   array[k] = (uint)k;
}
//The range-check elimination occurs
for (int k = 7; k < array.Length; ++k) {
   array[k] = (uint)k;
}
//The range-check elimination occurs
//The JIT removes the -1 from the bounds check and starts from the second element
for (int k = 0; k < array.Length - 1; ++k) {
   array[k + 1] = (uint)k;
}
//The range-check elimination does not occur

for (int k = 0; k < array.Length / 2; ++k) {
   array[k * 2] = (uint)k;
}
//The range-check elimination does not occur

staticArray = array; //"staticArray" is a static field of the enclosing class
for (int k = 0; k < staticArray.Length; ++k) {
   staticArray[k] = (uint)k;
}
```

To summarize, range-check elimination is a fragile optimization, and you would do well to make sure that performance-critical sections of your code enjoy this optimization, even if it means that you have to inspect the assembly code generated for your program. For more details on range-check elimination and additional corner cases, see the article "Array Bounds Check Elimination in the CLR" by Dave Detlefs on http://blogs.msdn.com/b/clrcodegeneration/archive/2009/08/13/array-bounds-check-elimination-in-the-clr.aspx.

Tail Call

Tail calling is an optimization that reuses the stack frame of an existing method to call another method. This optimization is very useful for many types of recursive algorithms. In fact, some recursive methods can be as efficient as iteration-based ones if tail call optimization is employed pervasively. Consider the following recursive method, which calculates the greatest common divisor of two integers:

```
public static int GCD(int a, int b) {
  if (b == 0) return a;
  return GCD(b, a % b);
}
```

Clearly, the recursive invocation of GCD(b, a % b) is not subject to inlining—it is a recursive invocation, after all. However, because the caller and callee's stack frames are fully compatible, and because the caller does not do anything after the recursive invocation, a possible optimization would be to rewrite this method as follows:

```
public static int GCD(int a, int b) {
START:
  if (b == 0) return a;
  int temp = a % b;
  a = b;
  b = temp;
  goto START;
}
```

This rewrite does away with all method invocations—effectively, the recursive algorithm has been turned into an iterative one. Although you could perform this rewrite by hand every time you encountered the possibility, the JIT compiler will do it automatically under some circumstances. Below are two versions of the GCD method—the first compiled with the CLR 4.5 32-bit JIT compiler and the second with the CLR 4.5 64-bit JIT compiler:

```
; 32-bit version, parameters are in ECX and EDX
push ebp
mov ebp,esp
push esi
mov eax,ecx              ; EAX = a
mov ecx,edx              ; ECX = b
```

```
test ecx,ecx             ; if b == 0, returning a
jne PROCEED
pop esi
pop ebp
ret
PROCEED:
cdq
idiv eax,ecx             ; EAX = a / b, EDX = a % b
mov esi,edx
test esi,esi             ; if a % b == 0, returning b (inlined base of recursion)
jne PROCEED2
mov eax,ecx
jmp EXIT
PROCEED2:
mov eax,ecx
cdq
idiv eax,esi
mov ecx,esi             ; recursive call on the next line
call dword ptr ds:[3237A0h] (Program.GCD(Int32, Int32), mdToken: 06000004)
EXIT:
pop esi
pop ebp
ret             ; reuses return value (in EAX) from recursive return
; 64-bit version, parameters in ECX and EDX
sub rsp,28h             ; construct stack frame – happens only once!
START:
mov r8d,edx
test r8d,r8d             ; if b == 0, return a
jne PROCEED
mov eax,ecx
jmp EXIT
PROCEED:
cmp ecx,80000000h
jne PROCEED2:
cmp r8d,0FFFFFFFFh
je OVERFLOW             ; miscellaneous overflow checks for arithmetic
xchg ax,ax             ; two-byte NOP (0x66 0x90) for alignment purposes
PROCEED2:
mov eax,ecx
cdq
idiv eax,r8d             ; EAX = a / b, EDX = a % b
mov ecx,r8d             ; reinitialize parameters
mov r8d,edx             ; . . .
jmp START             ; and jump back to the beginning (no function call)
xchg ax,ax             ; two-byte NOP (0x66 0x90) for alignment purposes
EXIT:
add rsp,28h
ret
OVERFLOW:
call clr!JIT_Overflow
nop
```

It becomes evident that the 64-bit JIT compiler uses the tail call optimization to get rid of the recursive method invocation, whereas the 32-bit JIT compiler does not. A detailed treatment of the conditions that the two JIT compilers use for determining whether tail call can be employed is beyond the scope of this book—below are some of the heuristics:

- The 64-bit JIT compiler is quite relaxed in terms of tail calling, and will often perform a tail call even if the language compiler (e.g. the C# compiler) did not suggest it by using the `tail.` IL prefix.
    - Calls followed by additional code (other than returning from the method) are not subject to tail calling. (Slightly relaxed in CLR 4.0.)
    - Calls to a method that returns a different type than the caller.
    - Calls to a method that has too many parameters, unaligned parameters, or parameter/return types that are large value types. (Relaxed considerably in CLR 4.0.)
- The 32-bit JIT compiler is less inclined to perform this optimization, and will emit tail calls only if instructed to do so by the `tail.` IL prefix.

Note A curious aspect of tail calling implications is the case of an infinite recursion. If you have a bug with the recursion's base case that would cause an infinite recursion, but the JIT compiler was able to turn the recursive method call into a tail call, the usual `StackOverflowException` outcome that is the result of infinite recursion turns into an infinite loop!

More details on the `tail.` IL prefix used to suggest tail calling to reluctant JIT compilers and on the criteria used by the JIT compiler to perform tail calling are available online:

- The `tail.` IL prefix, which the C# compiler does not emit, but is used frequently by functional language compilers (including F#) is described on the MSDN, as part of the `System.Reflection.Emit` class page: `http://msdn.microsoft.com/en-us/library/system.reflection.emit.opcodes.tailcall.aspx`.
- The list of conditions for performing a tail call in the JIT compiler (prior to CLR 4.0) is detailed in David Broman's article, "Tail call JIT conditions", at `http://blogs.msdn.com/b/davbr/archive/2007/06/20/tail-call-jit-conditions.aspx`.
- The tail calling optimization changes in the CLR 4.0 JIT compiler are described in depth in the article "Tail Call Improvements in .NET Framework 4", at `http://blogs.msdn.com/b/clrcodegeneration/archive/2009/05/11/tail-call-improvements-in-net-framework-4.aspx`.