Xbox 360 Memory Functions

Bruce Dawson
Software Design Engineer
Advanced Technology Group (ATG)

Published: October 17, 2005
Updated: April 8, 2008

Introduction

In an ideal world, when you needed to copy or fill memory you would always use **memcpy** or **memset**, and you would always get maximum performance. However, this ideal world is not practical. Memory copying and filling routines have to handle all types of memory, all possible alignments, and all copy and fill sizes. This generality means that sometimes other memory routines, with some restrictions, can have better performance.

Xbox 360 has four memory copying routines and three memory filling routines. These routines are **memcpy**, **XMemCpy**, **XMemCpy128**, **XMemCpyStreaming**, **memset**, **XMemSet**, and **XMemSet128**. Once you understand the purposes of these routines, you will be able to select the correct one, and you will know when copying memory manually is the best choice.

Factors That Affect Performance

The type of memory, the alignment of the source and destination buffers, and other factors can have a significant effect on memory copying performance and on which memory functions can be used. Understanding these factors is important in using the memory functions effectively.

Cacheable and Non-Cacheable Memory

Memory on Xbox 360 can be either cacheable (the default) or non-cacheable (allocated with PAGE_NOCACHE or PAGE_WRITECOMBINE). Non-cacheable memory, typically allocated with PAGE_WRITECOMBINE, is most often used when writing to various graphics resources. There are some instructions that cannot be used with non-cacheable memory.

The **dcbz128** instruction — available through the **__dcbz128** intrinsic — clears out the destination cache line without fetching it from memory. When copying large amounts of data, using this instruction reduces the memory bandwidth used by one third. When filling memory it improves performance by far more than that. However, because **dcbz128** operates on the cache, it cannot be used on non-cacheable memory. Using **dcbz128** on a non-cacheable destination causes an alignment exception.

The **lvlx**, **lvrx**, **stvrx**, and **stvlx** instructions are used for misaligned VMX loads and stores. They are very convenient for reading misaligned source data, and for copying the straggler bytes at the beginning and end of a buffer. However, these instructions are not supported for non-cacheable memory. Using these instructions on non-cacheable memory causes an alignment exception.

The XMem routines, aside from **XMemCpyStreaming**, assume that both the source and destination buffer are in cacheable memory. Thus, they can use all of these instructions, which give them a performance advantage over **memcpy** and **memset**. However, if you are copying to non-cacheable memory, you cannot use these routines — you must use **memset** and **memcpy** or **XMemCpyStreaming**.

Summary: Use **memset** and **memcpy** or **XMemCpyStreaming** on non-cacheable memory; use the XMem routines otherwise. Note that in normal cases, you should never be copying from non-cacheable memory — only copying into it. If you are copying from non-cacheable memory, consider redesigning to avoid this, as there is a substantial cost in doing so.

Destination Alignment

One of the biggest factors affecting memory copying performance is data alignment. If the destination is not fully aligned (alignment on 4-byte, 8-byte, 16-byte, and 128-byte boundaries each having advantages), then some initial work will be done to copy enough bytes to align the destination, and this initial copying will waste some performance.

XMemCpy can copy enough bytes to align the destination buffer on a 16-byte boundary in just a few instructions. Then it executes a short loop to copy enough bytes to align the destination buffer on a 128-byte boundary. Similarly, **memcpy** executes a short loop to copy enough bytes to align the destination buffer on an 8-byte boundary.

If the destination buffer is already fully aligned then performance will be slightly better. This is most noticeable on relatively small copies, especially with **memcpy**.

Summary: Aligning the destination on a 16-byte boundary gives the best performance for **XMemCpy**, while aligning on an 8-byte boundary gives the best performance for **memcpy**.

Source Alignment

Once the destination is aligned, the code needs to check the source alignment. If the source buffer is now also aligned, then memory copying happens at full speed. If the source buffer is not sufficiently aligned (alignment on 4-byte, 8-byte, and 16-byte boundaries each having advantages), then extra work must be done for every byte that is copied in order to fix the alignment.

XMemCpy runs fastest if the source buffer is aligned on a 16-byte boundary (after first aligning the destination buffer on a 16-byte boundary). Thus, it runs fastest if the difference between the source and the destination is a multiple of 16. This difference is most noticeable when copying small amounts of data. However, since VMX code can use the **vperm** instruction to shift data efficiently, the performance penalty for misalignment is not huge. If the source and destination buffers are not in the cache, then the alignment penalty is completely hidden.

Since it can use 8-byte reads and writes, **memcpy** runs fastest if the source buffer has an 8-byte alignment (after first aligning the destination buffer on an 8-byte boundary). Thus, **memcpy** runs fastest if the difference between the source and destination is a multiple of 8. If the source has a 4-byte alignment but not an 8-byte alignment, then **memcpy** falls back to a 4-byte copy routine. If the source has an alignment of less than 4 bytes, then **memcpy** falls back to code that uses single-byte reads and shifts to align the data. The performance cost for using **memcpy** with an alignment of less than 4 bytes is severe.

Summary: For best performance, the source buffer should be aligned on 16-byte boundaries, relative to the destination for **XMemCpy**, and aligned on 8-byte boundaries for **memcpy**. The performance penalty is particularly severe for a relative alignment below 4 bytes when using **memcpy**.

Page Size

If 4-KB pages are used, then significant time will be wasted on Translation Look-Aside Buffer (TLB) misses, and the effectiveness of prefetching will be reduced. Using 64-KB or larger pages is important for maximum performance. For more details, see the white paper "Xbox 360 Memory Page Sizes."

Memory Routine Features

With the factors affecting performance covered, we can now explain the different memory routines available.

XMemCpy and XMemSet

XMemCpy and **XMemSet** should be the default choice for copying and filling memory. Unless the source or destination is in non-cacheable memory, they are never a bad choice.

XMemCpy128 and XMemSet128

XMemCpy128 and **XMemSet128** avoid some of the alignment and size checks and can give better performance, especially on small buffers. You can call **XMemCpy128** and **XMemSet128** if your buffer meets the following criteria:

- the destination buffer has 128-byte alignment
- its size is a multiple of 128
- the source buffer has 16-byte alignment (only applies to **XMemCpy128**)

Summary: Use **XMemCpy128** and **XMemSet128** whenever applicable.

XMemCpyExtended and XMemCpyExtended128

Due to bugs in the implementation of **xdcbt**, these functions have been removed from the XDK.

XMemCpyStreaming, XMemCpyStreaming_Cached, and XMemCpyStreaming_WriteCombined

XMemCpyStreaming is a wrapper function that calls either **XMemCpyStreaming_Cached**, for a cacheable destination, or **XMemCpyStreaming_WriteCombined**, for a write-combined destination. If the caller knows the type of the destination memory, it can call the specialized functions directly. These functions all assume a cacheable source buffer.

XMemCpyStreaming_Cached and **XMemCpyStreaming_WriteCombined** have comparable performance to **XMemCpy** for data that is not already present in the L2 cache, but they have a smaller cache footprint. Each 128-byte aligned portion of copied data is evicted from the L2 cache immediately after use. Consequently, only 1/8 of the 8-way cache is modified by the operation (2/8 if the source and destination are aligned such that their cache use is perfectly coherent). These routines offer the best alternative for relatively large copy operations when either:

- the destination buffer is write-combined

  or

- both of the following are true:
  - The data to be copied is not already present in the cache.
  - The data to be copied does not need to remain in the cache.

Manual Copying and memcpy

The XMem functions are almost always the best choice — but there is one other situation to consider.

The XMem copy routines are very fast on medium to large blocks of memory, and they should be your first choice in these cases. However, these generic memory copy routines can’t make any assumptions about the size or alignment of the buffers, so they must do numerous checks at startup to decide which code paths to take. These checks can harm performance when copying small buffers. In addition, when copying small amounts of data, the overhead of calling the functions can take longer than copying the data.

If you have a small amount of data — less than about one hundred bytes worth — and you know the data alignment, particularly the destination alignment, you should consider writing a specialized inline function to handle the copying for you. For instance, if you need to copy four **__vector4s** that have a 16-byte alignment, then the following code will be faster than any library routine could ever be:

```
// Mark the pointers with __restrict for best code scheduling.
inline void Copy4Vectors(__vector4* __restrict dst,
                         const __vector4* __restrict src)
{
    dst[0] = src[0];
    dst[1] = src[1];
    dst[2] = src[2];
    dst[3] = src[3];
}
```

If the amount of memory to be copied is known at compile time, then there is an easier way to create inline copy code**.** Because it is also an intrinsic, **memcpy** is a special function. In some cases when you use **memcpy**, it actually generates inline code to copy the data instead of making a function call. For very small copy operations, this can be the fastest and simplest method.

The current rule is that if the size of the data to be copied is known at compile time, and if ten or fewer items need copying, then inline code is generated. If four or fewer items need copying, then the inline code generally has no branches. The details may change in the future, especially the exact cut-off points, but the basic idea should remain the same.

The key factor in using this optimization effectively is having the correct pointer types. The compiler looks at the type of pointers used for the source and destination and uses that information to infer the alignment and the item size to copy. If your pointers are void* or char*, then the compiler assumes nothing about the alignment and copies a byte at a time. If your pointers are int* or __int64*, then the compiler assumes that your data is aligned on 4-byte or 8-byte boundaries and copies 4 bytes or 8 bytes at a time. Thus, using the largest type that accurately reflects your alignment gives the best performance.

If your pointers do not point to natively aligned data, then the processor may need to go to microcode (if some of the reads or writes cross a 32-byte boundary); however, the results will still be correct. This would not be true if the compiler generated VMX loads and stores, so VMX loads and stores are never generated by the compiler for **memcpy**.

To get the best code possible, it is important to mark the pointers with **__restrict** so that the compiler knows that the source and destination do not overlap. This allows the compiler to arrange loads and stores as efficiently as possible.

If the data being copied is a structure, then the compiler infers the alignment from the types used in the structure. If the structure is copied by assigning one structure to another, then an implicit **memcpy** is generated, which generates a function call or inline copy code based on the rules described in the preceding text.

The following code is an example of efficiently copying a structure with an inlined **memcpy**:

```
struct TestStruct
{
    __int64 a[4];
};

// This will generate very efficient code to copy this struct.
// Four 8-byte loads and then four 8-byte stores.
void AssignStruct(TestStruct* __restrict dst,
                  const TestStruct* __restrict src)
{
    memcpy(dst, src, sizeof(*dst));
    // Structure assignment produces identical results
    //*dst = *src;
}
```

Summary: When copying small amounts of data with known alignment, consider doing an inline copy. If you need to copy 32 bytes or fewer, you may be able to get ideal inline code by using the **memcpy** intrinsic or by doing structure assignment.

Memset

The function **memset** is always slower than **XMemSet** so it should only be used when filling non-cacheable memory.

Performance Comparisons

The performance of copying and filling memory depends on many factors, including the contents of the cache, alignment, the activity of other threads, and other complex factors. However, there are some general rules about the performance that you can expect, as described in the following tables.

When copying a large amount of data, where the source and destination cannot fit in the cache, **XMemCpy** can overlap the aligning of data with the time spent waiting for memory, which eliminates the penalty for misaligned copies. **XMemCpy** also benefits from using **__dcbz128** on the destination buffers. Table 1 compares the speed of memcpy, **XMemCpy**, and **XMemCpyStreaming** on 4 MB of memory (without data in the cache).

Table 1. Memory copy speed for 4-MB buffers with various relative alignments

Relative Alignment	memcpy	XMemCpy	XMemCpyStreaming
16-byte	1.26 GB/s	2.43 GB/s	2.31 GB/s
8-byte	1.37 GB/s	2.53 GB/s	2.41 GB/s
4-byte	0.86 GB/s	2.53 GB/s	2.40 GB/s
1-byte	0.19 GB/s	2.53 GB/s	2.41 GB/s

When copying smaller amounts of data, the performance varies dramatically depending on whether the source and destination buffers are already in the cache. If the destination buffer is not yet in the cache, then **XMemCpy** has a significant performance advantage over **memcpy**; Table 2 shows this advantage for various relative alignments of data.

Table 2. Memory copy speed for 64-KB buffers, no data in the cache

Relative Alignment	memcpy	XMemCpy	XMemCpyStreaming
16-byte	1.22 GB/s	3.18 GB/s	2.21 GB/s
8-byte	1.40 GB/s	3.12 GB/s	2.25 GB/s
4-byte	0.90 GB/s	3.21 GB/s	2.25 GB/s
1-byte	0.19 GB/s	3.20 GB/s	2.16 GB/s

When the source and destination buffers are already in the cache and the relative alignment is 8 bytes, then **memcpy** has its biggest advantage, as shown in Table 3. However, the performance of **XMemCpy** is much more consistent and reliable.

Table 3. Memory copy speed for 64-KB buffers, data in the cache

Relative Alignment	memcpy	XMemCpy	XMemCpyStreaming
16-byte	3.56 GB/s	3.95 GB/s	3.52 GB/s
8-byte	4.77 GB/s	4.50 GB/s	3.74 GB/s
4-byte	0.92 GB/s	4.50 GB/s	3.60 GB/s
1-byte	0.23 GB/s	4.50 GB/s	3.39 GB/s

XMemSet always runs faster than **memset**, sometimes by a significant margin. Table 4 compares the fill rate for these two on 4 MB of data with zero and non-zero fill values.

Table 4. Memory-filling speed for 4-MB buffers with various fill values

Fill	memset	XMemSet
Zero	1.23 GB/s	9.68 GB/s
Non-zero	1.23 GB/s	7.42 GB/s

When filling memory by using **XMemSet**, filling a small range of memory is limited only by the L2-cache speed. So, it can be filled faster than a large range of memory, which is ultimately limited by the front-side bus speed. Table 5 shows the fill rate for 64 KB of data.

Table 5. Memory-filling speed for 64-KB buffers with various fill values

Fill	memset	XMemSet
Zero	1.07 GB/s	14.59 GB/s
Non-zero	1.10 GB/s	8.19 GB/s

Summary

- Use **XMemCpy** and **XMemSet** whenever possible.
- Use **XMemCpy128** and **XMemSet128** when applicable.
- Use **XMemCpyStreaming** and **memset** if your destination buffer is non-cacheable memory.
- Use **XMemCpyStreaming** if your data is not already in the cache, and it does not need to remain in the cache.
- Use **memcpy** or hand-written copy loops when copying small amounts of data.
- Keep your buffers well aligned when maximum performance is required.
- Always try to use 64-KB pages for maximum memory performance.