Xbox 360 CPU Data Alignment

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

Published: August 26, 2005
Updated: October 4, 2005

Introduction

Data on Xbox can be arbitrarily aligned. The processor will read and write this data without significant slowdown and without any alignment interrupts. Most RISC processors are not so forgiving, however, and the Xbox 360 CPU is no different.

The Xbox 360 CPU and compiler work best if all data is naturally aligned—that is, if each variable's address is a multiple of its size. To maximize performance and avoid alignment interrupts, it's best to keep data aligned on Xbox 360.

CPU Misalignment Behavior

How the Xbox 360 CPU deals with misaligned data depends on the type of data—integer, scalar floating-point, or VMX128.

Integers

The Xbox 360 CPU is most forgiving about alignment when it is handling integers, including **short**, **int**, and **__int64** (**char** can never be misaligned). When dealing with cacheable memory, it always handles integer loads and stores without an alignment interrupt, regardless of their alignment. However, in some cases there is a performance penalty.

If an integer load or store crosses a 32-byte boundary, the load or store will initially fail and the CPU will retry the load or store using microcode. Resorting to microcode is expensive—it will add forty to sixty cycles to the load or store time. This penalty can be hard to detect, since the only symptoms will be decreased performance, and the performance counters reporting a large number of microcoded instructions. You can use trace recording (**XTraceStartRecording** and **XTraceStopRecording**) and the trace dump analysis tool to detect misaligned loads and stores that are wasting performance.

The Xbox 360 CPU is less forgiving with unaligned loads and stores when you have non-cacheable memory, whether allocated with PAGE_NOCACHE or PAGE_WRITECOMBINE. Unaligned loads and stores to non-cacheable memory cause an alignment exception crash, whether they cross a 32-byte boundary or not.

The rule with integers is simple. Keep them aligned whenever possible, unless there are substantial memory savings to misalignment and the data is not accessed in performance-critical code. Unless you do something to change them, your integers will be aligned.

Float and Double

The Xbox 360 CPU is less forgiving about alignment when dealing with floating-point numbers. If a float or double is not **DWORD** aligned, any loads or stores will cause an alignment interrupt and your game will crash.

When loading and storing doubles on the Xbox 360 CPU, it is possible to cross a 32-byte boundary without causing an alignment interrupt. This occurs if the double is **DWORD** aligned but not eight-byte aligned. In this case, the load or store will go to microcode, adding forty to sixty cycles to the load or store time.

VMX128

The rules are different again for VMX128 loads and stores. VMX128 loads and stores, using **__lvx** and **__stvx**, always read from 16-byte–aligned addresses. The bottom four address bits are ignored, which means alignment interrupts cannot happen with VMX128.

With regular VMX, programmers deal with misaligned data by using **__lvsl**. This intrinsic takes an arbitrarily aligned address and creates a permute vector. By loading from the desired address and the desired address plus 15, and then permuting the two loaded values together with the results of **__lvsl**, an arbitrarily aligned 16-byte value can be loaded. The second address is 15 greater than the start address so that the next 16-byte block will be loaded when necessary, while not going far enough to cause an access violation when near the end of a memory page.

On the final Xbox 360 CPU with VMX128, there is a new option. The **__lvlx**, **__lvrx**, **__stvlx**, and **__stvrx** intrinsics can be used to load and store from arbitrarily aligned addresses. The catch is that they will load and store data only within an aligned 16-byte block. Thus, a misaligned load of 16 bytes requires an **__lvlx** from the desired address, an **__lvrx** from the desired address plus 16, and then using a bitwise OR to combine the results. The second address is 16 greater than the start address so that the second load will be properly aligned in the register. If the second address is on a 16-byte boundary, and thus unnecessary and potentially on a new memory page, no memory access occurs.

The chart below shows a block of memory and the data loaded from it by **__lvx**, **__lvlx**, and **__lvrx**.

Memory, starting at address 0:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	…

__lvx from address 5:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

__lvlx from address 5:

5	6	7	8	9	10	11	12	13	14	15	0	0	0	0	0

__lvrx from address 21 (5 + 16):

0	0	0	0	0	0	0	0	0	0	0	16	17	18	19	20

__lvrx from address 16:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Misaligned VMX128 data can be written with **__stvlx**, **__stvrx**, or with the store vector element intrinsics.

The VMX128 instructions for misaligned loads and stores—**__lvlx**, **__lvrx**, **__stvlx**, and **__stvrx**—cannot be used on non-cacheable memory, whether allocated with PAGE_NOCACHE or PAGE_WRITECOMBINE. Using these instructions on non-cacheable memory cause an alignment exception crash.

Misaligned vector data is sometimes necessary to conserve space. Supporting misaligned vector data requires just a few extra instructions to load and store data. The performance cost is minimal—it is simply the time to execute the additional instructions.

Alignment Handling in the Compiler

The compiler natively aligns local variables, global variables, and structure members to their native size. That is, a 4-byte **int** will naturally be 4-byte aligned, an 8-byte **double** or **__int64** will be 8-byte aligned, and a 16-byte **__vector4** will be 16-byte aligned, and so on. This ensures that the CPU can safely and efficiently load and store these values.

By default, memory allocations on Xbox 360 return 16-byte–aligned memory, so these alignments will naturally be maintained even on allocated objects and arrays.

You can override this behavior in structure packing by using **#pragma pack**. This pragma, which is also available on Xbox and Windows, lets you specify the maximum alignment that you want the compiler to use. A pack value of 1 means "insert no padding"; a pack value of 2 means "align variables to 2-byte boundaries when needed, but no more," and so on.

With the default packing settings, the **x** variable in the structure below will be stored at an offset of 4, and the size of the structure will be 8. With **#pragma pack(1)** or **#pragma pack(2)** the **x** variable will be stored at an offset of 2, and the size of the structure will be 6, but access to the **x** variable will be inefficient.

```
struct Widget
{
    char a;
    char b;
    float x;
};
```

With the default pack settings, the **Widget** structure requires 4-byte alignment, because it contains a float. Therefore, the **myWidget** variable in the structure below will be stored at an offset of 4, and the size of the structure will be 12.

```
struct WidgetPlus
{
    char c;
    Widget myWidget;
};
```

With the default pack settings the **WidgetPlusV2** structure below takes up just 8 bytes. However, its layout is not compatible with the **Widget** structure, which is why the compiler can't automatically apply this optimization. The compiler needs to ensure that all **Widget** objects are laid out the same way in memory.

```
struct WidgetPlusV2
{
    char c;
    char a;
    char b;
    float x;
};
```

Windows programmers will sometimes specify **#pragma pack(1)** to make their structures smaller. However, on Xbox 360 this option should be used cautiously, if at all. Once alignment is disabled, integer values will sometimes load and store more slowly, floating-

point values may cause alignment interrupts, and VMX128 data will require writing extra code to load and store it.

When you create a structure using **#pragma pack(1)**, the compiler knows the elements are misaligned. For floating-point variables, it will create extra instructions to load and store the data so as to avoid the alignment interrupts. The performance will be terrible, but it will work. The compiler does not attempt to automatically deal with misaligned VMX128 data, and it does not generate extra instructions for integer data, since the CPU can handle this case by itself.

When dereferencing a pointer, the compiler assumes that it points to naturally aligned data. If it is a misaligned pointer to floating-point data, this assumption may result in a crash. Therefore, the compiler supports the **__unaligned** keyword, which tells the compiler a pointer may not be naturally aligned. For integer types, this is ignored because the CPU can deal with the misalignment. For VMX128 types, this is ignored because it is the programmer's responsibility to deal with misaligned VMX128 data. However, for floating-point data the **__unaligned** keyword tells the compiler to generate the extra instructions necessary to load or store safely from the pointer. For instance, the following function returns the **float** value pointed to by ptr.

```
float AlignmentTest(float* __unaligned ptr)
{
    return *ptr;
}
```

Without the **__unaligned** keyword, this function is two instructions—one to load the float value, the other to return. With the **__unaligned** keyword, the compiler does an integer load to deal with the potential misalignment, an integer store to an aligned temporary variable, then a float load (causing a load-hit-store penalty), before returning. The code is twice as big, the performance is much worse, but it works correctly without crashing.

Knowing when to use the **__unaligned** keyword and when to avoid it is important. The compiler tries to help by warning you when you take the address of unaligned data and assign it to a pointer that is not marked as **__unaligned**. In the following example, the **BadPacking** structure is packed without respecting the alignment needs of its member variables. When the address of **y** is passed to **AlignmentTest**, the compiler warns that you may have a problem. If you ignore this warning, you may encounter alignment interrupts.

```
float AlignmentTest(float* ptr)
{
    return *ptr;
}

#pragma pack(push, 1)
struct BadPacking
{
    char  x;
    float y;
};
#pragma pack(pop)

BadPacking g_badPacking;

float AlignmentTrouble()
{
    // The following line of code triggers this warning:
    // C4366: The result of the unary '&' operator may be unaligned
```

```
    return AlignmentTest(&g_badPacking.y);
}
```

When creating structures, try to leave the compiler's alignment settings unchanged. To avoid wasting space, carefully plan how you pack your data. A good approach is to start with your large variables, such as **__vector4** types, and then move on to progressively smaller types. This simple rule ensures a minimum of padding is needed to ensure alignment.

Note that the size of structures is also padded, to a multiple of the largest alignment requirement in the structure. For example, a structure that contains a **double** and a **char** will have a size of 16, regardless of the order of elements. This is necessary so that in an array of structures each structure will be aligned properly.

To make more efficient use of space in structures, use bit fields. The Xbox 360 CPU has instructions for efficiently inserting and extracting bit field data, and the compiler will use these when dealing with bit fields.

The C++ compiler has two warnings that can be useful when checking for structure padding:

- Warning C4324 is triggered if padding was added to a **class** or **struct** because of **__declspec(align())**. This warning is enabled under warning level four.
- Warning C4820 is triggered any time padding is added to a structure. It prints a warning saying how many bytes of padding were added and where. This warning is normally disabled, but you can enable it with **#pragma warning**.

The following code demonstrates the alignment warnings:

```
// Enable both alignment warnings for warning level three
#pragma warning( 3 : 4324 )
#pragma warning( 3 : 4820 )

__declspec(align(16))
struct S {
    int i;
    short s;
    int j;
    // No warnings for bitfields with holes in them.
    int k: 3;
    int l;
};
```

The preceding code generates the following warnings:

```
warning C4820: 'S' : '2' bytes padding added after data member 'S::s'
warning C4324: 'S' : structure was padded due to __declspec(align())
warning C4820: 'S' : '12' bytes padding added after data member 'S::l'
```

Summary of Alignment Advantages

The main advantages to keeping your data naturally aligned on Xbox 360 are avoiding alignment interrupts, avoiding code bloat, and avoiding performance problems from additional microcode. However, there are some other less obvious advantages.

You can use the **__lvlx** intrinsic of VMX128 to load 1-, 2-, 4-, or 8-byte items into VMX128 registers in a single instruction. However, this works only if the items don't cross a 16-byte boundary. Naturally aligned items are guaranteed not to cross 16-byte boundaries.

The simplest way to store an 8-bit, 16-bit, or 32-bit value from a VMX128 register is to use **__stvebx**, **__stvehx**, or **__stvewx**. These intrinsics work only if the destination is natively aligned.

A load-hit-store occurs when you store to an address and then immediately load from the same address, or from an address that aliases to the same address. Misaligned loads and stores alias to more addresses, thus increasing the chances of spurious load-hit-store penalties. If you walk a misaligned array of integers, incrementing every integer, you may get an expensive load-hit-store on every read. If the integers are natively aligned, however, you won't get any load-hit-stores.

Misaligned data may end up crossing a cache line boundary, causing two cache misses for a single item.

The GPU requires all of its data to be at least **DWORD** aligned.

Recommendations

- Keep integer data on Xbox 360 naturally aligned, unless the memory savings outweighs the substantial performance cost.
- Keep scalar floating-point data on Xbox 360 naturally aligned.
- Vector data may be stored misaligned but extra instructions will be needed to load and store the data. If vector data is not 16-byte aligned, it should generally at least be 4-byte aligned.