Xbox 360 Memory Page Sizes

Bruce Dawson
Software Design Engineer
Advanced Technology Group (ATG)

Published: April 16, 2005
Updated: January 23, 2008

Introduction

The Xbox 360 CPU uses a memory management unit (MMU) to map the virtual memory addresses seen by the programmer to the physical addresses seen by the hardware. The MMU also allows address-specific memory attributes such as read-only, write-combined, and executable.

However, this translation is not free. There are a number of caches on the Xbox 360 CPU that can quickly translate virtual addresses to physical addresses, but when the limits of these caches are exceeded performance may drop. In the worst case, performance can drop by more than half, although a 5–10 percent drop is more common.

By allocating and using memory appropriately, the cost of the virtual-to-physical address translation can be almost entirely eliminated.

ERATs

Each CPU core has two Effective-to-Real Address Translation caches (ERATs). These ERATs are small, fast caches of recently used translations from effective addresses to real (or physical) addresses, and they are queried on every address reference. From the perspective of games, effective addresses and virtual addresses are the same thing, so the terms are used interchangeably. One ERAT is used for translating the addresses of instructions and is known as the I-ERAT. The other is used for translating the addresses of data and is known as the D-ERAT. An ERAT entry is used for all memory, even physical memory allocated with **XPhysicalAlloc** and non-cacheable memory, so using **XPhysicalAlloc** does not reduce the number of ERAT misses.

Each ERAT is implemented as a 64-entry two-way (32×2) cache, with each cache entry remembering the translation for a 4-KB block of memory (aligned on a 4-KB boundary). ERAT entries always cache the translation for a 4-KB block of memory, regardless of the page size used by the MMU. Each ERAT is laid out as shown in the diagram in Figure 1.

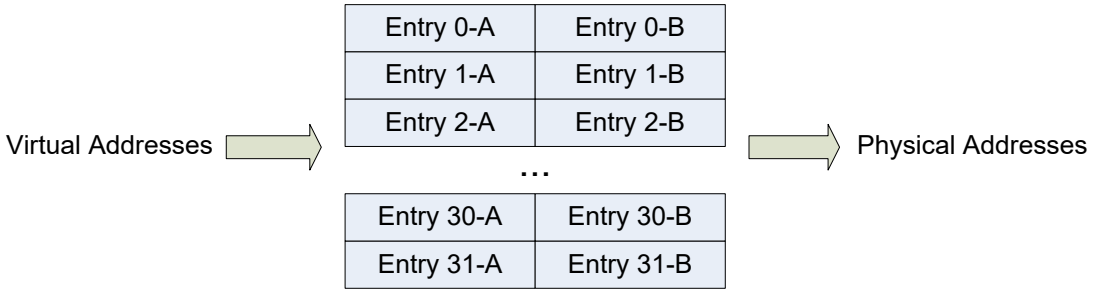

Figure 1. Layout of an Effective-to-Real Address Translation (ERAT) cache

The ERAT entry used for a particular virtual address is chosen by dividing the virtual address by 4-KB and then using the result, modulo 32, to select a pair of entries. If neither entry contains the requested translation, then the least-recently-used entry from the pair is loaded with the translation data.

A single core's ERATs can cache translations for 256 KB of data and 256 KB of code. This is much more than the 32 KB that each of the L1 data and instruction caches can hold, so usually the number of ERAT misses is low compared to the number of L1 cache misses. Since an ERAT miss has comparable cost to an L1 miss and should happen less frequently, ERAT misses are not usually a significant performance factor.

However, there are a few factors that can make ERAT misses happen more frequently:

- If your code references cache lines widely scattered through memory, then it is possible to have more ERAT misses than L1 cache misses, because the ERATs have fewer entries than the L1 caches. This can happen if you make function calls to many small scattered functions, or walk a data structure with randomly allocated nodes.
- The D-ERAT is used on writes as well as reads, so code that writes a lot of data may incur many ERAT misses.
- If two hardware threads are running on one core, then they must share the caches and the ERATs. The two threads share the caches cooperatively, with one thread able to read code and data loaded by the other thread. However, the two threads do not share ERAT entries. If both threads reference the same address they use one ERAT entry each to cache the translation. Therefore the effective size of the ERATs is reduced when two threads are running.
- If your code cycles through three addresses separated by a multiple of 128 KB, then the ERAT will thrash, as all three addresses map to the same set in the ERAT, and there are only two entries per set.

The best way to minimize the number of ERAT misses is to try to maximize locality of reference. If you are going to read some code or data from a 4-KB block, try to ensure that you use most or all of the code or data from that 4-KB block. That will ensure that you get maximum usage from the ERAT entries.

Prefetches that miss in the ERAT will cause a pipeline flush - the cost of loading the ERATs cannot be hidden.

TLB

If an address can't be found in an ERAT, then the address is sent to a Translation Look-aside Buffer (TLB). Each CPU core has a TLB which is a virtual-to-physical translation cache used for both code and data. The TLB is implemented as a 1024-entry four-way (256x4) cache, with each entry remembering the translation for one page. Unlike the ERATs, the page size in the TLB is variable. Each entry can translate 4 KB, 64 KB, or 16 MB of memory. A TLB lookup entry is used for all memory, even physical memory allocated with **XPhysicalAlloc**, so using **XPhysicalAlloc** does not reduce the number of TLB misses.

The TLB is arranged similarly to the ERATs but with more rows and columns, as shown in Figure 2.

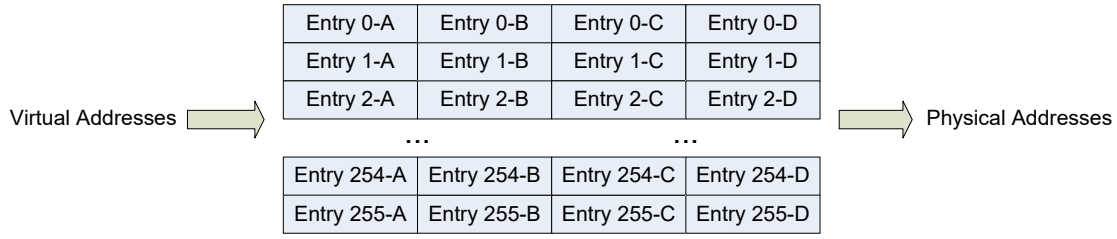

Figure 2. Layout of a Translation Look-aside Buffer (TLB)

The TLB entry used for a particular virtual address is chosen by dividing the virtual address by the relevant page size (4 KB, 64 KB, or 16 MB) and sending the result through a hash function to generate a number between 0 and 255. This number is used to select a set of four entries. If none of the entries contain the requested translation, then the least-recently-used entry from the set is loaded with the translation data.

TLB misses are much more expensive than ERAT misses, because they require running the software TLB handler and they may require going to main memory to load additional data.

A typical game's primary thread references 5,000-6,000 individual 4-KB blocks in a single frame. If each entry in the TLB maps 4 KB, then thousands of TLB entries are needed, and there will be thousands—or hundreds of thousands—of TLB misses per frame. However, if 64-KB pages are used then the number of TLB misses will drop dramatically—almost to zero. The 1024 entries in the TLB can map 64 MB of memory when 64-KB pages are used, which exceeds the working set for most games.

TLB misses are expensive, but an equally important problem is their interaction with the prefetching instruction—**dcbt**. This instruction is important for hiding memory latency in order to get maximum performance. However, if a prefetch is done to an address that misses in the TLB, then the prefetch is discarded. Assuming 4-KB pages are being used and prefetching is being done eight cache lines in advance, then after every 4 KB the CPU discards eight of the 32 prefetches due to TLB misses. This can dramatically decrease performance on code that processes large quantities of data. Ironically, the performance slowdown is greatest when the most prefetching is done.

It is possible to force a TLB entry to be loaded, by loading data from the new page, but it is difficult to do properly and can hurt performance if done incorrectly. A much better solution is to reduce the number of TLB misses by using larger pages.

Using 64-KB pages can improve the performance of memory bandwidth benchmarks by two-and-a-half times or more. Real code isn't affected quite as dramatically but consistent use of 64-KB pages can easily give games a 5–10% performance performance boost for very little effort.

Larger pages do have some disadvantages. The minimum granularity for adjusting memory protection settings (with **VirtualProtect** or **XPhysicalProtect**) is a single page. Therefore, larger page sizes give less precision for applying attributes such as **PAGE_READONLY**. Similarly, the minimum granularity for allocating memory from the OS with **VirtualAlloc** or **XPhysicalAlloc** is one page. If used carelessly, larger page sizes can waste significant amounts of memory.

Four-KB pages maximize memory usage but at a high performance cost. Sixteen-MB pages waste too much memory for most purposes. Sixty-four-KB pages usually strike just the right balance.

Identifying Page Sizes

The Xbox 360 memory map defines memory ranges for the various combinations of memory type and page size. This mapping will not change, so it can be used to identify the page size a particular block of memory is using. The page size ranges for different types of memory are shown with these #defines:

```
#define MM_VIRTUAL_4KB_BASE     ((ULONG)0x00000000)
#define MM_VIRTUAL_4KB_END      ((ULONG)0x3FFFFFFF)
#define MM_VIRTUAL_64KB_BASE    ((ULONG)0x40000000)
#define MM_VIRTUAL_64KB_END     ((ULONG)0x7FFFFFFF)
#define MM_IMAGE_64KB_BASE      ((ULONG)0x80000000)
#define MM_IMAGE_64KB_END       ((ULONG)0x8BFFFFFF)
#define MM_ENCRYPTED_64KB_BASE  ((ULONG)0x8C000000)
#define MM_ENCRYPTED_64KB_END   ((ULONG)0x8DFFFFFF)
#define MM_IMAGE_4KB_BASE       ((ULONG)0x90000000)
#define MM_IMAGE_4KB_END        ((ULONG)0x9FFFFFFF)
#define MM_PHYSICAL_64KB_BASE   ((ULONG)0xA0000000)
#define MM_PHYSICAL_64KB_END    ((ULONG)0xBFFFFFFF)
#define MM_PHYSICAL_16MB_BASE   ((ULONG)0xC0000000)
#define MM_PHYSICAL_16MB_END    ((ULONG)0xDFFFFFFF)
#define MM_PHYSICAL_4KB_BASE    ((ULONG)0xE0000000)
#define MM_PHYSICAL_4KB_END     ((ULONG)0xFFFFFFFF)
```

The following function, available in the XDK, uses these #defines to translate from an address to a page size.

```
DWORD XMemGetPageSize( const
VOID* address )
{
    ULONG ulAddress = (ULONG)address;
    if (ulAddress <=MM_VIRTUAL_4KB_END) return 4096;
    if (ulAddress <=MM_VIRTUAL_64KB_END) return 65536;
    if (ulAddress < MM_IMAGE_4KB_BASE) return 65536; // Note: less than
                        // base rather than less than or equal to end
    if (ulAddress <=MM_IMAGE_4KB_END) return 4096;
    if (ulAddress <=MM_PHYSICAL_64KB_END) return 65536;
    if (ulAddress <=MM_PHYSICAL_16MB_END) return 16 * 1024 * 1024;
    return 4096;
}
```

The mapping of address ranges to page sizes can be more easily seen in the diagram in Figure 3.

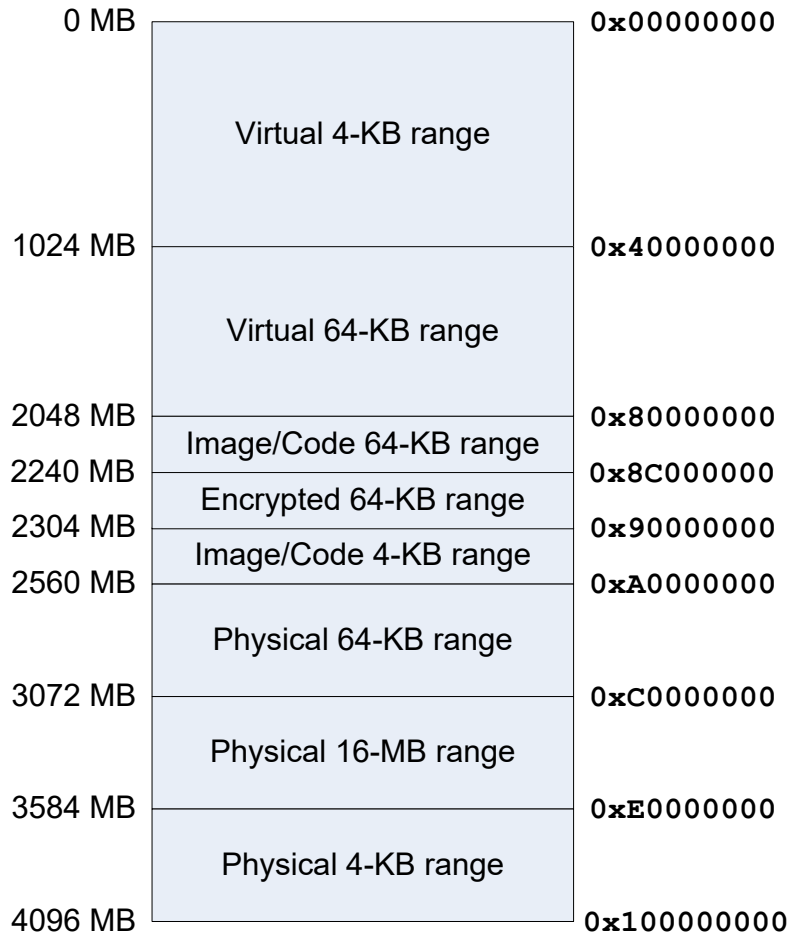

Figure 3. Xbox 360 memory map

If you capture a CPU execution trace with **XTraceStartRecording** or with the **Record CPU Trace** command in PIX, and then analyze it in PIX, it will tell you how many 4-KB ERAT pages and how many TLB pages of each size you referenced in that recording. This can be used to detect excessive ERAT and TLB usage. Since trace recordings of CPU execution record only one thread, and don't record system code, total ERAT and TLB usage may be higher than what trace analysis shows.

Requesting Large Pages

The method for requesting large page sizes varies depending on the type of memory being allocated. The page size cannot be changed once the memory is allocated.

Code

Code defaults to using 64-KB pages.

The Xbox 360 linker has an alignment switch that takes two arguments. The first argument specifies the alignment for sections of like attributes (like multiple code sections or multiple data sections) and the second is the desired page size. The default is /ALIGN:128,65536.

If an executable has many different sections that cannot be merged, then each section uses a multiple of 64 KB, which wastes some space. However, for any reasonable game executable, the default page size of 64-KB should be used.

The align option can be manually specified in the command line area of the linker section in project properties, but there should be no reason to change it.

Stack

Thread stacks use 64-KB pages as long as the stack size is a multiple of 64 KB. Make sure your stack size is a multiple of 64 KB. To set the default stack size, set the stack size in the properties for your project.

To set the stack size in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box.
2. Click the **Linker** folder.
3. Click the **System** property page.
4. Modify the the appropriate property for your version of Visual Studio:
    - In Visual Studio 2010, modify the property **Stack Commit Size**.
    - In Visual Studio 2008, modify the property **Stack Size**.

If you specify a stack size when creating a thread, be sure to make it a multiple of 64 KB.

Physical Allocations

Physical allocations can use 4-KB, 64-KB, or 16-MB pages. Four-KB pages are the default. You can request large pages by using **XPhysicalAlloc** with a bitwise OR operation on the parameter *flProtect* to specify one of the following flags: MEM_LARGE_PAGES, for 64-KB pages, or MEM_16MB_PAGES, for 16-MB pages. This will round up your allocation size and alignment to a multiple of the page size, and give you the requested type of page. In general, we recommend using MEM_LARGE_PAGES with physical allocations.

```
// Allocate physical memory with 64-KB pages
int* pPhysical = (int*)XPhysicalAlloc(dataSize, MAXULONG_PTR, 0,
                 PAGE_READWRITE | MEM_LARGE_PAGES);
```

Virtual Allocations

Virtual allocations can use 4-KB or 64-KB pages. Four-KB pages are the default. You can request 64-KB pages with **VirtualAlloc** by performing an OR operation in MEM_LARGE_PAGES to the *flAllocationType* parameter. This will round up your allocation size to a multiple of the page size and give you the requested type of page. In general, we recommend using MEM_LARGE_PAGES with virtual allocations.

```
// Allocate virtual memory with 64-KB pages
int* pVirtual = (int*)VirtualAlloc( NULL, dataSize,
                MEM_COMMIT | MEM_LARGE_PAGES,
                PAGE_READWRITE );
```

Heap Allocations

Heap allocations, which include **LocalAlloc**, **HeapAlloc**, **malloc**, and **new**, always use 64-KB pages. The 64-KB page heap can be less memory efficient in some cases. If you create a heap with **HeapCreate**, and make few allocations from that heap, then the 64-KB granularity wastes space.

If you make large allocations from the heap—around 1-MB or greater—then these allocations are forwarded to **VirtualAlloc** and do not share pages with other allocations. This means that an average of half a page is wasted on each allocation. The worst-case scenario is allocating an exact multiple of 64 KB, since this allocation size is padded with header overhead and *then* rounded up to a multiple of 64 KB, guaranteeing that almost 64 KB will be wasted. The header overhead is typically 48 bytes but is greater when using **new** or **malloc** in debug builds. Consider using **VirtualAlloc** for large allocations, or allocate slightly less than a multiple of 64 KB.

Small allocations are handled as efficiently with the 64-KB heap as with the old 4-KB heap. Multiple allocations are packed together onto one page. However, with both the old and the new heap, it is important to remember that all allocations are rounded up to a multiple of 16 bytes, and there are 16 bytes of overhead per allocation. Thus, there are 16–31 bytes of memory wasted for each allocation. If you are doing many small allocations, you should consider your own fixed-size allocation pool to reduce or eliminate this overhead.

XMemAllocDefault

When calling **XMemAllocDefault**, there is no way to specify that you want large pages, except for the special case of write-combined physical pages. These are represented by XALLOC_MEMPROTECT_WRITECOMBINE_LARGE_PAGES. The MEM_LARGE_PAGES flag cannot be used with **XMemAlloc**. If you implement **XMemAlloc**, you need to call **VirtualAlloc** or **XPhysicalAlloc**, instead of **XMemAllocDefault**, to allocate large pages.

CPU Hardware Hang

The Xbox 360 CPU has a hardware bug that can be triggered by code that maps the same memory in at different locations, if some locations map the memory as cacheable, and other locations map it as non-cacheable.

The hardware bug is caused by code that first reads memory in a large page size allocation from a cached read-only view, and then tries to read the same data from the non-cached view. If the cached read-only data is still in the L1 cache, the non-cached read causes the core to hang, and neither Visual Studio nor the kernel debugger can break in. Note that the hardware fault hangs the CPU core only where the reads were executed. This bug happens only with large pages. With 4-KB pages the bug is avoided. To work around this issue, avoid reading data from non-cached memory.

If you suspect that you are encountering this bug, use the **XEnableSmallPagesOverride** interface to force the use of the 4-KB page size for all physical allocations. If the hang goes away, this bug is the likely culprit. Use trace recording to track down all non-cached reads and eliminate them.

For more details see the Memory Views sample.

Summary

Following these steps will ensure that your game doesn’t waste performance on excessive ERAT and TLB misses.

- Use PIX to analyze CPU Execution Trace Recordings, to identify excessive ERAT and TLB misses.
- Use large pages to minimize TLB misses. Make sure you are using 64-KB pages for your code, stack, physical allocations, virtual allocations, and heap allocations.
- Use good locality of reference to minimize ERAT and TLB misses. Use the **Memory Accesses** tab in PIX when analyzing CPU Execution Trace Recordings to look for poor locality of reference.