

Link-time Code Generation

Matt Pietrek

In all the hoopla surrounding the release of the Microsoft® .NET Framework and Visual Studio® .NET, a really slick new feature in Visual Studio .NET is often overlooked. This enhancement is a special goodie for those of you who still write unmanaged code in good old C++. The new feature goes by a couple of names: whole program optimization and link-time code generation. John Robbins covered these features briefly in the August 2001 installment of Bugslayer. This month, I'll take the time to really drill into the subject and show you just how cool it can be. To keep things easy to read, I'll refer to link-time code generation as LTCG throughout this column.

To understand how LTCG improves matters, it's important to know all the players. When you build a plain old non-.NET program, three primary components that come into play: the front end, the back end, and the linker. The first of these is the compiler front end. In Visual C++®, the front end is comprised of C1.DLL and C1XX.DLL. The front end takes your source code, tokenizes it, parses it, verifies its syntactic correctness, and so on. The output of the front end is intermediate language (IL).

The IL represents all the flow constructs, variable declarations, and so on in your code. However, it is a binary format and not easily read by humans. Comments, spacing, variable names, and so on are unimportant at this point. Theoretically, IL is not specific to any particular CPU. Instead, it is portable across any CPU that the compiler might target.

The second player is the back end, which is also known as the code generator because it takes the IL from the front end and generates code targeted to the CPU. In Visual C++, C2.DLL is the back end. The back end is also where optimizations take place. This is a key point that I'll discuss later. The usual output from the back end is an .OBJ file that has processor-specific instructions, as well as any data that's declared. You can see the processor-specific code for yourself by running DUMPBIN /disasm on an .OBJ file.

The final player is the linker. In Visual C++, the linker is LINK.EXE. The linker's job is to take all the OBJ files, along with any supplied .LIB files, and create an executable image. This usually consists of merging all the code from the contributing OBJs into a single section in the executable. Ditto for data. Of course, there are things like performing fixups that make the linker more than a simple merge utility. The rules that the linker must follow are pretty simple on a grand scale, but get pretty esoteric when you drill down a bit. Parts 1 and 2 of "Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format" in the February 2002 and March 2002 issues describe some of the rules that the Microsoft linker follows.

But What About .NET?

In the .NET universe, these players have slightly different roles. The end result of the compiler front end and back end isn't processor-specific code. It's IL, but not the same IL that the Visual C++ compiler uses when creating a classic executable. Instead, the IL for .NET goes into the executable file. At run time, the .NET JIT compiler takes the IL and generates real code in dynamically allocated memory. The fact that the actual code for a method can exist pretty much anywhere in memory is one reason why .NET has completely different debugging and profiling APIs than Win32®-based programs.

The linker is still used in .NET for certain languages (in particular, Visual C++ with managed extensions). It performs much the same job as it does when creating a regular executable. The primary difference is that instead of working with actual code when merging together all the OBJ's code sections, it's working with IL.

The Normal Build Process

In every version of Visual C++ up to Visual C++ .NET, and when using Visual C++ .NET without LTCG, the compiler front end invokes the back end to generate code and corresponding OBJ files.

Link-time Code Generation

Matt Pietrek

Next, the IDE or the make file invokes the linker, supplying it with the list of OBJ files and LIB files to process. (If you run CL from the command line without using the /c switch, it will invoke the linker for you.)

It's important to note that in a normal build, the linker is working with actual CPU-specific code. While the linker is a pretty clever piece of code, it's not so smart that it will start making modifications to the code in the OBJ files supplied to it. Basically, the linker's hands are tied. It can only work with the code that's handed to it by the back end.

You might be thinking, "why is this a bad thing?" After all, the Visual C++ back end can do some pretty nifty optimizations. But there are limits to what it can do, primarily because of boundary conditions and lack of knowledge.

An example of a boundary condition is when one function calls another function in a different source file. Because the compiler has limited knowledge of what the other function looks like, it has to rely on calling conventions to ensure it's called correctly. A calling convention specifies that parameters will be passed at certain locations on the stack or in certain registers, the order in which parameters are passed, and who will clean up the stack if necessary. The three prevalent calling conventions are cdecl (the default), stdcall, and fastcall.

While the use of calling conventions ensures a certain level of safety, they also potentially add to the amount of code generated. To ensure that the calling conventions are adhered to, the code generator may need to add additional instructions that might not be absolutely necessary.

Another example of when lack of knowledge hinders the optimizer occurs with function inlining. Let's say in one source file you have a function like this:

```
int DoubleTheValue( int x )
{
    return x * 2;
}
```

In a second source file, you call that function. Unfortunately, when generating the code for the second source file, the back end has no knowledge of the DoubleTheValue function, other than that it exists elsewhere. If the code generator knew that the DoubleTheValue function was so simple, it might just incorporate its logic into the function it was currently working on. This process is known as inlining a function.

In the past, you could frequently get a function to be inlined by declaring it in a header file, and including that header file appropriately. This happened often with small C++ class methods such as accessor methods like get and set. However, if the back end hasn't seen the source code for a function during the current compilation run, there's no way it can inline it. Thus, functions in another source file, no matter how small, would never be inlined.

Link-time Code Generation

To get around these limitations, Visual C++ .NET introduces link-time code generation. When building with LTCG, the compiler front end doesn't invoke the back end. Instead, it emits an OBJ file with IL in it. It bears repeating: this IL is not the same IL that the .NET runtime uses. While .NET IL is standardized and documented, the IL used with LTCG is undocumented and subject to change from version to version of the compiler.

Link-time Code Generation

Matt Pietrek

When the linker is invoked with IL-based OBJ files, it calls COM methods in the back end (C2.DLL) to generate the final, processor-specific code. When the linker invokes the back end, it gives the back end all of the IL from all the OBJs (as well as any pregenerated code like you might find in a .LIB file.) Because the code generator now has almost perfect knowledge of the code, it can be much more aggressive when it optimizes. The primary changes you'll notice from LTCCG-generated code is that many more functions are inlined, and that many functions are called without using one of the standard calling conventions. I'll drill into some of these later.

It's worth noting that neither the linker nor the compiler back end got significantly smarter here. The major work in implementing LTCCG was the juggling required to allow the back end to be called from either the front end or from the linker. The linker also was smartened up so that it could work with a combination of IL-based OBJ files and normal OBJ and LIB files already containing actual code.

To be fair to other companies, Microsoft did not invent the concept of LTCCG. One notable earlier implementation was the Intel C++ compiler for Windows®. The equivalent functionality is termed Inter Procedural Optimization or IPO (cute acronym, eh?).

Building With LTCCG

Hopefully by this point, you're excited enough to want to try LTCCG with your own projects. To use LTCCG from the Visual Studio .NET IDE is as simple as ensuring that the Whole Program Optimization option is set to Yes. This can be found in the project's Configuration Properties | General pane.

To use LTCCG from the command line, you need to inform both the compiler and linker what's going on. The compiler switch to enable link-time code generation is /GL. If you invoke the linker, make sure to add the /LTCCG command-line option. If you forget to add /LTCCG and give the linker an OBJ with IL in it, the linker restarts with /LTCCG enabled.

Drilling into LTCCG Optimizations

Enough theory. Let's see firsthand what LTCCG can do for you. Although there are a variety of optimizations enabled by LTCCG, I'll focus on three here: cross-module inlining, custom calling conventions, and smaller thread local storage (TLS) access. Warning: in this section, I assume you can read x86 assembly language. It's hard to show the benefits of these optimizations if you don't know the basic x86 constructs.

To see the effects of cross-module inlining, first examine the source files for inlining.cpp and inlining2.cpp in Figure 1 and Figure 2. This is a do-nothing program that illustrates how the back end can embed the code for Goober within the main program.

To build the program in Visual C++ 6.0, use the command line:

```
CL /O2 inlining.cpp inlining2.cpp
```

Figure 3 is an annotated assembly listing that shows the code generated by Visual C++ 6.0. It's not much of a surprise. As you'd expect, function main makes a cdecl style call to the Goober function. Now, let's see how LTCCG can improve things. Switching to Visual C++ .NET, add the /GL switch to the previous command line like so:

```
CL /GL /O2 inlining.cpp inlining2.cpp
```

The result is much better code, as seen in Figure 4. Without inlining, four instructions are needed to make the call, including two PUSH instructions to put parameters on the stack. The Goober function

Link-time Code Generation

Matt Pietrek

itself takes seven instructions. The MOV EAX,[ESP+4] and RET instruction in Goober's code are unnecessary if Goober is inlined. With inlining, Goober's code in main only uses five instructions. The net effect of inlining here is a saving of six instructions.

The next optimization that LTCG adds is custom calling conventions. Normally, all functions are either cdecl, stdcall, or fastcall. With custom calling conventions, the back end has enough knowledge that it can pass more values in registers, and less on the stack. This usually cuts code size and improves performance.

To see the benefits of custom calling conventions, examine CCC.CPP in Figure 5. The foo function doesn't do anything significant. It does take four parameters (two integers and two pointers). Likewise, the main function doesn't do anything worthwhile, except to call the foo function.

Incidentally, the compiler desperately wants to inline function foo inside main. However, in doing so, it would obscure the effects of custom calling conventions. Therefore, I cheated and used the `__declspec(noinline)` directive on foo to prevent it from being inlined. Build CCC.CPP from the command line like so:

CL /GL /O2 CCC.CPP

Figure 6 shows the annotated assembly code for functions foo and main. Again, I have preceded each instruction group by a comment that identifies which source line is responsible for it. The key point is that all four parameters were passed in registers (EAX, EDX, ESI, and EDI). Even the fastcall convention passes at most two parameters in registers.

It's also interesting to note that the back end remembers subexpression calculations. In the line

```
*k = i + l;
```

the value of `i + l` is calculated and stored in ECX.

In the next line

```
return i + *j + *k + l;
```

`i` and `l` are added again. The compiler remembers that it has already calculated this value previously, and reuses the value previously stored in ECX.

The benefits of custom calling conventions don't come only from functions in other source modules. To use a custom calling convention for a function, the back end must see and generate all the calls to the function at the same time. This is the only way to ensure that the function is always called correctly. If there's even a chance that a function might be called from previously generated code, the compiler backs off and decides not to use a custom calling convention at all.

The final performance improvement from LTCG that I'll look at is called Small TLS Encoding. When you use `__declspec(thread)` variables, the code generator stores the variables at a fixed offset in each per-thread data area. Without LTCG, the code generator has no idea of how many `__declspec(thread)` variables there will be. As such, it must generate code that assumes the worst, and uses a four-byte offset to access the variable.

With LTCG, the code generator has the opportunity to examine all `__declspec(thread)` variables, and

Link-time Code Generation

Matt Pietrek

note how often they're used. The code generator can put the smaller, more frequently used variables at the beginning of the per-thread data area and use a one-byte offset to access them.

Figure 7 shows a simple do-nothing program that declares and uses a `__declspec(thread)` variable named `i`. Figure 8 shows the results of compiling as follows:

```
CL /O2 tls.cpp
```

I won't attempt to explain the code gyrations in the generated code. It is important to note that the "mov dword ptr [ecx+4],eax" instruction uses a four-byte encoding for the +4 value. The whole routine takes 0x1B bytes.

Adding the `/GL` option to enable LTCG generates much better code, as shown in Figure 9. Not only does the code use two fewer instructions, the storage instruction "mov dword ptr [ecx+8],eax" uses only one byte to encode the value +8. Here, the whole routine takes only 0x11 bytes for a code size savings of 37 percent.

Microsoft claims that on some programs, LTCG boosts performance by 10 to 15 percent. A boost of 3 to 5 percent is more common for real world x86 programs. In my own informal test, I found a similar improvement on code size and about a 15 percent improvement in speed.

Details Behind LTCG Decision Making

One of my very first questions about LTCG had to do with how it decides when to inline a function. Playing around with simple scenarios didn't yield a good answer, so I did the next best thing: I asked. Here's the general algorithm that the code generator uses when deciding whether to inline a function or not.

Any function that may be inlined has a cost associated with it. If the cost is below a certain threshold, the inlining occurs. The thresholds when inlining for size and speed are different. The size of the application also affects the threshold when optimizing for speed. Smaller applications are more aggressively inlined than large ones, which means the threshold value is higher for small programs.

The cost of a function begins with its initial size in non-inlined form. Various factors increase or decrease the cost. If parameters with constant values are passed to a function which uses those parameters in if or switch statements, the cost is reduced. In addition, the overhead of setting up for a function call (including pushing parameters) is subtracted from a function's cost. Functions that don't branch have their cost reduced, as linear code is more easily optimized.

If compiling for size, the number of calls to a function affects its cost. If a function is only called once, its cost is heavily reduced. If it's called only a few times, its cost is moderately reduced. A function that's called all over the place will have its cost increased, as the inlining will likely increase the code size of the final program.

When compiling for speed, the cost of a function goes up if the function contains large loops. Large loops aren't easily optimized. If the call is heavily nested inside if statements, the cost increases, as the benefits of inlining will only be seen when all right conditions are met. If the function call is inside nested loops, its cost is decreased, making it more likely to be inlined.

After going through all these heuristics, a final cost is calculated. If it's below the target threshold, the inlining occurs. Of course, a function will never be inlined if the code generator can't determine all the call sites of a given function in advance. This topic itself raises another interesting question: how does

Link-time Code Generation

Matt Pietrek

the code generator know which functions it can safely inline or generate custom calling conventions for?

The brief answer is that the first condition needed for effective use of LTCG is that the function exists in IL form, rather than in its final CPU-specific form. Thus, calls to functions in static libraries can't be optimized. There's also a set of boundary conditions that cause the code generator to back off and use standard rules.

When the linker invokes the back end, the code generator creates a control flow graph for as much of the program as possible. The control flow graph is a big picture of what calls what, and what all the dependencies between functions are. Certain items cause code to go outside of the normal flow graph, and hence lose the ability to be optimized. The big killers here are DLLs. Any calls to or from a DLL go outside the visible flow graph, and thus aren't candidates for LTCG optimization. Standard conventions must apply to any code that is declared external, is exported or imported, or has its address taken. The same goes for variables.

Limitations on LTCG Use

While LTCG is generally a good thing, there are a few potential pitfalls that might affect you. First, precompiled headers and LTCG are incompatible. This shouldn't be an issue for most users, since you typically only turn on LTCG in release builds, where compilation time usually isn't a problem.

Next, the OBJ files produced when using LTCG aren't standard COFF format OBJs. Again, this isn't a problem for most people, but if you examine OBJ files with tools like dumpbin, you're simply out of luck—it won't work.

Finally, there's the subject of libraries. It's possible to create .LIB files with code in its IL form. The linker will happily work with these .LIB files. Be aware that these libraries will be tied to a specific version of the compiler and linker. If you distribute these libraries, you'll need to update them if Microsoft changes the format of IL in a future release.