Xbox 360 CPU: Best Practices

By Bruce Dawson
Software Design Engineer
Advanced Technology Group (ATG)

Published: November 8, 2006

Introduction

The Xbox 360 CPU is quite different from the x86 CPUs that are familiar to many game programmers. The design of the instruction set is quite different, and the implementation has some unforgiving quirks. If you program the Xbox 360 CPU as if it is an x86 processor, then you will get poor performance. However, if you adjust your thinking and follow a few simple (and some not so simple) guidelines, then your code will automatically run much faster.

This paper is a brief overview of the issues, solutions, and guidelines. Applying these guidelines in the critical functions of your game will give it substantially improved performance.

Code for Multiple Cores

The Xbox 360 CPU's power is spread across three CPU cores, each having two hardware threads. Getting maximum performance from the Xbox 360 CPU (and from multi-core x86 processors) requires writing multi-threaded code.

Proper design and planning is crucial in order for multi-threaded programming to be successful and manageable. Your multi-threaded design should be clear, simple, and documented, and it should generally require as little interaction between threads as possible.

References

White Paper
    Coding for Multiple Cores on Xbox 360 and Microsoft Windows

Prefer 64-KB Pages

Each Xbox 360 core has a 1,024-entry Translation Look-aside Buffer (TLB) for translating virtual to physical addresses. This buffer, or the high-speed Effective to Real Address Translation (ERAT) units that cache the TLB values, are referenced on every read and write. TLB misses are quite expensive on Xbox 360, so it is important to minimize them, or else you can easily lose approximately 5% of CPU performance.

The Xbox 360 supports 64-KB pages for code, data, stack, virtual, and physical memory. By using 64-KB pages for the vast majority of memory touched by the CPU, it should be fairly easy to keep the number of TLB pages that are referenced per frame well under 1,024. This is because 64 MB of memory is far more than most games reference from the CPU per frame.

CPU trace recording (with PIX or **XTraceStartRecording**) and CPU trace analysis in PIX can be used to find out how many TLB entries you are using. The summary page of a full-frame CPU trace records how many TLB entries of each size were used, and the trace information can be used to track down unwanted 4-KB pages. Using 64-KB pages is as easy as making your stack a multiple of 64-KB and passing the MEM_LARGE_PAGES flag to **VirtualAlloc** and **XPhysicalAlloc**.

Large pages can also be requested on Windows, where they are typically 2-MB or larger. See the **VirtualAlloc** documentation for details.

References

White paper
    Xbox 360 Memory Page Sizes

Memory Latency, Cache Control, and Data Locality

Memory latency on modern CPUs, especially the Xbox 360 CPU, is longer in clock cycles than it has been in the past. In order to avoid wasting too much CPU time waiting on this memory latency, it is important to use cache control instructions, and think about your data locality.

The memory latency on Xbox 360 is approximately 610 cycles, and L1 misses and ERAT (TLB cache) misses cost approximately 40 cycles. This is expensive enough to make it important to minimize the number of cache misses. One important strategy is to use **dcbz128** to pre-zero cache lines that you will be filling completely. This avoids the cost of fetching a cache line from memory when you will be writing to all of it. Another important strategy is to use **dcbt** to prefetch data to hide some of the latency.

Another way to reduce time wasted on cache misses is to reduce the number of cache lines that are used. Many games use only about a third of the data that they bring into the cache, which dramatically increases their cache footprint and their cache miss rate. The number of cache lines used can be reduced by using compressed data structures, and by avoiding having rarely used data bytes, *cold data*, mixed in with frequently used data bytes, *hot data*. Linked lists, arrays of structures that mix CPU and GPU data, and heap-allocation overhead can all cause hot and cold data to share cache lines.

For example, data locality can be improved by using arrays of data instead of lists of data (**std::vector** instead of **std::list**).

Improving your data locality helps not only with cache misses but also helps reduce the number of memory pages used. This helps to avoid TLB misses and reduces the number of ERAT misses.

Finally, when filling in buffers, such as vertex or index buffers, a useful technique can be to write to non-cacheable write-combined memory (specified as such by PAGE_WRITECOMBINE) so that the generated data never uses space in the cache. Just remember to never *read* from non-cacheable memory of any kind.

Improving your data locality improves performance on all processors, not just on Xbox 360.

References

Presentation
    Trace Analysis and Memory Optimization

White paper
    Xbox 360 CPU Caches

Write Efficiently to Write-Combined Memory

In order to efficiently write to memory, it is important that the CPU gather up multiple writes and then send them over the front-side bus (FSB) as a unit. For cacheable memory this happens automatically, and in a forgiving manner. However, when writing to non-cacheable write-combined memory, the rules are very strict. Failing to follow the rules can drop the available write bandwidth by a factor of 50 or more.

The rules are straightforward:

- All writes should be 4 bytes or greater, and naturally aligned
- All writes should immediately follow the preceding write, with no gaps, rearrangements, or overlap

Every time that you break one of these rules, the CPU must send the accumulated data over the FSB and start gathering another batch of data. When writing to write-combined memory, the CPU can send blocks of up to 64 bytes at a time; so, if you break the rules when the CPU has gathered much less data, performance suffers.

If an array of 16-bit indices needs to be written, it is much more efficient to combine pairs of indices into 32-bit values, and then write these.

If you record and analyze a CPU trace using Performance Investigator for Xbox (PIX) then the **Top Issues** tab will point out if your code violates these guidelines too frequently.

Similar rules apply to x86 CPUs when writing to non-cacheable write-combined memory.

References

Presentation
    Xbox 360 CPU Performance Update

Use __restrict To Allow Better Scheduling

Many instructions on the Xbox 360 have fairly high latencies, so it is important to give the compiler the flexibility to rearrange instructions to hide this latency. This can be particularly important when unrolling loops. The **__restrict** keyword tells the compiler that a particular array is not referenced by any other pointers in the same scope, which lets the compiler reschedule the store instructions and other related instructions. Marking a pointer as **__restrict** also tells the compiler that it doesn't need to reload other values after writing through that pointer, which can save many instructions.

As a general rule, try to mark your destination pointers as **__restrict** whenever possible — whenever they genuinely don't point to memory that is referenced through any other variable in that scope.

In the following example, marking *dest* with **__restrict** lets the compiler overlap the two operations in this loop, and cache the value of g_val, the global variable, in a register instead of constantly reloading it:

```
float g_val;
void SumArraysFast( float* __restrict dest, float* src1,
            float* src2, size_t count )
{
    // Assume count is even
    for (size_t i = 0; i < count; i += 2 )
    {
        dest[i] = src1[i] + src2[i] + g_val;
        dest[i + 1] = src1[i + 1] + src2[i + 1] + g_val;
    }
}
```

The **__restrict** keyword is also supported for Windows by the C++ compilers in Microsoft Visual Studio 2010 and Visual Studio 2008, and it can help improve code generation there also.

References

Presentations
    Xbox 360 CPU Performance Update
    Effective Use of the Xbox 360 Compiler

Use VMX128

Each Xbox 360 core can sustain a rate of one floating-point operation, vector or scalar, every clock cycle. Therefore, using the scalar floating-point unit means that 75% of the potential floating-point power is being wasted — vectorizing critical routines can often give a 4× speedup. It is important to know the capabilities of VMX128 and know how to use the instructions, either using the intrinsics in vectorintrinsics.h directly or using the math library in xboxmath.h.

Proper instruction scheduling and efficient loading and storing are crucial to getting good performance from VMX128. If all VMX128 instructions depend on the previous instruction's result, then performance is mediocre. Therefore unrolling loops can be crucial.

Not all VMX128 instructions can reference all 128 registers — this sometimes affects your choice of instructions.

References

Book
AltiVec Technology Programming Environments Manual

White paper
VMX128 New Instructions

Presentations
VMX128: Theory and Practice
Effective use of the Xbox 360 Compiler (lists VMX128 restrictions)

Use __declspec(passinreg)

Normally, structure parameters are passed in memory or in the integer registers. This causes inefficiencies when wrapping a **__vector4** in a structure, as it prevents the **__vector4** from being passed in a VMX register. Starting with the Oct 2006 XDK, it is possible to mark any structure that contains a single built-in type (**int**, **float**, **__vector4**, etc.) as being passed and returned in the appropriate type of register. This allows operator overloading and function overloading based on custom vector types. Typical usage looks like this:

```
__declspec(passinreg) struct MyVec
{
    __vector4 vec;
    // Member functions go here
};
```

Initialize __vector4 Constants and Variables Efficiently

Initializing **__vector4** constants is fundamentally different from initializing floating-point or integer constants. If not done properly, it can be very inefficient. With **float** or **double** constants, the compiler stores the value in the data segment and then loads it. For integer constants, the compiler generates a few instructions to build the constant from immediate values. However, for **__vector4** values, the compiler initializes it like a structure or an array. That means that it copies individual items into an in-memory representation of the **__vector4**, which can then be loaded into a register. This can require over a dozen instructions, and it causes an expensive load-hit-store penalty.

To force the compiler to put the **__vector4** data into the data segment, you need to mark your **__vector4** constants as static constants, like the following:

```
static const __vector4 usefulValues = { 0, 0.5, 1, 2 };
```

By adding the **static** keyword, the compiler knows to place the data into the data segment so that it can be loaded with 3 instructions, instead of regenerating the constant each time that the code is executed. By marking it as a constant, you avoid accidentally and permanently changing the values.

Loading and storing of VMX data can also take many instructions, particularly when loading or storing misaligned or partial vectors. It may be worth unwinding loops so that you can, for instance, load three aligned **__vector4** values, and then process them as four 3-float items.

For vector constants that are frequently used throughout your code, consider using the **/QVMXRESERVE** command line-switch and the **__VMXSetReg** and **__VMXGetReg** intrinsics to permanently store these values in a reserved set of registers.

Prefer 32- and 64-bit Local Variables and Parameters

The PowerPC instruction set has a limited number of instructions for working on 8-bit and 16-bit values. They can be loaded and stored, but all mathematical and logical operations must be performed on 32-bit or 64-bit values. Therefore, if you use 8-bit or 16-bit function parameters or local variables, you force the compiler to insert many sign-extending or clearing instructions, such as **extsh** or **clrlwi**, to give you the requested semantics. Using local variables and parameters that are 32- or 64-bit saves instructions and improves performance.

References

Presentations
- Introduction to PowerPC and the Xbox 360 CPU
- Xbox 360 CPU Performance Update

Prefer Small Data Types in Memory

When allocating arrays of data, the most important thing is to conserve memory, cache space, and memory bandwidth. Therefore, you should use the smallest type possible — such as **char**, **short**, **float-16**, and bit-fields — even if it takes a few extra instructions to load or store it. The **float-16** type can be packed and unpacked using **vpkd3d128** and **vupkd3d128**, and is compatible with the GPU.

References

White paper
VMX128 New Instructions

FastCPU sample

Be Careful with Booleans

The Xbox 360 compiler sometimes has difficulty generating efficient code when using Boolean expressions and variables (**boo**l or **BOOL**).

Complex logical expressions, such as the following, require multiple branches, which prevent the compiler from doing efficient scheduling:

```
if ((x && y) || w)
```

Rewriting these expressions using logical operations may lead to more efficient code; this is especially true with the condition operator (a ? b : c) if both values to the right of the question mark are integer constants.

The compiler also has difficulty generating efficient code for functions that return a Boolean value, even when these functions are inline. One particular instance of this is when doing comparisons with STL iterators. If the STL iterators were just raw pointers, then iterator comparisons would be just two instructions. Instead, they typically expand to seven instructions, even though the comparison function is fully inlined.

In the case of the STL, the best solution is to use array indexing and an integer loop variable to generate more efficient code. In general, try to avoid calling functions that return **bool** inside of critical loops.

References

Presentation
Xbox 360 CPU Performance Update

Avoid Shifts by Variable Amounts

The Xbox 360 CPU's integer units can execute most integer instructions, including **rlwinm** and **cntlzw**, with a throughput of 1 per cycle and a latency of 2 cycles. However, some instructions are microcoded and take much longer. The most unexpected of these instructions is the family of instructions that shift by a variable amount. These are **sraw**, **srad**, **slw**, **sld**, **srw**, **srd**, **rldc**l, **rldcr**, and **rlwnm**. The shift instructions that have a compile-time shift amount have a latency of 2 cycles, but the variable shift instructions have a latency of about 16 cycles, and a throughput of 1 every 16 cycles. Some algorithms require variable shifts, but they should be avoided in your critical loops, if possible. Often a variable shift can be replaced by shifting left by one position on each iteration of a loop, or even, in some cases, by looking up the value in a table.

In the following code sample, the **SlowLoop** function shifts by a variable amount on every iteration, but the **FastLoop** function does not.

```
void SlowLoop(unsigned mask) {
    for (unsigned i = 0; i < 32; ++i) {
        if ((1 << i) & mask)
            DoSomething(1 << i);
    }
}

void FastLoop(unsigned mask) {
    unsigned flag = 1;
    for (unsigned i = 0; i < 32; ++i) {
        if (flag & mask)
            DoSomething(flag);
        flag <<= 1;
    }
}
```

Avoid fcmp and vcmp

The **fcmp** instruction, like all the scalar floating-point instructions, has a throughput of 1 per cycle and a latency of 10 cycles. However, the usual purpose of **fcmp** is to control a conditional branch. Because the branch pipeline is so much shorter than the scalar floating-point pipeline, the result of the comparison is rarely available in time for the branch that follows. The CPU detects this and flushes the most recent instructions from the pipelines, and then reissues them. It may have to do this several times before the **fcmp** result is available.

In many cases **fcmp** can be avoided by using **fsel** instead. For instance, the floating-point versions of **fmin** and **fmax** from ppcintrinsics.h, shown in the following code, are branchless, avoid pipeline flushes, and also give the compiler more scheduling flexibility:

```
#define fpmax(a,b) __fsel((a)-(b), a,b)
#define fpmin(a,b) __fsel((a)-(b), b,a)
```

There is sometimes a temptation to use **fsel** to implement floating-point absolute value. However, **fabs** does the job better.

Often floating-point branches are used to detect special cases so that calculations can exit early. On Xbox 360, this often hurts performance, since the cost of the **fcmp** may be greater than the average time saved.

Sometimes, floating-point branches are used to choose which calculation should be used. In many cases it is actually faster to calculate both results and then use **fsel** to select the appropriate version. If the compiler can interleave the calculations, then the calculation cost may be about the same as doing just one of the calculations, and the cost of the flush caused by **fcmp** is avoided.

Similar guidelines exist for the various **vcmp** instructions. Instead of using these to control branches, use the bit-mask that **vcmp** creates, together with bit-operations, to select the appropriate results.

References

White paper
    A Detailed Examination of the Xbox 360 CPU Pipelines

Avoid bctr

The **bctr** or *branch-to-counter-register* instruction is used to implement calls through function pointers, including virtual functions, thunking between DLLs, and some switch statements. The **bctr** instruction usually implies a branch misprediction, because the CPU's branch predictor generally cannot predict the target address when **bctr** is used. Also, branches through function pointers, including virtual function calls, often imply missed optimizations, because the target function cannot be inlined.

The **bctr** instruction is the most efficient way to implement the tasks described in the previous paragraph; however it can be a waste of performance if used unnecessarily in an inner loop. If **bctr** is used frequently — as reported by trace analysis —check to see whether it can be replaced with a compile-time determination of where to branch.

References

White paper
    A Detailed Examination of the Xbox 360 CPU Pipelines

Avoid Other Expensive Instructions

The Xbox 360 CPU has other instructions that are expensive. These include the instructions for integer multiplication and division, floating-point division, and calculating square roots. None of these instructions are pipelined, which means they have high latency and low throughput (no other instructions can be issued until they complete), and they stall both hardware threads on the core.

It is not realistic, or even necessary, to completely avoid using these expensive instructions. However it is important to minimize use of expensive instructions, and to keep them out of critical loops. The **Top Issues** tab in PIX's trace analysis view can be a vital tool for discovering when these instructions are used too frequently.

Avoid Load-Hit-Stores

A load-hit-store occurs when a thread stores data to a memory location and then — shortly afterward — reloads from that same location. A load-hit-store can also happen when the loading address is separated from the storage address by a multiple of 4 KB, but this type of load-hit-store is less common. Load-hit-store events can be very expensive — 40-80 cycles — which makes minimizing them crucial.

There are several possible causes of a load-hit-store, and even more possible solutions. Three of these are transferring data between register sets, updating data in arrays, and passing structures by value.

Transferring Data between Register Sets

Transferring data between the register sets (integer, scalar floating point, and vector) can only be done by going through memory. This unavoidably leads to a load-hit-store penalty. This often happens when converting a floating-point value to an integer, or when assigning the results of scalar math to a vector. The ideal solution to this is to keep data in one register set. The vector registers in particular can do both integer and floating-point math.

If sticking with a single register set is not possible, then it may be possible to unwind a loop to minimize the cost. If values are stored four times and then loaded four times, only one load-hit-store penalty is paid. Using the **__frnd**, **__fctid**, and **__fcfid** intrinsics can sometimes effect the desired conversion of floating-point to integer without requiring a transfer of the results to the integer registers. Assigning data directly to an element of a vector may result in a load-hit-store, and **lvlx** and **vrlimi** may be useful for avoiding this. Sometimes a table-lookup can be used to transfer data from the integer registers to the floating-point or vector registers. This technique only works if the table size is fairly small.

Updating Data in Arrays

Arrays of data need to be kept in memory, and they can lead to a load-hit-store event in many ways. Updating a histogram may require frequent read-modify-write sequences to the same element. Also, if you write data through a pointer that is not marked as **__restrict**, then the compiler assumes that other arrays and global variables are potentially being overwritten by writing to the array, which may force the compiler to flush their values to memory and then reload them later. Using **__restrict** on write targets is crucial for telling the compiler that it doesn't need to flush values to memory. Make sure that this is actually the case, and try to avoid the use of global variables.

Passing Structures by Value

Function calls that pass structures by value may have to pass the structures in memory. The calling function writes the structure to the stack, and the called function reads it back, causing a load-hit-store. Passing structures by reference or by pointer, passing built-in types instead of structures, or using **__declspec(passinreg)** to specify that the compiler pass the declared type by using registers can all help minimize this problem. The same issue can occur with returning values, and similar solutions apply. Operator overloading can lead to many temporary values being returned, and the compiler may find it difficult to avoid spilling some of these temporaries to the stack. So when writing critical functions, it may be necessary to avoid operator overloading, and to use VMX intrinsics directly to avoid load-hit-stores.

References

White paper
A Detailed Examination of the Xbox 360 CPU Pipelines

Tool
CPU Pipeline Animator

Avoid Excessive Stack Traffic

The Xbox 360 CPU has more than 192 registers per hardware thread, and it is capable of doing significant computations without running out of registers. Ideally the stack should be used just for return addresses and preserving registers in the function prologue. If your code is devoting significant instructions to reading and writing stack data in the middle of a function, then these instructions are probably avoidable, and they are probably causing load-hit-store events and other performance slowdowns. The solutions are typically the same as those for avoiding load-hit-stores. Look for references to r1, the stack pointer, to see if your code is spending much of its time reading and writing stack memory.

Prefer Native Alignment

Misaligned values — values whose address is not a multiple of their size — have multiple disadvantages on Xbox 360. Integer values that aren't naturally aligned cost about an extra 50 cycles to load and store whenever they cross a 32-byte boundary, and they cause alignment exceptions when used with non-cacheable memory. Floating-point values cause alignment exceptions any time that they are not 32-bit aligned.

Native alignment is also needed for **Interlocked** functions, and native alignment is required if loading and storing need to be atomic operations.

With VMX data, the cost of misalignment is the need for extra instructions to load and store the data. With careful coding — perhaps batch loading and unloading of data to reduce the number of load-store instructions — this cost can be minimized.

Naturally aligned data is also more efficient on x86 processors.

References

White paper
    Xbox 360 CPU Data Alignment

Don’t Enable Floating-Point Exceptions

The floating-point units on CPUs can be configured to trigger exceptions — crashing your program — when various unusual conditions occur. These can include floating-point division by zero, subtracting infinity from infinity, and others. Enabling these floating-point exceptions — with the **_controlfp** function — can be a valuable debugging tool. However, the performance of the Xbox 360 CPU drops precipitously — by a factor or five or more — when floating-point exceptions are enabled. This slowdown occurs even if no exceptions are triggered.

Link with nothrownew.obj and Don’t Throw C++ Exceptions

C++ exception handling is not supported reliably by the Xbox 360 compiler because of the complexity of handling all cases and because of our belief that C++ exceptions are not compatible with high-performance game code. Compiling with C++ exceptions enabled on Xbox 360 generally has a modest performance cost, because exception-handling code and data are stored separately from the main line code. However, *throwing* exceptions can be very expensive, and may lead to data corruption or unpredictable crashes.

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.

Standard Template Library

The Standard Template Library (STL) handles failures by using exceptions. There is a **#define** named _HAS_EXCEPTIONS which can be set to zero to tell the STL implementation not to use exceptions, and on Xbox 360, this is automatically set to zero when C++ exceptions are disabled. It is crucial to understand that with C++ exceptions disabled, there is no way for the STL to tell if an allocation has failed. If there isn’t enough memory to add a new item to a container, then the game simply crashes. Therefore, when using the STL with exceptions disabled, it is crucial to make sure that you won’t run out of memory.

The versions of STL that is provided with Visual Studio 2010 and Visual Studio 2008 have special debug iterators that do detailed checking for illegal iterator usage, designed for debug builds. The debug iterators default to *on* in debug builds. They can be disabled by setting the **#define** _HAS_ITERATOR_DEBUGGING to zero. However, they should generally be left enabled in debug builds to help find bugs in your code, and they won't slow your code substantially.

This version of STL also has special checked iterators. These iterators do less detailed checks of your iterators, designed for your final release builds. If you find that the extra checks are hurting performance, disable the checked iterators by using **#define** to set _SECURE_SCL to zero.

References

MSDN
    Standard Template Library

Use the Compiler Effectively

Proper use of the Xbox 360 compiler is crucial to get the best performance possible, and to help you follow the other guidelines given here. The Xbox 360 can generate excellent code, if you help it.

Many Xbox 360 features — including VMX128 instructions, cache control instructions, and special floating-point instructions — are directly accessible through compiler intrinsics. Using compiler intrinsics is much more efficient than using inline assembly language, because it gives the compiler the information it needs to continue allocating registers and scheduling code. Inline assembly generally disables or inhibits the optimizer in functions that use it. The intrinsics are prototyped in ppcintrinsics.h and in vectorintrinsics.h.

Link Time Code Generation (LTCG), also known as Whole Program Optimization, is an option that lets the compiler do all code generation for your program at link time. This gives the compiler access to all of the code at once which allows for cross-module inlining, and other aggressive optimizations. LTCG slows down your link times enough to make it inappropriate for day-to-day work, but you should plan to ship your final release builds using LTCG.

The Xbox 360 compiler can optimize your code for time (speed) or for size. If optimizing for time, then the compiler avoids using microcoded instructions because they execute slowly. If optimizing for size, then the compiler uses microcoded instructions, when appropriate, to reduce code size and instruction cache misses. Careful use of these options can give you the benefit of small and fast code.

Profile Guided Optimization (PGO) is a technique that lets the compiler optimize your code based on your typical execution patterns. With PGO, you run a specially instrumented version of your game that records data about which code paths were taken, and then you re-link based on this data. PGO can automatically decide which code should be optimized for time and which should be optimized for size, as well as moving cold code away from hot code, to make better use of the instruction cache. Using PGO in games can be difficult because of the challenge of doing automated profiling runs of game-representative code.

The PowerPC integer division instruction does not trigger an exception on division by zero — instead it gives undefined results. It is the compiler's responsibility to insert extra instructions to check for division by zero. These checks should be left in during development, but they can be removed for your final release build. The check can be removed by selecting **Trap Integer Divides Optimization** on the Xbox 360 compiler page.

Branches are moderately expensive on Xbox 360, and they form scheduling barriers that make it harder for the compiler to hide instruction latency. Using **__fsel** and **__vsel** to avoid branches can help the compiler generate better code. Using the conditional operator can also help:

```
(a ? b : c)
```

If b and c are integer constants, then in many cases, the compiler automatically generates branchless code. In addition the **_CountLeadingZeros** intrinsic can be used to implement many branchless integer operations.

It takes practice to learn how to work well with the compiler, and it is important to examine the results in order to learn what works well, and what doesn't. The assembly language generated can be viewed in the debugger or in the source window when doing PIX trace analysis. Alternately, the assembly language can be viewed without even linking or running the program by generating a file with the extension .cod. A .cod file can be generated by going to the C/C++ section in the file or project properties, to the **Output Files** tab, and setting **Assembler Output** to **Assembly, Machine Code and Source (/FAcs)**.

References

Presentations
- Effective use of the Xbox 360 Compiler
- Under The Hood: Link-time Code Generation

White paper
- Profile-Guided Optimizations

Use the Available Profilers

Following all of the CPU best practices can be time-consuming, so it is important to focus your work on the places where it will have the greatest impact. It is important to use the available CPU profiling tools to focus your efforts. To get started, use the following tools, which are listed in the order that you should typically use them:

PIX System Monitor
: Use the PIX **System Monitor** to find stalls and idle time. If your main thread is spending half of its time idle or blocked on D3D, then optimizing CPU performance will not help.

PIX Record Timing
: Use PIX timing captures to visualize high-level thread activity. **Record Timing**

can show you how threads are swapped in and out and can also display timing information for D3D rendering events. Timing captures can also display timing information for user-defined events if you insert markers with **PIXBeginNamedEvent** and **PIXEndNamedEvent**. These markers can be used on non-rendering threads also, to delimit areas such as physics, AI, etc.

XbPerView

Use XbPerfView to find which functions are using the most time. The XbPerfView instrumented profiler — either with **/callcap** or **/fastcap** — can tell you how much time is spent in different functions, as well as how many times each function is called. The XbPerfView sampling profiler works on non-instrumented builds and can tell you how much time is spent in different functions, without any distortions from instrumentation.

Trace Recordings

Trace recording, either triggered with **XTraceStartRecording** or with the **Record CPU Trace** option in PIX, can be used to detect violations of many guidelines. Trace analysis in PIX can tell you how efficiently you are using the caches and the TLBs, and it can point out places where you violate many of the guidelines too frequently. Trace recording is best done on your final or near-release builds, to avoid distortion from instrumentation.

Performance Monitor Counters

The performance monitor counters, available through functions such as **PMCInstallSetup**, are the most accurate way to determine what factors are wasting CPU performance. The performance monitor counters are best used on your final or near-release builds.

References

Presentation

Effective Profiler Use on Xbox 360

Summary

The Xbox 360 CPU has impressive peak performance. With a bit of practice and a few changes to your coding style, you can make use of more of this performance, and make your game run faster and look better.