Best Practices: Dark Secrets for Build Settings and Profiling on Xbox 360

By Tomas Vykruta
Software Development Engineer
Advanced Technology Group (ATG)

Published: June 1, 2009

An improperly set up build configuration for a title can be costly, not only for the developer, but also for the gamer. Countless hours analyzing dozens of Xbox 360 titles—under development and recently shipped—has identified the most common pitfalls that occur with build configurations. These pitfalls occur in four primary areas: CPU profiling, GPU profiling, compile speed, and runtime. Specific issues in each area are listed in order of frequency of occurrence.

Area of concern	Problem
CPU Profiling	1. Using **/callcap** and **/fastcap** profiling methods 2. Not adequately annotating code 3. Not using instrumented, optimized libraries; not using the **/opt:noicf** setting in linker options; not using a link option; and not using PGOLite 4. Enabling debug output 5. Using **assert** 6. Using **sprintf** 7. Using C++ exception handling
GPU Profiling	1. Using vsync 2. Not adequately annotating with PIX events 3. Not sufficiently annotating code 4. Not using debug symbols
Compile Speed	1. Not using precompiled headers 2. Using LTCG at the wrong time or not at all
Runtime	1. Using **printf**

It is not surprising that CPU and GPU profiling top the list. Developers occasionally profile the wrong code—code that behaves differently from a retail build—because of an improper profile configuration.

Have you ever spent hours, or even days, profiling and optimizing a system that was identified as a bottleneck, only to be disappointed that your hard work led to little, if no discernable change in overall performance? The problem was likely an incorrectly set up profile build. It's a problem we encounter continually and consistently when analyzing configurations. It is paramount that a profile build exhibit the same performance characteristics as a release build; otherwise, the wrong systems can be identified as blocks and man-hours can be wasted.

The good news is that you can ameliorate these build configuration problems by implementing the recommendations and best practices identified in this white paper. You can improve performance, generate fewer bugs, reduce compile time, and set up a more effective profile build just by making some simple changes, many of which take mere minutes to apply. In addition, you'll have a build that potentially:

- provides a more stable development environment
- delivers more accurate profiling results
- produces faster compile times
- provides a more optimal runtime

The sections in this white paper are organized to match the identified areas of concern for build configurations.

Build Settings for CPU Profiling

Choose the right profiling method

There are three profiling methods available with the Xbox CPU Profile View (XbPerfView): **callcap**, **fastcap**, and **sampling**. Using the **/callcap** or **/fastcap** profiling method in a build is no longer recommended. This type of instrumentation is designed specifically for XbPerfView instrumented profiling. We now recommend using trace recording and XbPerfView sampling profilers instead. Trace recording does not require any type of instrumentation and provides a much deeper analysis (for details, see "Capturing of a CPU Instruction Trace Using PIX" in the XDK documentation).

Sampling-based profiling with XbPerfView is unobtrusive and accurate, and offers significant advantages over instrumented profiling. Begin your profiling by using **sampling profiler** to isolate hot spots. Follow-up by profiling hot spots in detail using the Performance Investigator for Xbox (PIX) to launch a CPU trace recording.

Compiling with **/callcap** or **/fastcap** instruments your executable by inserting new code, which causes two problems. First, additional code must be executed at the entry and exit points of all functions. Second, the additional code causes small functions that normally would be inlined to compile as not inlined. This, unfortunately, can significantly skew performance results.

Advantages of profiling

Profiling with XbPerfView:

- Allows for the simultaneous profiling of all six threads. This capability is enabled using the Options command on the Tools menu.
- Allows for profiling of any code (PIX event instrumentation is not needed).
- Provides ability to turn on a second event and record L2 cache misses, load-hit-store per function, and so on. This is particularly useful if the PIX system monitor shows a bottleneck. Capturing a second event is only possible with an instrumented XbPerfView profiling method. For more details, see "/callcap – Enable_callcap_profiling" in the XDK documentation.
- Allows for profiling over long periods of time. PIX trace captures are meant for short time slices because significant memory is consumed capturing all instructions and memory accesses.
- Operates as a sampling profiler without the need for **/fastcap** or **/callcap**. While this will not distort performance characteristics, a sampling profile is not inherently

detail-oriented. A sampling capture has the advantage of being performed even when there is insufficient memory for a trace capture.

Characteristics of referenced profiling options are compared next.

Issue	Trace recording	XbPerfView plus CallCap/FastCap	XbPerfView Sampling capture
Performance Impact to Title	No performance impact.	Significant performance impact because of instrumented code. Results will be distorted.	Minimal performance impact.
Code Impact	Profile discrete blocks of code already marked with a PIX event code instrumented with **XTraceStartRecording** and **XTraceStopRecording**, or profile entire frame of rendering thread. Single threaded.	Profile up to six threads simultaneously.	Profile up to six threads simultaneously.
Time Impact	Profile a few frames at most.	Profile for a few hundred frames.	Profile continuously for long periods of time.
Type of Build	Use profile build instrumented with PIX events or capture rendering thread only with any standard build.	Use build with **/callcap** or **/fastcap** instrumentation.	Use any build (including release).
Captures	Detailed capture of every instruction executed, and every memory address read and write.	Accurately records all title-specific function call counts, but ignores system calls. Only some CPU characteristics are recorded.	Sparse sample-based profiling that captures CPU details. Function call counts are not recorded.
Profiler	Integrated with PIX profiler.	Use standalone XbPerfView profiler (ships with XDK).	Use standalone XbPerfView profiler (ships with XDK).

Annotate code with PIX events

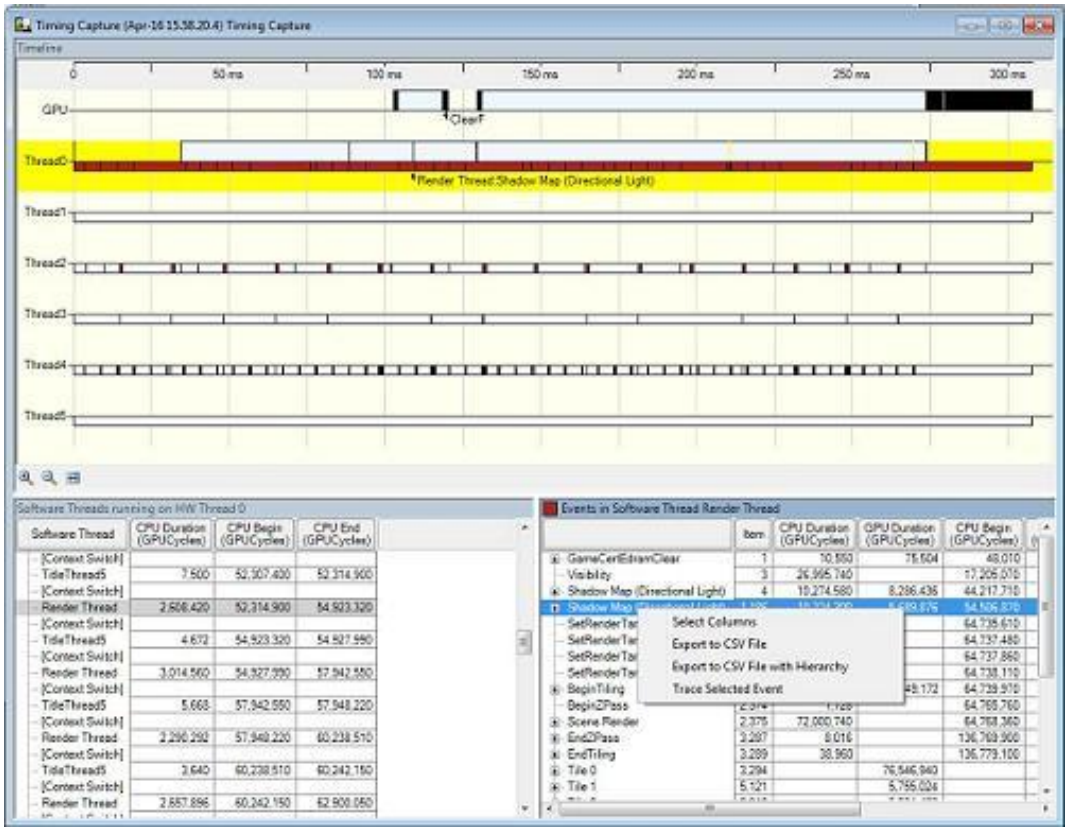
Many titles annotate code with a bottom-up approach, adding instrumentation to specific functions only as needed. Instrumenting from top-down with PIX events by instrumenting the main game loop, the highest-level systems, and all subsystems is

recommended. It's a common misconception that PIX events are intended only for GPU profiling. PIX events are just as powerful for CPU profiling.

Here are some general guidelines for annotating code with PIX events:

- Instrument every subsystem, even systems that do not call GPU functions.
- Instrument the entry function (such as **main**) for each thread.
- Use the **SetThreadName** function to give meaningful names to your hardware worker threads.
- Use a macro that infers the file location, for example, function and line number for PIX event names such as **PIXBeginNamedEvent**(0, __FUNCTION__);

  Allow for one PIX event per draw call. Be very granular, but within reason. Remember that EVERY draw call is already a PIX event, so events are not that expensive. One event per draw call is as granular as we recommend.
- Use **PIXBeginNamedEvent** with annotation creatively because it is a variable argument-style function.
- Analyze PIX events from a PIX CPU timing capture (illustrated below) using PIX CPU trace analysis. This is accomplished by right clicking the PIX event in the Events window. This is the easiest way to get a CPU trace from a nonrendering thread. Internally, this analysis method is used heavily; by default, a PIX CPU trace captures the render thread only. Using the right-click method allows the capture of any arbitrary data wrapped in a PIX event.



Be cautious when using PIX events; they are not completely free. One call costs approximately 0.000125 ms; eight thousand calls add up to approximately 1.0 ms. If your code is heavily instrumented, adding a runtime toggle to enable/disable events in sets is recommended. The expense of a PIX event is incurred only when a PIX capture is active and is negligible during normal execution.

PIX events are still much faster than the **sprintf_s** function because the ASCII conversion and general assembly takes place on the PC. The Xbox 360 console just sends a raw data stream over the network.

Use instrumented, optimized libraries

For profile builds, use the instrumented version of libraries such as d3d9i.lib and xapilibi.lib. For retail builds, revert to the standard libraries or link-time code generation (LTCG) libraries. For information about instrumented libraries, see "Instrumenting for Xbox 360 Performance Tools" in the XDK documentation.

Both XbPerfView sampling profiling and a PIX trace capture can be done on a fully optimized build. It doesn't require instrumented D3D libraries. PIX trace captures are limited to the render thread in this scenario, unless **XTraceStartRecording** and **XTraceStopRecording** are compiled into the code.

If you don't use the instrumented D3D libraries, PIX events do not appear. While PIX captures still function, there's no granularity in the capture, and a flat list of all GPU draw calls is produced.

Use /opt:noicf

Use the **/opt:noicf** setting in linker options for profiling builds, otherwise functions that are bitwise identical are collapsed together. This is great for reducing code size in the release build, but confusing for debugging and profiling.

Ensure the **/opt:icf** and **/opt:ref** settings are turned on for the release build.

Use the Favor Fast Code, not the Favor Small Code optimization setting

To optimize the build, set **Favor Fast Code (/Ot)**, not **Favor Small Code (/Os)** for the Optimization configuration property. This setting is in project configurations under the C/C++ property and the Favor Size or Speed option.

Choosing a setting of **Favor Fast Code (/Ot)** provides two significant gains in the build process:

- More function inlining
- More loop unrolling

Be aware that the executable size increases when compiling with this option. The most effective strategy is to enable Favor Fast Code only for files that contain code—such as per-object and per-draw call functions—that is executed many times per frame. Code that is executed only once per frame or less frequently should continue using Favor Small Code to save memory and to decrease instruction cache thrashing.

Do not use incremental linking

Incremental linking intentionally spaces out functions and bloats the executable. It should be disabled in the profile build. In addition, using incremental linking can cause the compiler to generate extra function stubs for virtual functions and for functors. This would result in a 100 percent mispredict penalty that otherwise would not be present in a standard build.

Use PGOLite, not PGO

PGOLite is a new tool that is frequently confused with Profile-guided optimizations (PGO). PGO is fragile and difficult to maintain on Xbox 360. PGOLite on the other hand is easy to maintain and is guaranteed to reduce instruction cache misses on Xbox 360.

It works by tracking function calls during execution, and rearranging function order during the link phase for improved instruction cache and I-ERAT (which translates the addresses of instructions) utilization. The tool has been used on many titles and on average produces a five percent increase in CPU speed. The tool is easy to maintain and does not require an update with each build. For more information about the tool, see "PGOLite for Xbox 360" in the XDK documentation.

Use SetThreadName

The **SetThreadName** function call allows each thread to have a meaningful name. This is especially important for schedulers that spawn jobs. The named threads will show up in PIX, as well as in the Visual Studio debugger.

Build Settings for GPU Profiling

Do not use vsync

For the purpose of analysis, Microsoft recommends that you temporarily disable vsync by removing the code that binds your rendering frequency to a frame rate. In other words, allow your title to render each frame immediately after the last, without any artificial delays. This allows you to see just how fast your rendering process can run and provides counter information that is more useful. Small improvements in your code can be measured more reliably.

To accomplish this, modify your code so that the swap mode is set to immediate (when **IDirect3D9::CreateDevice** is called, the **PresentationInterval** member of **D3DPRESENT_PARAMETERS** is set to D3DPRESENT_INTERVAL_IMMEDIATE). You can use the PROFILE **#define** to conditionally compile this code.

Ensure you do not use this mode for the retail and release builds.

Generate UPDB debug symbols for shaders

Shader symbols are essential for being able to quickly analyze and improve rendering, especially for engineers unfamiliar with the code. In addition, beginning with the March 2008 XDK, edit-and-continue in the HLSL debugger is supported!

There are three options for generating shader or debug symbols:

Option 1

1. Create a folder named xe:\DumpShaderPDBs.
2. Compile at run time on your Xbox 360.

Option 2

1. Create registry key **HKEY_CURRENT_USER\Software\Microsoft\XenonSDK**.
2. Compile on your PC using the fxc.exe tool.

Option 3

1. Use the D3DXSHADEREX_GENERATE_UPDB flag.
2. Compile on one PC.

A bonus feature is that after a GPU capture is run through PIX analysis and the symbols for the shaders are loaded and the PIX capture file is re-saved, the HLSL symbols are saved in the capture so they can be transferred easily. This is critical when sending captures to Xbox Developer Connection for performance analysis.

Debug build

Because GPU timing depends only upon the contents of the command buffer, the CPU runtime has no affect on GPU performance during a single frame. For this reason, it is fair to use the slow, debug build that is not optimized to do GPU-only PIX captures. There is one exception, however. GPU stalls can be introduced because of slow CPU code that normally would not appear in an optimized build.

Improve Compile Speed

Projects of today are orders of magnitude larger than those of yesterday. This same growth may continue to apply to future code bases. It is important to understand how to optimize compile time to keep your team productive and maintain a fast iteration turnaround time.

Use Precompiled Headers (PCHs)

PCHs are fully supported on Xbox 360, and, when used effectively, can significantly improve build speed. Headers included in your PCH should rarely change, and should be used by multiple source files. Avoid including headers such as resource.h that change frequently.

Use /MP to enable true multiprocessor builds

True multiprocessor compiling is available with Visual Studio 2010. This feature works on a per-file basis, rather than a per-project basis.

Use Link-Time Code Generation (LTCG)

Use LTCG in the CPU profile build, in the retail build, and in the full release build. Be aware that LTCG can substantially increase link time. For that reason, trying both builds is recommended. Link time and performance differences can be measured and a determination made if having two distinct builds (Profile and LTCG Profile). Ideally, do trace captures with LTCG builds. This feature is also sometimes referred to as Whole Program Optimization.

Optimize Build Runtime

Disable debug output

printf and **OutputDebugString** are the most expensive functions seen in profile builds. We recommend these functions be turned off for profiling and that **PIXSetMarker** be used instead. **sprintf** can also be expensive and is sometimes used for profiler display and should be disabled during profiling.

Avoid assert functions

Do not use assert-equivalent functions in a profile or in a release build unless assured they do not affect code generation. Built-in assertions and most user-created assert implementations produce a function call. This means any function using assertions is prevented from being a leaf function and is unlikely to get inlined. This is the case even if the assert is never triggered, or is disabled by a runtime check. Fast asserts are possible using **DebugBreak** and trap intrinsics that are fully pipelined and have virtually no effect on performance.

Use PixSetMarker

sprintf can easily become one of the most expensive functions in a title. **PIXSetMarker** is a great alternative. It's inexpensive because assembly and conversion are done on the PC, not on the Xbox 360 console. It also is time-stamped and thread-attributed.

The cost savings in time achieved by substituting PIXSetMarker for **sprintf** is clearly illustrated next.

Code	Execution time
`sprintf_s(buffer, "Delta t: %.3f\n", value);` `OutputDebugString(buffer);`	95 ms
`PIXSetMarker(0, "Delta t: %.3f\n", value);`	0.133 ms

Turn off C++ exception handling

C++ exception handling is not designed for high performance code and its behavior on Xbox 360 is undefined. Part of the problem is that the compiler needs to allow every function to pass uncaught exceptions through the call stack, which inhibits some optimizations.

As written in a white paper by Bruce Dawson entitled, Xbox 360 CPU: Best Practices, "The C++ standard mandates that **operator new** should indicate failure by throwing an exception. Since this does not work reliably on Xbox 360, you should consider linking with nothrownew.obj to force **operator new** to return NULL if it cannot allocate memory. Otherwise, checking for NULL returns from **operator new** are meaningless, since the standard **operator new** never returns NULL." The nothrownew.obj library ships with the XDK.

Best Practices

The property settings you choose for a build configuration are no less important than the initial conceptualization of your game. Implementing best practices for the configuration activity can help you avoid some common pitfalls in performance and can lead to a successful build.

Use newest version of the XDK

One of the best practices to observe when selecting settings for a build is to use the most current XDK. Use new XDK feature optimizations and new tools to make your title perform better and look sharper.

Use CodeAnalysis and fix warnings

The **/analyze** build configuration conducts static code analysis on C/C++ source code and provides the developer with information about code defects, including buffer overruns, uninitialized memory, null pointer references, and memory leaks. The compiler switch is even more powerful when combined with header Standard Annotation Language (SAL) annotation. For more information about SAL and CodeAnalysis, see "Build Configurations for Xbox 360 in Visual Studio" in the XDK documentation. Additional information about compilers and code can be found in a presentation from Gamefest 2008: SAL, Security, Settings, and the Safe-CRT: Writing Robust Code Made Easy.

Compile PIX events into all builds with exception of retail

PIX is the Xbox 360 game developer's best tool for CPU profiling, and for GPU profiling and debugging. Enabling PIX instrumentation is very simple:

1. Ensure PROFILE is a **#define** directive.
2. Link with d3d9i.lib.
3. Begin using **PIXBeginNamedEvent** and **PIXEndNamedEvent** functions to annotate code.

Use the secure CRT library

The Visual C++ compiler for Xbox 360 provides security-enhanced versions of many C runtime (CRT) library functions. The likelihood of encountering common security errors such as buffer overruns is reduced with use of these functions. Each security error is also a lingering bug. Secure CRT functions are as fast as non-secure counterparts, and in some cases, faster. See Security-Enhanced Versions of CRT Functions on MSDN.

Use warning Level 4

We highly recommend using and maintaining the configuration setting for warning at Level 4 for all projects. Once zero warnings are attained, it is a good opportunity to enable warnings to be thrown as errors to maintain the code in this state. The general philosophy is that fixing a bug by having the compiler output the exact condition and line number during static code compilation is far more economical than subjecting the product to a conventional test cycle to isolate and fix the same bug. A healthy build will run more optimally, will be less strenuous on the test department, and will ship with fewer bugs.

There is another, more methodical approach for achieving a clean build. Start with a setting for the lowest warning level in your build configuration to expose and resolve all suspect constructs. Maintain the configuration setting until all errors at that warning level are fixed. When you have a clean build, move the setting to the next warning level until Level 4 is reached. This progressive use of warning levels ultimately results in fewer errors in the game.

Include symbols in all builds

There is zero run-time cost for building with full symbols, so this is highly recommended. There is a slight compile time penalty, and the size of the executable will increase by ~100 bytes (for the addition of a .pdb file entry).

For more information about best practices, refer to Xbox 360 CPU: Best Practices.