Xbox 360 CPU Caches

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

Published: October 7, 2004
Updated: February 23, 2007

Introduction

Every year Moore's law gives us faster transistors and lets us squeeze more of them onto a single chip. This lets hardware designers give us faster and more powerful processors. Unfortunately, Moore's law doesn't apply to the speed of light, and by extension it doesn't apply to the speed of electricity. In fact, as the wires in processors get thinner, the electrical signals actually propagate *more* slowly. Since microprocessor chips are staying the same size—or getting bigger—this means that the number of cycles that electricity takes to get across a chip is increasing. These wire delays—and limits on how much logic can be done in the shorter cycles—mean that even level-one caches can't respond to memory requests in a single cycle.

The situation is worse when accessing memory. Memory speeds have not increased as quickly as processor speeds, and the long wires between the CPU and the memory—with a complex memory controller in-between—add substantial latency.

These are the realities of modern processors: even the fastest memory can't keep up. Keeping these processors fed with data and instructions requires careful planning. Code and data structures must be laid out to maximize cache efficiency. Prefetching and cache-line zeroing are needed in order to hide latency and conserve bandwidth. Finally, in a multiprocessor system it is important to get the separate processors to efficiently communicate and share data. Many of these optimizations are architectural changes that are best done early in the development process.

By understanding the design of the Xbox 360 CPU caches, you can help your game live up to its full potential.

Hardware Features and Xbox Comparison

The Xbox 360 CPU is a custom IBM part based on the PowerPC architecture. It features three identical cores and a single 1-MB L2 cache, all on one chip. The main cache-related features of the Xbox 360 CPU are:

- Three identical custom 64-bit PowerPC cores running at 3.2 GHz
- Two hardware threads per core
- 32-KB two-way 51.2 GB/sec L1 instruction cache, per core
- 32-KB four-way 51.2 GB/sec L1 data cache, per core; no-allocate, write-through
- Shared 1-MB eight-way 102.4 GB/sec L2 cache; write-allocate, write-back
- 128-byte cache lines on all caches
- CPU-to-RAM (front side bus) peak bandwidth of 10.8 GB/sec for reads, 10.8 GB/sec for writes

- Big-endian byte ordering
- Cache locking and direct read access to the L2 cache by the GPU
- Cache snooping for I/O

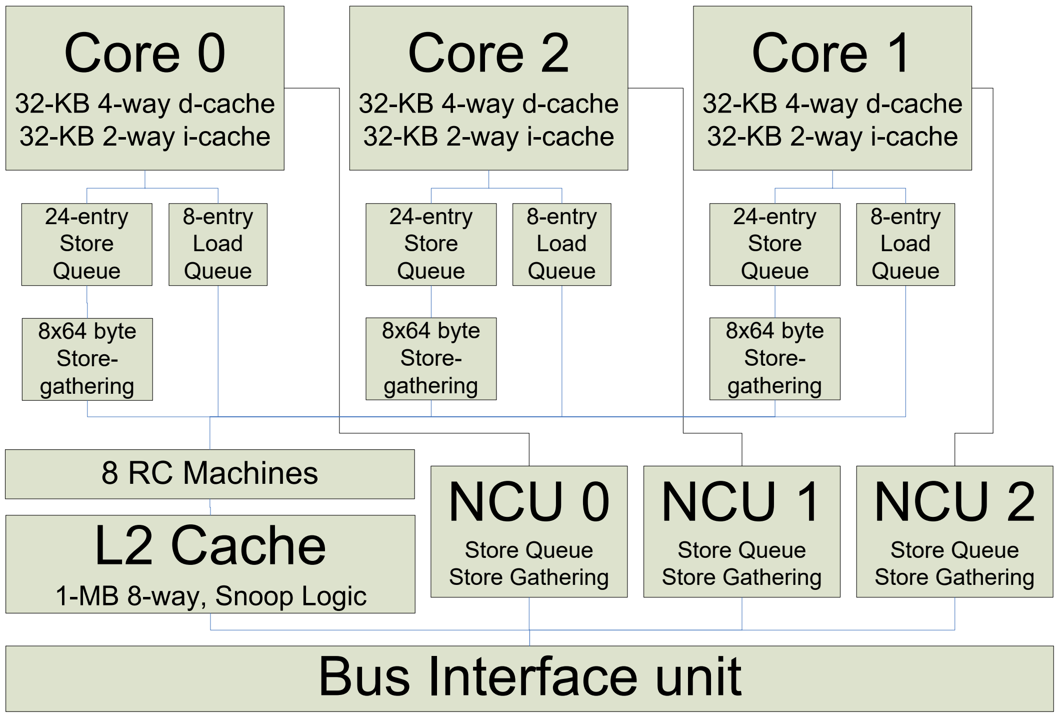
For comparison, the Xbox CPU is an Intel Pentium III CPU with L2 cache on chip. Its main cache-related features are:

- One core running at 733 MHz
- One hardware thread
- 16-KB four-way L1 instruction cache
- 16-KB four-way L1 data cache, write-allocate, write-back
- 128-KB eight-way L2 cache, write-allocate, write-back
- 32-byte cache lines on all caches
- CPU-to-RAM (front side bus) bandwidth of 1.0 GB/sec total
- Little-endian byte ordering

One of the important changes made for the Xbox 360 platform was to increase the size of all of the caches. The L2 cache, in particular, was made eight times larger, which dramatically reduces the L2 cache miss rate.

Another significant change was increasing the size of the cache lines, from 32 bytes to 128 bytes. This change means that an Xbox 360 cache miss brings in four times as much data as an Xbox cache miss, which significantly reduces the number of cache misses and therefore improves performance. If all bytes in cache lines are used, the increased cache line size avoids 75 percent of cache misses.

The following high-level diagram shows the main cache-related features of the Xbox 360 CPU chip.



Reading Data

In this section, we will cover the process of reading data. This includes the varying latencies on reads, the CPU's behavior on cache misses, instructions available for prefetching, and non-cacheable reads.

L1 Hit

The L1 data cache has a latency of five cycles. Most of this latency is hidden by pipeline offsets, so the visible latency is usually two cycles. For example:

```
lwz  r1,0(r3)  // Issues on cycle n
add  r2,r2,r1  // Stalls, issues on n + 2 if data was in L1
```

In some cases, the full five-cycle latency is visible—for instance, when the result of one load is used as the address of a subsequent load, as in the following example:

```
lwz  r1,0(r3)  // Issues on cycle n
lwz  r2,0(r1)  // Stalls, issues on n + 5 if data was in L1
```

Finally, if an integer instruction does math on an address and that address is used immediately, there will be a five-cycle delay before the load instruction can start. For example:

```
add  r1,r1,r3  // Issues on cycle n
lwz  r2,0(r1)  // Stalls, issues on n + 5
```

The Xbox 360 compiler knows about all of the instruction stalls, and when possible it will rearrange instructions to hide them. The more flexibility the compiler has to rearrange code, the more it can hide these stalls. Marking pointers with **__restrict** is one way to do this. For instance, this code requires that the compiler do a load and a store, and then a load and a store.

```
void CopyTwoInts(int* dest, const int* src)
{
    dest[0] = src[0];
    dest[1] = src[1];
}
```

The compiler can't hide the latency of the loads because *src* and *dest* might point to overlapping memory ranges.

The following improved version of the code uses the **__restrict** keyword to tell the compiler that the pointers don't overlap, which allows it to schedule the code as two loads, and then two stores, which hides the load latency.

```
void CopyTwoInts(int* __restrict dest, const int* __restrict src)
{
    dest[0] = src[0];
    dest[1] = src[1];
}
```

L1 Miss, L2 Hit

If data being loaded is not in the L1 cache, the behavior is different in a couple of important ways. The most obvious change is that the latency is greater—it takes a minimum of about 41 cycles for data to come from the L2 cache. The actual time varies depending on how busy the L2 cache is servicing requests from other threads and is a few cycles longer for core 1 and core 2 than for core 0 (due to their greater physical distance from the L2 cache).

The other significant change is that on cache misses the CPU core doesn't stall if you try to access recently loaded data before it is ready—instead it flushes the thread. For example:

```
lwz  r1,0(r3)  // Issues on cycle n
add  r2,r2,r1  // Flushes, re-issues on n + ~41 if data was in L2
```

The load instruction itself doesn't stall or flush. It is only when the CPU core tries to use the data that a stall or flush is needed. If the data is not coming from the L1, the instruction—and those following it—is flushed and is then re-issued when the data arrives. This is important because a stall of 41 cycles or more would block the other thread on that core, whereas flushing lets the other thread continue running. The distinction between stalling and flushing is not important for the thread that is waiting for data—the delay is the same—but it is a critical difference for the other thread running on the same core.

Depending on how the L2 cache is being used, reads from the L2 cache from one core may occur out of order, but this will be visible only if you try sharing data between hardware threads without synchronization.

The latency of an L1 miss is too large for the compiler to schedule around. Instead, the compiler schedules code assuming L1 hits.

L2 Miss

An L2 miss behaves similarly to an L1 miss, but with greater latency. The latency of an L2 miss is approximately 610 CPU cycles, as measured by following a linked list that doesn't fit in the L2 cache. The latency may be greater if the L2 cache is busy servicing other requests. Contention for the memory controller will also cause some variation, but the CPU has high priority on memory accesses.

The L2 cache can process up to eight L2 cache misses simultaneously. Additional misses to the same cache line don't add to this count.

After a cache miss the entire cache line—all 128 bytes—will be loaded into the L1 and L2 caches. That means that subsequent reads within the same 128-byte cache line will execute without a cache miss penalty.

Cache Control—Prefetching

Some of the cache miss latency can be hidden if you tell the CPU in advance that you will need the data. The CPU can start reading the data earlier so that the data will be in the cache—or at least on the way—when you try to use it. This is particularly important when processing long streams of data. The Xbox 360 CPU does some automatic instruction prefetching but does not do automatic data prefetching. That must be done by the programmer. The prefetch instruction is **dcbt**—named for *data cache block touch*.

Data Cache Block Touch (dcbt)

The **dcbt** instruction—available via the **__dcbt** intrinsic—tells the CPU to prefetch the cache line containing the specified address into the L1 and L2 caches.

Extended Data Cache Block Touch (xdcbt)

Due to bugs in the implementation of **xdcbt** this instruction has been removed from the XDK.

Data Prefetching Summary

Each CPU core can have up to eight outstanding reads—due to prefetches or regular reads—in its queue. If additional reads or prefetches are done while the queue is full, the thread will be flushed and the instruction will be reissued later. This means that a prefetch on Xbox 360 is normally guaranteed to fetch the specified cache line. The one situation in which a prefetch instruction will not fetch the specified cache line is if the address causes a miss in the Translation Look-aside Buffer (TLB). TLB misses can be minimized by using large pages—see the white paper Xbox 360 Memory Page Sizes for details.

Additionally, the L2 cache can be processing up to eight reads and writes, including associated castouts.

Prefetching on the final hardware can give a theoretical maximum speedup of 700 percent by allowing eight reads to be processed in parallel, instead of the one read at a time that may happen without prefetching. For example, this piece of code reads from a series of cache lines:

```
int sum = 0;
for( int i = 0; i < dataSize; i += 128 )
      sum += pChar[ i ];
```

If each read is an L2 miss, the code will take about 610 cycles for each cache line it processes. This is because one cache line read doesn't start until the previous one has finished.

However, if the cache lines are prefetched, there can be eight cache line reads happening in parallel at all times, and the code could run up to eight times faster. The actual improvement measured in this artificial scenario is a speedup of 6.6 times when prefetching is used.

```
for( int i = 0; i < 1024; i += 128 )
      __dcbt( i, pChar );
int sum = 0;
for( int i = 0; i < dataSize; i += 128 )
{
    __dcbt( i + 1024, pChar );
    sum += pChar[ i ];
}
```

Even this code spends a lot of time waiting for the first cache line to show up. Prefetching earlier would help to hide this. This code also has some inefficiency because it prefetches eight cache lines beyond the end of the data it is processing. This wastes bandwidth and cache lines that might be needed by other threads. You should avoid prefetching unnecessary data. The code is also very inefficient because it uses only one byte from each cache line it reads.

Prefetching more than 1024 bytes in advance would not help because there can only be eight cache line reads being processed simultaneously. In more realistic scenarios, where the L2 cache bandwidth is shared with other threads and more calculation is being done per cache line, it might make sense to prefetch an even smaller distance ahead.

Prefetching can make a huge difference to your performance. If you can structure your data and algorithms to facilitate prefetching, your code will run much faster. Try to prefetch as much of your data as possible. Even if you can’t prefetch the full 610 cycles in advance, prefetching will still hide part of the cache miss latency. Prefetching is also beneficial for data that is in the L2 cache, to make sure it is in the L1 cache when you need it.

Non-cached Reads

Non-cached reads from memory take about as long as L2 misses. This makes them extremely expensive, so they are best avoided. Reading from non-cacheable memory (allocated with PAGE_NOCACHE or PAGE_WRITECOMBINE) should not be necessary on Xbox 360.

Writing Data

This section covers the process of writing data. This includes the policies of the different cache levels, the behavior of the store-gather queues, instructions available for more efficient writing, and non-cacheable writes.

L1 Cache

The L1 data cache is a *no-allocate, write-through* cache. No-allocate means that if the address you are writing to is not already in the L1 cache, then the data you write will not be stored in the L1 cache—no cache line will be allocated for it. This avoids wasting space in L1 with data that you never read. If the address you are writing to is in the L1 cache, the L1 will be updated.

The no-allocate policy avoids polluting the L1 cache with data that you are writing. If you do read-modify-write operations, however, the read will load your destination data into the L1 cache.

Write-through means that the data immediately starts heading towards the L2 cache; there are never any modified cache lines in the L1 cache. The write-through policy ensures that synchronizing data between processors is always fast because there is never any modified data stored in L1 caches that needs to be flushed to L2.

Store Gathering

There are several levels of buffering and queuing on the way to the L2 cache, but the most interesting parts are the eight 64-byte store-gather buffers in each CPU core. These buffers gather up writes to an aligned 64-byte address range so they can be written to the L2 cache in one operation. This gathering is important in order to maximize write bandwidth and reduce L2 usage. The store-gathering buffers are very flexible. They can gather writes in any order, they support writing the same address multiple times, and they support skipping some bytes—writing every second byte, for instance.

When a new buffer is needed or a time-out expires, the oldest store-gather buffer is written to the L2 cache. This write takes the same amount of time whether the store-gather buffer contains one byte of data or 64 bytes of data. In other words, you can make more efficient use of the L2 cache if your writes are close together.

When you use store-gather buffers, writes do not necessarily go to the L2 cache in order. One write can make it to the L2 cache sooner than an earlier write to a different 64-byte block.

L2 Cache

The L2 cache is a write-allocate write-back cache. The term *write-allocate* means that if the address you are writing to is not already in the L2 cache, then a cache line will be allocated for it. The newly allocated cache line will be filled with data read from memory before the write happens. This read can cause a substantial performance hit because of the long memory latency. If the cache line ends up being completely written to, then the read is just wasted time and bandwidth. As explained below, prefetching the destination or zeroing it with **__dcbz128** is an important tactic to avoid this penalty.

The term *write-back* means that writes to the L2 cache are not written to memory until they have to be. Cache lines can be forced out when the cache line is needed for other data, or they can be explicitly forced out by cache control instructions. If you need your writes to be visible to the GPU, you need to explicitly force them to memory. The GPU does not snoop the L2 cache, and it will not see data there.

Cache Control—Zeroing and Flushing

Following are the cache control instructions associated with writing data.

dcbz128—Data Cache Block Zero 128

If you are writing to an entire cache line, it is wasteful to have the data read from memory only to have you overwrite it. This wastes memory bandwidth and makes the initial write take longer. Instead, you should use the **dcbz128** instruction—available via the **__dcbz128** intrinsic—to zero the cache line. **dcbz128** allocates a cache line for the specified address and sets it to zero. It's the same as writing 128 bytes of zeroes, but much faster.

It is important to understand that **dcbz128** doesn't zero 128 bytes starting with the address you specify. Instead, it zeroes the 128-byte aligned cache line that *contains* the specified address. For instance, a **dcbz128** to address 0x10040 would allocate and zero a cache line for bytes from 0x10000 to 0x1007f. Thus, you have to know the alignment of your data to use **dcbz128** safely.

After doing a **dcbz128,** there is a delay of about 140 cycles before writes to that cache line can proceed without stalling. This is much faster than an L2 cache miss, but for ideal performance, it is best to avoid this stall by doing the cache-line zeroing a few cache lines ahead of where you are writing.

dcbz128 can give a huge performance boost—properly used it can improve cacheable memory writing speeds by 200-300%.

Because **dcbz128** works on the L2 cache, it is illegal to do a **dcbz128** to non-cacheable memory. Doing so will cause an alignment exception.

dcbst—Data Cache Block Store

The **dcbst** instruction—available via the **__dcbst** intrinsic—tells the L2 cache to store the specified cache line to memory, while still keeping it in the cache. If the address is not cached, or if the cache line is not modified, then nothing happens.

dcbf—Data Cache Block Flush

The **dcbf** instruction—available via the **__dcbf** intrinsic—is the same as **dcbst** except that the cache line is then invalidated—it is marked as unused. If the data will not be needed again in the near future, **dcbf** can be used in order to free up space in the L2 cache, as well as to store the data to memory.

To facilitate proper use of the cache control instructions, the XDK contains four functions for optimized copying and setting of memory. These functions are **XMemSet**, **XMemSet128**, **XMemCpy**, and **XMemCpy128**. They use **dcbt** and **dcbz128** as appropriate, which in most cases gives them a significant performance advantage over **memcpy** and **memset**. These functions can only operate on cacheable memory. See the XDK documentation and the Xbox 360 Memory Copy Functions white paper for details.

Non-cacheable Writes

Non-cacheable writes go through a separate path from cacheable writes. Writes going to memory allocated with PAGE_NOCACHE are sent over the front-side bus one at a time, giving extremely poor write performance. PAGE_NOCACHE should not be used.

Writes going to memory allocated with PAGE_WRITECOMBINE are collected together using separate store-gather buffers from cacheable writes. The non-cacheable write-combined store-gather buffers are not as flexible as the cacheable store-gather buffers. Non-cacheable store-gathering requires that writes be properly aligned, with a length of four bytes or greater, and the data must be written sequentially, with no gaps, and no duplicate writes to the same address. If these rules are followed, each core can efficiently gather stores to two non-cacheable streams. If these rules are not followed, then write performance will be reduced.

Non-cacheable write-combined writes are useful for data that is destined for the GPU, since these writes avoid the need to flush data to memory after writing it. Non-cacheable write-combined writes avoid wasting space in the L2 cache on data that will never be read. They also avoid some of the L2 bottlenecks that can occur when all three cores are heavily using the L2 cache. Non-cacheable write-combined writes increase the total amount of write bandwidth available to the Xbox 360 CPU. However, non-cacheable memory should not be used for data that you will need to read, including read-modify-write updates of data, because the performance on reads will be very poor.

Load-Hit-Store

Data that was just written is not immediately available for reading. There is a delay of approximately 20+ cycles from when data is written until it can be read. This is due to the time that it takes for writes to make it through the store queue. If you read from an address before it has made it out of the store queue, then when you use the data your thread will be flushed, just as it would be on an L1 cache miss. Additionally, the instruction that used the data will reissue about 40 cycles after the write. This is called a *load-hit-store* because a load instruction hits a store that is still in flight.

> **Note** If the data is not in the L1 cache, the L1 cache miss penalty is added to the load-hit-store penalty.

There are a few common operations that can cause a load-hit-store. For example, **float**-to-**int** and **int**-to-**float** conversions often hit this penalty because data is transferred between the integer and floating-point units through memory, which necessitates reading the data immediately after it is written. Using VMX instructions for data that will need type conversions can avoid this penalty because the conversion can be done entirely in VMX registers, but this helps only if nearby processing of the data is also done in the VMX registers.

Load-hit-store penalties are common in high-speed processors. A related problem that affects many of these processors is false load-hit-store penalties. The Xbox 360 load-hit-store detection logic doesn’t compare full addresses when watching for a load-hit-store. The top twenty bits of the address are ignored. This means that if a write is followed by a read and the addresses are separated by a multiple of 4 KB—that is, if their addresses are the same modulo 4 KB—then the load-hit-store penalty will be paid.

Set Associativity

In an ideal world, it would be possible for any cache line to cache any address. However, such a cache would be too slow and complicated for any modern CPU.

A simpler design would be a direct-mapped cache. In such a scheme, a particular address is mapped to whichever cache line’s address range overlaps it, modulo the cache size. Direct-mapped caches tend to have too high a miss rate because two addresses that are separated by a multiple of the cache size cannot simultaneously be cached.

A more practical design, used by most caches, uses set associativity to control the mapping between addresses and cache lines. If you have a 32-KB cache that is four-way set associative, it is the same as having four direct-mapped 8-KB caches. These 8-KB blocks are called “ways.” With Xbox 360’s four-way 32-KB data cache, any address can end up being cached by any one of four cache lines, and up to four addresses that are separated by a multiple of 8 KB can be simultaneously cached. This lowers the number of conflict cache misses.

The instruction caches are the same size but are two-way set associative because code usually requires lower levels of associativity. However, if you run different code on the two hardware threads of one core, the different blocks of code may contend for the instruction cache and run slower.

The L2 cache is used for instructions and data and is 1-MB, eight-way set associative.

Cache Eviction Policies

The Xbox 360 caches use an LRU scheme when choosing which of the possible cache lines to evict when a new cache line is loaded. The L1 instruction caches use a true LRU scheme while the L1 data caches and L2 cache use a pseudo-LRU scheme. A true LRU always evicts the cache line that was least recently used. The pseudo-LRU scheme is a simplification of this that gives very similar performance. Using a cache line means reading from it or writing to it. If an invalid cache line is available, that cache line will always be chosen over a cache line that contains useful data.

Set Associativity Implications

The Xbox 360 L1 data caches have 256 cache lines, each 128 bytes, arranged as four direct-mapped 8-KB caches, or four sets of 64 cache lines. For example, this arrangement means that the addresses 0 KB, 8 KB, 16 KB, and 24 KB can all be cached simultaneously. However, if the code then tries to read from address 32 KB, then one of the other addresses will be removed from the cache—whichever one was used last. Normally this isn't a problem, but there are a few things that can make this happen more frequently.

For instance, if you have code that implements CPU skinning, you may have several input streams—a skin information stream, a vertex position stream, and a couple of streams of normals. If all of these streams have the same stride and all start out aligned—8 KB or greater—then the four streams will always be separated by a multiple of 8 KB and will always be competing for the same four cache lines. Worse yet, if the output stream also has the same stride and alignment, many false load-hit-store penalties will result.

Since the L1 data cache is four-way, the four streams will all fit in the cache simultaneously. There will be contention, however, whenever their alignment matches the alignment of something else—such as the palette matrix, the active region of the stack, or data being processed by the other thread on the same core. The different streams and the other data will take turns kicking each other out of the cache, and performance will plummet.

There are several solutions. One is to read from a single stream with all of the vertex information interleaved. This ensures that the streaming data uses only one cache way at a time, instead of four. Another solution is to ensure that the streams all have different strides so they don't stay offset by a multiple of 8 KB. Finally, if all of the streams have the same stride, then starting each stream with an offset of 128 bytes ensures that they are always using adjacent cache lines instead of fighting for the same cache lines.

Four-way set associativity is enough to avoid most artificial cache contention, so you normally have to worry about it only when reading from multiple arrays or when you have many data structures that are aligned to large powers of two.

Address Translation

Before a read, write, or cache-control instruction can execute the CPU needs to translate the virtual address specified by the programmer to a physical memory address. This translation is necessary even for memory that is allocated as physical memory, and even for memory that is allocated as non-cacheable. These address translations are performed by the Translation Look-aside Buffers (TLB) and by the high-speed Effective-to-Real Address Translation caches (ERATs). Prefetches that miss in the ERAT will cause a pipeline flush – the cost of loading the ERATs cannot be hidden. Prefetches that miss in the TLB will be discarded. More information about the ERATs and TLBs can be found in the Xbox 360 Memory Page Sizes white paper.

RC Machines—L2 Gateways

There are eight RC (Read and Claim) machines that control access to the L2 cache. All CPU access to the L2 cache goes through the RC machines. RC machines can be a bottleneck since they are tied up for the duration of memory reads—approximately 570 cycles to get data from main memory to the L2 cache. Reads take approximately 40 cycles to get from the L2 cache to the CPU, but the RC machines aren't tied up for most of that process. If all RC machines are busy, no L2 reads or misses can be processed.

The RC machines limit the maximum cacheable memory bandwidth of the CPU. Because there are eight RC machines, and they are tied up for approximately 570 cycles for each L2 cache miss, the L2 cache can service eight L2 cache misses every 570 cycles. That's 1024 bytes every 570 cycles, or about 5.6 GB/sec. The actual maximum read performance from a single thread is lower than that—approximately 4.4 GB/sec. If the L2 cache is constantly handling L2 cache misses, the RC machines will not be available to handle L1 cache misses, and writes to the L2 and these operations will have to wait. The RC machines will be an important bottleneck on multithreaded programs that have many cache misses.

Non-cacheable reads and writes don't use the RC machines, so they are one technique for bypassing the RC machines. While non-cacheable reads are too slow to be recommended, non-cacheable writes to write-combined memory make sense in many situations and should be considered. In fact, non-cacheable write-combined writes should be a good solution as long as the store-gathering rules are followed. Non-cacheable accesses are not coherent with cacheable accesses.

Bandwidth Implications

With good prefetching, a single thread can read eight cache lines every 610 cycles. However, if other threads are accessing memory or the L2 cache, the actual read bandwidth available to each thread will be less. A convenient rule of thumb is to assume that code that is processing a stream of data can read eight cache lines about every 1024 cycles, or one byte per clock-cycle. This means that you might as well do one clock-cycle's worth of computation for each byte that you read, since you won't run any faster if you do less computation, because you'll be memory bound.

If you can store less data and calculate more, you will usually get better performance. Shrinking your data structures—by using **short** or **char** instead of **int**, or **float16** instead of **float**—will reduce your bandwidth needs and is usually worthwhile even if you have to use a few instructions to expand the data.

Snooping

The South Bridge has the ability to "snoop" the CPU's L2 cache. That is, when the South Bridge needs to read or write memory, it checks whether the CPU's L2 cache contains data from that address. If that address is cached, the snoop logic flushes the data from the L2 cache, and from any L1 data caches that have a copy of it.

This capability means that data that is written by the CPU and consumed by the South Bridge does not need to be explicitly flushed out of the L2 cache. Synchronization is still required to make sure the writes have made it far enough to be visible to the snoop logic, but manually flushing the data to memory is not necessary.

The GPU does not have the ability to snoop the CPU's L2 cache. The GPU uses too much memory bandwidth for it to be practical to send each memory request over to the CPU. There are two ways to make data visible to the GPU. One is to write it to memory so that the GPU can see it. This is done by writing to non-cacheable memory, or by flushing the data to memory using cache control instructions. The other method is a unique Xbox 360 feature that lets the GPU stream data directly from the L2 cache.

L2 Cache Locking and L2 to GPU Streaming

One innovative feature of the Xbox 360 CPU's L2 cache is that the GPU can read directly from it. A range of the L2 cache—some number of the eight 128-KB ways—can be locked and mapped to an address range that doesn't map to physical memory. The CPU can then fill this area of the L2 cache with data for the GPU—typically vertex data—and can direct the GPU to read from that range. When the GPU is told to read from that address range, it knows to direct the read to the CPU's L2 cache instead of to memory. This capability allows the GPU to get data from the CPU without going through memory. This avoids having to do two memory bus transactions (a CPU write and then a GPU read) to get data to the GPU, and it lets the GPU read vertices from the L2 cache while simultaneously reading other data from memory—increasing the total bandwidth available to the GPU.

The L2 cache can potentially be locked for other purposes also, such as if a thread needed frequent access to a large block of data and wanted to ensure that the data was never evicted from the cache. However, cache locking should be used with care since locking and unlocking are relatively expensive, and other threads will have increased cache misses when the cache is locked due to having fewer L2 cache ways available.

Caches and Multithreading

Write-through L1 caches simplify cache coherency because all writes head to the L2 cache immediately. Additionally, the Xbox 360 L2 cache keeps track of what data is in the L1 data caches, and it will invalidate that data when necessary. The L2 cache will invalidate an L1 cache line when another processor has written to that address range. This avoids having stale data in L1. The L2 will also invalidate an L1 cache line when that cache line is evicted from L2—so that the L2 cache always contains all the data that the L1 data caches contain.

> **Note** The L2 cache will never invalidate an L1 instruction cache line because there is no need because instructions are assumed to be read-only.

While the CPU automatically keeps the caches coherent, it does not do this on an instruction-by-instruction level. There are substantial delays between when a store instruction is executed and when that data is visible to other cores. Stores will not necessarily reach the L2 cache in order; the store-gather buffers are copied to the L2 cache in an unpredictable order. You should use synchronization primitives such as critical sections to control access to shared data and to ensure that writes have completed. For more information about synchronization primitives, see the white paper Introduction to Multithreaded Programming.

If two threads write to the same address without synchronization, the results are unpredictable. There is no way to know which write will complete first. However, two threads can reliably write to different bytes in the same cache line, and the store-gather buffers will merge the writes into the L2 cache without conflict.

If two threads on different cores are reading and writing from different addresses that are on the same cache line, the behaviour is predictable, but some performance will be wasted. Each time one of the threads writes to the cache line, it will be evicted from the other thread's data cache, forcing a reload when the other thread reads from it. For this reason, threads on different cores should avoid sharing a cache line containing read-write data.

If two threads on one core are executing the same code then they will share the L1 caches well. The instruction cache will contain the instructions that they are both using, and the data cache will contain shared data. All of the shared code and data will only need to be fetched once. Even the similarities in *how* they use the L1 data cache will usually help them work efficiently together. On the other hand, if two threads on one core are executing completely different code then they may not share the caches well and they may both run slower, depending on how much of the caches they each use and how much cache thrashing results.

Because the two threads on one core share the L1 data cache they can more easily hit the limits of set associativity. If one thread reads from two 8-KB aligned addresses and the other thread reads from three 8-KB aligned addresses then only four of those addresses will fit in the L1 data cache simultaneously. Similarly, if each of the six hardware threads reads from two 128-KB aligned addresses then the twelve different addresses cannot be simultaneously cached by the L2 cache (the 1-MB eight-way L2 cache is equivalent to eight 128-KB direct mapped caches).

Cached writes to the same 64-byte aligned 64-byte block from one core are guaranteed to happen in-order or simultaneously, but writes to different 64-byte blocks may be reordered. The store gather buffers, each of which is 64-bytes in size, are not necessarily written to L2 in-order. When a timer expires or a ninth buffer is needed the oldest buffer is written to the L2 cache, which makes the order of writes unpredictable. A write to L2 may get blocked initially due to that part of the L2 cache being busy, in which case it will be retried a random time later. Similarly, read instructions can effectively be reordered on their way to L2, causing unpredictable results when reading shared data structures without proper synchronization.

Synchronization—lwarx and stwcx

Multithreaded synchronization on PowerPC is implemented using the instructions **lwarx** and **stwcx** (or the 64-bit versions, **ldarx** and **stdcx**). These two instructions are used to implement an atomic read-modify-write that can be used to implement higher-level synchronization primitives such as **InterlockedExchange** and **EnterCriticalSection**.

The **lwarx** (Load Word and Reserve Indexed) instruction loads a 32-bit integer from the specified address and creates a *reservation*. This tells the L2 cache to watch that address (actually the entire cache line) and keep track of whether any other threads write to it. Then, a **stwcx** (Store Word Conditional Indexed) is done to the same address. The **stwcx** instruction conditionally stores the data based on whether the reservation is still valid. The instruction sets the EQ bit of CR field 0 if the store happened; otherwise, it clears it. The reservation is cleared—and therefore the write doesn't happen—if another thread has written to the reserved cache line since the **lwarx** instruction.

The **lwarx** and **stwcx** instructions are sufficient for simple atomic read-modify-write operations, but more is required for full synchronization. To make sure that all writes are visible to other processors, you must issue an **lwsync** to ensure that all stores have made it to the L2 cache and to avoid read reordering. This is usually done when a lock is acquired or released, to make changes visible to other processors.

For more details, see Appendix E in the document ppc_pem.pdf, installed with the Xbox 360 Development Kit. Because of some subtle complications with using **lwarx** and **stwcx**—including some bugs in the Xbox 360 CPU's implementation of these instructions—it is strongly recommended that you use the Windows NT synchronization functions, such as **InterlockedExchange** and **EnterCriticalSection**, instead of writing your own. These functions take care of all the synchronization details, bug workarounds, and may have additional optimizations.

The important thing to understand about **lwarx** and **stwcx** is that the granularity of the reservations that they use is one cache line. Therefore, if other unrelated data is stored on the same cache line as a synchronization object, writes to that unrelated data can cause a reservation to be lost unnecessarily. Putting related data on the same cache line—such as the data protected by the synchronization object—works more efficiently, since only somebody who already owns the object will be writing to that data. However, every time a thread tries to acquire the synchronization object, it will flush the cache line from other thread's L1 caches, so synchronization objects that are heavily used should be on a cache line that is not used for anything else.

Acquiring and releasing an uncontested critical section takes about 700 clock cycles on the Xbox 360 CPU. If code must spin several times to acquire the lock, either because another thread owns the lock or because another thread is writing to the same cache line, then acquiring and releasing the critical section could take much longer.

InterlockedIncrement takes about 250 clock cycles on the Xbox 360 CPU. It's a bit faster than the critical section functions because it does not fully synchronize memory. This means that when using **InterlockedXxx** functions to gain or release access to shared memory, you must use **lwsync** to avoid reordering those reads and writes with respect to the **InterlockedXxx** operation. Just as with **EnterCriticalSection**, care must be taken not to use **InterlockedIncrement** on an address where other threads might be writing to the same cache line, or performance might suffer.

Summary and Recommendations

- Use the **__restrict** keyword wherever possible to give the compiler more flexibility to schedule around load latencies.
- Learn and use the cache control instructions whenever appropriate—**dcbt**, **dcbz128**, **dcbf**, and **dcbst**. This will be easier if your data structures are powers of two—so they fit evenly into cache lines—and your alignment relative to cache lines is known.
- Use the optimized memory setting and copying routines when appropriate: **XMemSet**/**XMemSet128** and **XMemCpy**/**XMemCpy128**.
- Consider using non-cacheable write-combined memory for output buffers (PAGE_WRITECOMBINE) to avoid wasting L2 cache space and L2 bandwidth with write-only data, but be sure to follow the guidelines for non-cacheable write combining.
- Use the smallest data types possible, and keep related data structures close together in order to reduce cache footprint and cache miss frequency. Prefer calculating data over loading it.
- Beware of load-hit-store penalties, especially on **float**-to-**int** and **int**-to-**float** conversions.
- Avoid read-modify-write (that is, avoid reading from your destination buffers) as this pulls them into the L1 cache, and may also trigger load-hit-store penalties.
- Use spatial and temporal locality on reads and writes to maximize cache and store-gathering efficiency.
- Structure your data so you can use as much data as possible from each cache line that is loaded in.