Xbox 360 Compiler Technology

By Pete Isensee
Development Manager
Advanced Technology Group (ATG)

Published: August 13, 2004
Updated: October 29, 2008

Compilers are one of the most critical tools for making games. Although Microsoft has produced excellent compilers over the years, Xbox 360 represents some new challenges, both with respect to the PowerPC architecture and the multithreaded nature of Xbox 360. This white paper explains the compiler technology available to Xbox 360 developers.

The Good Old Days

The compiler included in the Xbox Development Kit is Microsoft Visual C++ .NET 2003; it is technically known as the "Microsoft 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86." This compiler is the same as the Win32 version of the same compiler (though there are a few enhancements in the Xbox version of the IDE). Because Xbox uses an 80x86 part (Pentium III), this compiler technology is perfectly suited to the hardware. The compiler was developed by the Microsoft Visual Studio team, and updated only a couple of times over the life of the Xbox development kit (XDK) to match the shipping Visual Studio version.

A New Frontier

The Xbox 360 team evaluated a number of options for compilers. Writing a new compiler from scratch was out of the question. Game developers push compilers to their limits, and there was no reason to abandon the thousands of engineering-years of investment in the existing Microsoft compiler. There was also another interesting factor to consider. Not so long ago, Microsoft Windows NT ran on PowerPC, and Microsoft created a compiler for that architecture, which was last externally supported in Microsoft Visual Studio 4.2 Cross-Platform Edition for Macintosh. We made a decision with three key points.

- Use the existing front end from the Visual Studio development system and update it with regular releases of the standard product. The front end is a proven, stable technology that can compile advanced C++ template libraries with ease.
- Build on the back end of Visual Studio 4.2 Cross-Platform Edition for Macintosh. The back end was fairly old and required many updates. But building on an existing framework was better than starting from scratch and less costly than finding other technology.
- Hire an in-house compiler team with PowerPC experience. Finding good compiler writers is extremely difficult. But we knew that it was essential if we were to deliver the best console compiler on the planet.

An Aside: Compilers 101

Most modern C++ compilers operate in at least two passes. The first pass is performed by what's known as the *front end*. The front end performs macro preprocessing and lexical analysis; it reports syntactical errors and warnings. If the code is semantically correct, the front end generates machine-independent intermediate language output. It's the job of the front end to be compliant with the C++ language standard.

The *back end* performs the task of converting the machine-independent intermediate language to actual machine code. The Xbox 360 back end produces the Xbox 360 CPU machine code. It's the job of the back end to produce code that runs really, really fast.

The Xbox 360 Compiler Today

The decisions that we made have proven to be good choices. The Xbox 360 compiler was originally based on the front end of Visual C++ 2005. It now uses the front end of Visual C++ 2010.

The Xbox 360 back end is a world-class compiler. It produces code that is both small and fast. The compiler team continues to make small improvements and adjustments based on your feedback, but the code that the compiler generates is generally quite efficient. Most additional optimizations are better introduced in the context of your game engine itself. However, there are still edge cases where the compiler may produce inefficient code. Send your code snippets to your developer account manager, or visit the Entertainment Developer Forum, whenever you believe the compiler could be improved. In terms of code size, executables are generally only 20 percent larger than x86 versions, which is quite good given that many RISC compilers generate code that is 30-50 percent larger than equivalent CISC versions.

Compiler and Library Features and Coming Upgrades

The remainder of this document discusses the various features of the compiler and libraries.

C++ Conformance

Unlike the move to Visual C++ .NET 2003, which usually required developers to make a number of code changes, the move to Visual C++ 2010 is more straightforward.

Specific improvements that have been made in Visual C++ include:

- Unicode support. The compiler accepts Unicode source files and also supports the \u and \U escape sequences.

- Improved support of friends with templates. For instance, the compiler now correctly requires that a template friend of a template must be templatized in the friend statement—for example:

```
template< typename T > class task { ... };
template< typename T > class schedule
{
   friend class task; // no longer compiles in Visual C++ 2005
   friend class task< T >; // compiles in Visual C++ 2008 and
                           // Visual C++ 2010
};
```

- Type restrictions have been tightened up, particularly for const and volatile variables. Most cases are esoteric edge cases that would not occur in typical code. For instance, assigning **T*** to **const T*&** is now correctly considered an error:

```
int* p;
const int*& q = p; // no longer compiles in Visual C++ 2005
const int* const& cr = p; // compiles in Visual C++ 2008 and
                          // Visual C++ 2010
int*& r = p; // compiles in Visual C++ 2008 and Visual C++ 2010
```

- Type restrictions on pointers to members have also been tightened up:

```
struct obj { void fn( int ); };
typedef void (obj::*pfn)( int );
pfn f = obj::fn; // no longer compiles in Visual C++ 2005
pfn f = &obj::fn; // compiles in Visual C++ 2008 and Visual C++ 2010
obj x;
(x.*f)( 42 );
```

The list of C++ Standard features that are not supported in Visual C++ is very short:

- The **export** keyword. For good reasons why **export** is not as useful as originally thought, see the following report to the C++ Standards committee: "Why We Can't Afford Export" by Herb Sutter and Tom Plum.
- Two-phase lookup. Two-phase lookup refers to name lookup on templated functions and classes. Two-phase lookup is related to export, which partially explains why it's not implemented. The other reason it's not implemented is that it can break existing code.
- Exception specifications. These continue to be ignored by the compiler. Modern coding standards recommend against the use of exception specifications. For an exploration of the challenges of exception specifications, see "A Pragmatic Look at Exception Specifications" by Herb Sutter.

C++ Technical Report 1 (TR1)

C++ Technical Report 1 (TR1), ISO/IEC TR 19768, is a set of proposed extensions to C++ that include expressions, smart pointers, hash tables, random number generators, improved function objects, and more. The Xbox 360 compiler now supports TR1.

C Conformance

Visual C++ has not adopted some of the more obscure new features of the C99 Standard, including variable-length arrays and designated initializers. However, the compiler does implement *variadic macros*. Variadic macros allow you to pass any number of macro parameters using the same syntax as C functions with variable-length argument lists. This allows you to create macros like this:

```
#define LOG( s, ... ) fprintf( LogFile, s, __VA_ARGS__ )
```

And call them like this:

```
LOG( "Begin level load: %s (Type=%d)", strLevel, iType )
```

Compiler Warnings

The Visual C++ compiler includes more warnings for C++ syntactical problems. Here's one example:

```
__int64 x = 1 << 48;
```

The Visual C++ .NET 2003 compiler would happily set *x* to 0, since it interprets the 1 as a 32-bit value. In Visual C++ 2008 and Visual C++ 2010, the compiler reports "shift count negative or too big; undefined behavior," prompting the developer to realize what he or she really meant to write was:

```
__int64 x = (__int64)1 << 48;     // or 1i64 << 48
```

Compilation Speed

Visual C++ 2005 removed the automatic generation of precompiled headers, because it turns out that in most projects, automatic precompilation actually *increases* compile times and is incompatible with Link Time Code Generation. We still recommend you use precompiled headers—they can drastically improve build speeds. You just need to exercise care in what header files you precompile. See MSDN for details. The Xbox 360 sample code shared library shows best practices for precompilation.

Visual C++ 2008 and Visual C++ 2010 have the ability to use multiple processors to improve compilation times. The build system can perform multiprocessor builds at both project-level and file-level granularity. File-level multi-processor builds require using the **/MP** compiler switch and can significantly accelerate the compilation, even of solutions that contain only a single project.

Code Analysis

Visual C++ 2005 introduced **/analyze** and SAL annotations. The **/analyze** compiler switch tries to find bugs in your code, looking more deeply than during a normal compilation. SAL annotations are special macros which give the analysis phase of the compiler more information—particularly about buffer sizes when calling functions—so that more errors can

be detected. The code analysis and SAL annotations were further improved for Visual C++ 2008 and Visual C++ 2010.

Restrict Keyword

Visual C++ 2005 introduced a new keyword: __**restrict**. The best way to show the usefulness of this keyword is by an example. Given the following, the compiler must assume that *p could* point into *pArr*, even though in practice it never actually would point into *pArr*:

```
void Add( float* pArr, const float* p )
{
   for( int i = 0; i < n; ++i )
      pArr[i] = *p + 1.0f;
}
```

That means that every iteration of the loop, the compiler must reload the value **p* from memory and incur the resulting performance penalty. In other words, even though you'd probably never write code like the following, the compiler must correctly handle this scenario:

```
a[0] = 1.0;
Add( a, &a[0] ); // evil, but technically allowed
```

This performance curse is called *pointer aliasing*. The best way to avoid these problems is to minimize the use of pointers (notice that *p* could just as easily have been passed by value) or hoist pointer invariants out of loops. However, there are some cases where the pointer is necessary and you'd like to tell the compiler that you know what you're doing. The **__restrict** keyword allows you to do exactly that:

```
void Add( float* __restrict pArr, const float* __restrict p )
{
   // Make sure p doesn't overlap pArr
   assert( p + 1 <= pArr || p >= pArr + n );
   for( int i = 0; i < n; ++i )
      pArr[i] = *p + 1.0f;
}
```

Notice that the keyword goes after the star. In this example, the first **__restrict** is the most important. It tells the compiler that no other objects refer to the array (it's "restricted"), so the compiler can automatically hoist the **p* out of the **for** loop and improve performance. If your game still happens to invoke **Add()** in the evil way shown above, all bets are off, because you've promised the compiler that nothing could point into the float array. If you don't keep your promise, the **__restrict** keyword won't buy you anything but painful debugging sessions. Used properly the __restrict keyword allows the compiler to avoid redundant loads, and it also gives the compiler more scheduling flexibility by letting it reorder loads and stores more freely.

You can learn more about pointer aliasing at:

- Draft of C Standard Proposal for Restricted Pointers
- Christer Ericson's Memory Optimization talk from GDC 2003
- The Aliasing section of the Intel compiler User Guide
- SGI white paper on pointer aliasing

OpenMP

OpenMP is a simple and flexible interface for building multithreaded programs. The specification and details can be found on the web site of the OpenMP Architecture Review Board. OpenMP consists of a small set of pragmas used to instruct the compiler about which sections of code should be parallelized. Here's an example:

```
#pragma omp parallel
for( int i = 0; i < nMax; ++i )
   a[i] = ( b[i] * c[i] ) + ( d[i] * e[i] );
```

The **#pragma** tells the compiler that the code can be split up across multiple processors. One thread might evaluate the first *dwMax*/2 operations and the other thread would evaluate other half. OpenMP on Xbox 360 uses six threads by default. Developers can specify how many OpenMP threads are used and what CPUs those threads reside on.

OpenMP allows Xbox 360 developers to take advantage of Xbox 360's hardware threads without having to use specific threading functions. Significant performance gains can be obtained by using OpenMP in key sections of a game. For example, initial results of parallelizing the PointSprites sample show framerate improvements of up to 50 percent. However, OpenMP typically only helps you multithread small portions of your game. Keeping all six hardware threads of the Xbox 360 CPU fully utilized usually requires custom threading code.

64-bit Support

The Xbox 360 CPU is a 64-bit processor; for most computations there is no performance cost to using **__int64** instead of plain old **int**. Pointers, however, are 32-bit values, primarily because there's no need for a 64-bit address space. On Xbox 360, **sizeof(int)**, **sizeof(long)**, and **sizeof(void*)** are all 4 bytes. For more details on why this is so, see Matt Pietrek's article.

Because the Xbox 360 CPU is so efficient at bit-level operations like shifts, rotates, and masking, it can be useful to store multiple values in 64-bit unsigned integers. The Xbox 360 compiler automatically optimizes bit operations to use the built-in functionality of the CPU.

Intrinsics

Intrinsics are C functions that expose underlying CPU functionality. They allow games to directly access the CPU instruction set without having to worry about register allocation and other "gotchas" that direct assembly language code must handle. Although inline assembly is supported on Xbox 360, we highly encourage the use of compiler intrinsics.

The Xbox 360 compiler intrinsics can be found in the ppcintrinsics.h and vectorintrinsics.h files in the Xbox 360 XDK. Most intrinsics map directly to Xbox 360 CPU instructions, so you can find out details on how they operate by consulting the PowerPC Programming Environments Manuals included in the Xbox 360 XDK. For example uses of Xbox 360 intrinsics, see the FastCPU sample and the Xbox 360 math library in Xboxmath.h plus the corresponding *.inl files. For details of using the VMX intrinsics, see VMX128 Instruction Set Summary.

Optimizing for the Xbox 360 CPU

One of the important jobs of any compiler is producing code that truly makes the most of the hardware. The Xbox 360 compiler has a distinct advantage over more generic compilers in that it understands precisely what hardware to target. The compiler knows how to best schedule code for the Xbox 360 CPU pipelines.

Tuning the compiler to produce the most efficient code is an iterative process. If you discover places where you believe the compiler could do a better job at optimization, send your code snippet to your developer account manager or visit the Entertainment Developer Forum. Real-world examples often uncover hidden blemishes.

C Runtime Library (CRT)

The most important changes in the CRT are the Xbox 360-specific optimizations being done in key functions. For instance, buffer functions such as **memcpy** and **memcmp** have been optimized to manipulate data in 64-bit chunks and take advantage of other processor-specific features. Mathematical functions such as **sin**, **floor**, and **sqrt** have been updated to use Xbox 360 CPU compiler intrinsics.

With the latest version of the CRT, **operator new** throws by default. If **new** fails to allocate memory, it throws **std::bad_alloc**. To disable this feature, you can use the std::nothrow version of **new**.

The next major change in the CRT is the addition of over 400 new, more secure functions. Here's an example of the safer **strcpy** function:

```
char Dest[6];
strcpy_s( Dest, ARRAYSIZE(Dest), "Hello" );
```

When the destination of one of the safer CRT string functions is an array, then the compiler can infer the size of the array and the code can, without loss of safety, be simplified to the following:

```
char Dest[6];
strcpy_s( Dest, "Hello" );
```

If the compiler can infer the buffer size, then the preceding code will compile. If it cannot infer the buffer size, then the code will fail to compile, and you will need to manually specify the buffer size. If the caller does something incorrect like the following:

```
strcpy_s( Dest, "Hello, world" ); // String is too long
```

Then the API will assert in debug builds and force a run-time termination in release builds instead of corrupting memory and crashing at some point in the future.

Not only do the new functions make it much easier to write secure code, they're also a great debugging tool. For example, they check input parameters for NULL pointers. Functions, such as **printf**, that take format strings validate the format string. The C and C++ Standards committees are already considering the adoption of these functions (see this report), so the functions are likely to end up on other platforms as well. Xbox 360 developers will continue to be able to use the less secure functions if they wish.

One other addition to the CRT is a static assertion macro called **_STATIC_ASSERT**. It's useful for validating assumptions that can be checked at compile time:

```
#include "crtdbg.h" // _STATIC_ASSERT lives here
_STATIC_ASSERT( sizeof(OldObj) == sizeof(NewObj) );
```

Standard Template Library (STL)

The STL has been improved, primarily for debugging and security reasons. Algorithms can detect whether an iterator range is invalid. You cannot accidentally mix iterators from different containers. Iterators will now detect when they've walked off the end of a container. For instance, consider the following code:

```
std::vector<int> v(10);
std::vector<int> w(5);
std::copy(v.begin(), v.end(), w.begin()); // uh-oh
```

The **copy** function has undefined behavior, since vector **w** doesn't have enough room to hold 10 elements.

With Visual C++, the STL detects this condition and will call **__debugbreak**. Two flags enable these checks:

- _HAS_ITERATOR_DEBUGGING (enabled in debug builds; disabled in release builds)
- _SECURE_SCL (always enabled)

The great thing is that these validations don't require any code changes. Without touching a single line of code, developers using STL automatically get a much higher level of validation and error checking. _HAS_ITERATOR_DEBUGGING will sometimes have a significant performance impact, which is why it is only enabled in debug builds. _SECURE_SCL is designed to have low overhead, but in some cases, it may cause a measurable slowdown.

Link Time Code Generation (LTCG)

Link Time Code Generation is also known as Whole Program Optimization (WPO). WPO tells the compiler to produce object files that contain the machine-independent intermediate language rather than machine language. The linker is then able to optimize all of these files as a single unit, providing better inlining opportunities, elimination of copy construction on return value temporaries, copy-propagation of constants across functions, partial function body inlining, and so forth. The compiler supports LTCG via the **/GL** compiler option and

/LTCG linker option. To enable WPO, choose **Project** | **Properties** | **Configuration Properties** | **General** | **Whole Program Optimization**: **Use Link Time Code Generation**.

Profile Guided Optimizations (PGO)

Visual C++ includes a new optimization technology based on runtime evaluation of an executable. Until now, Microsoft compilers have only performed static optimizations. With PGO, profiled builds of a game are instrumented at runtime, and the results are fed back into the linker, which produces a final executable that has been optimized based on what code has *actually being executed the most often*. By doing focused optimizations on code that is being called by every frame of the game, the linker can produce a highly optimized build. For instance, functions can be ordered within the image to improve cache coherency, common and even quite large functions can be inlined, and registers can be allocated more effectively. More information on PGO can be found on MSDN.

As a real-world example, Microsoft SQL Server was recompiled and linked using PGO and obtained a 30-percent performance improvement in many common scenarios. We expect that games can see similar improvements. One caveat is that PGO-enabled game builds require additional memory during the profile recording process. Games that are near the limit of Xbox 360 RAM may not be able to use PGO successfully. Developers should consider having a low-memory mode for their game (perhaps discarding top-level mip-maps) in order to leave enough memory for recording PGO profile data. This extra memory can also be useful for other debugging and profiling purposes, such as doing XbPerfview captures.

Because PGO optimizes your game based on the scenario that you measure, it is important to measure a representative scenario. The ideal scenario is to start your game with data recording disabled so that you aren't optimizing your menu code, and then enable data recording while executing a scripted game sequence that is representative of the CPU-intensive portions of your game.

PGO can be difficult to use on Xbox 360. You may want to visit the Entertainment Developer Forum prior to trying PGO. You may also want to consider PgoLite which gives some of the benefits but is much simpler to use.

Improved Debugging

Debugging multithreaded programs is hard. Debugging a processor with kilobytes (!) of registers is hard. Although features like crash dump support and the @hwthread watch window meta variable can help, the Xbox 360 team is continuing to improve the debugging experience. Planned features include *enhanced thread debugging*. We plan to support features like viewing of multiple threads simultaneously; allowing lock-step debugging and thread-specific conditional breakpoints; providing more thread information; and detecting deadlocks and race conditions.

Compiler Switches

Many standard Visual Studio compiler switches have been removed from the Xbox 360 compiler because they no longer make sense. For instance, all switches related to x86-specific optimizations aren't relevant, so they aren't supported. Xbox 360 only has a single

calling convention, so calling convention options are gone, too. A number of options have been removed from Visual C++ on both Windows and Xbox 360 because they were previously deprecated. The following additional options have been removed from Visual C++ on Xbox 360:

- **/clr** (compile for common language runtime). The CLR doesn't currently exist on Xbox 360.
- **/ML, /MLd** (link with LIBC[D].LIB). The single-threaded CRT libraries are no longer supported, starting with Visual C++ 2005, and are being removed from the product (including the Windows version). Extensive work has been done to optimize the CRT library so that the multithreaded version is nearly as fast as the single-threaded version, so the performance impact is minimal. Add to that the fact that Xbox 360 is a multithreaded platform and we recommend using multiple threads for best performance, LIBC shouldn't be missed.
- **/Oa, /Ow** (assume no aliasing). The Xbox 360 compiler by default assumes that pointer aliasing could occur. Use the **__restrict** keyword to indicate to the compiler that data is not aliased.
- **/Op** (improve floating-point consistency). This compiler switch has been replaced with **/fp** (see below).
- **/YX** (automatically generate precompiled headers). In most cases, the automatic generation of precompiled headers actually decreased compilation speed. **/Yc** (create precompiled header) and **/Yu** (use precompiled header) are still supported, and much more effective at reducing build times.
- **/ZI** (enable Edit and Continue debug info). Edit and Continue is not currently supported.
- **/Wp64** (enable 64-bit porting warnings). 64-bit compatibility is not an issue on Xbox 360.

The following compiler switches were introduced with Visual C++ 2005:

- **/fp** (floating-point model). Choose **/fp:fast** (the default) for speed over accuracy and **/fp:precise** for accuracy over speed. The vast majority, if not all, of Xbox 360 games should prefer speed over accuracy. For instance, using **/fp:fast** with code like "a = b + c * d" typically results in a single fmadd instruction.
- **/openmp** (OpenMP). Choose to enable the OpenMP 2.0 language extension.
- **/analyze**

The following compiler switches are Xbox 360 specific:

- **/Ou** (enable prescheduling). Performs an additional code scheduling pass before the register allocation phase.
- **/Oz** (Enable inline assembly optimization). Reorder inline assembly instructions to minimize latencies. Without this switch all optimization is disabled in functions that contain assembly language.

- **/QXSTALLS** (query Xbox 360 CPU stalls). Dumps timing information for the final Xbox 360 CPU. The compiler includes an integrated CPU pipeline simulator. Turning on this option enables the simulator. It provides final Xbox 360 CPU cycle count estimates for the compiled code, including inline assembly code. This switch must be used in conjunction with **/FAcs** (which outputs an assembly listing .cod file).
- **/QVMXRESERVE**
- **/Oc**
- **/fastcap**
- **/callcap**

The XDK documentation includes details on the current compiler switches and how to use them effectively. For release builds, we recommend using the **/Ox** option, which combines the optimization options proven to be effective for most games.

Common Language Runtime (CLR) and C#

Many Xbox 360 developers have requested the ability to execute C# code on Xbox 360, whether for game scripting or for things like UI where performance is noncritical. The Xbox team is currently evaluating the prospect of providing the common language runtime as an Xbox 360 library. We are also evaluating support of C#, C++/CLI (Common Language Infrastructure), and an Xbox 360-specific subset of the .NET Framework, possibly even including a managed version of DirectX.

Even if we do decide to support the CLR on Xbox 360, there's a lot of work required to make it happen. Because we're primarily driven by game developer feedback, be sure to let your developer relations manager (DRM) know your thoughts on the CLR for Xbox 360.

XML Comments

The Visual C++ compiler supports XML-style comments.

```
/// <summary>
/// These comments can be automatically parsed out using
/// the compiler /doc switch in order to generate code
/// documentation
/// </summary>
```

The Visual C++ IDE uses these comments for IntelliSense purposes. In addition, the new compiler **/doc** switch can be used to extract XML help docs based on these comments. Many third-party tools are also available to extract these comments into useful code documentation. For details on XML comment keywords, see MSDN. The **/doc** switch is not currently supported on Xbox 360. Developers can use the Windows version of the compiler to generate documentation.

New IDE

The Xbox 360 supports the IDE of Visual Studio 2010, which includes the following features:

- IntelliSense is used only for included libraries and namespaces (IntelliSense is more intelligent about what it senses).
- Improved browsing without the need to specify browse files.
- Auto-completion of template types.
- Improved performance (for example, reduced start-up time).
- Multiple-processor builds.
- XML comments for IntelliSense.

We recommend that all development teams transition to Visual Studio 2010, if they've not done so already. As of early 2012, the IDE of Visual Studio 2008 is no longer officially supported for Xbox 360 development. Current XDKs contain libraries, samples, and tools that target only the Visual Studio 2010 environment.

References

- *MSDN Magazine*, May 2004, "Write Faster Code with the Modern Language Features of Visual C++ 2005."
- http://msdn.microsoft.com/visualc/