Under the Hood: Of POWER7 Processor Caches

Most of us have a mental image of modern computer systems as consisting of many processors all accessing the system's memory. The truth is, though, that processors are way too fast to wait for each memory access. In the time it takes to access memory just once, these processors can execute hundreds, if not thousands, of instructions. If you need to speed up an application, it is often far easier to remove just one slow memory access than it is to find and remove many hundreds of instructions.

To keep that processor busy - to execute your application rapidly - something faster than the system's memory needs to temporarily hold those gigabytes of data and programs accessed by the processors AND provide the needed rapid access. That's the job of the Cache, or really caches.

Your server's processor cores only access cache; they do not access memory directly. Cache is small compared to main storage, but also very fast. The outrageously fast speed at which instructions are executed on these processors occurs only when the data or instruction stream is held in the processor's cache. When the needed data is not in the cache, the processor makes a request for that data from elsewhere, while it continues on, often executing the instruction stream of other tasks. It follows that the cache design within the processor complex is critical, and as a result, its design can also get quite complex.

It is also true that the design of performance sensitive software - and there is quite a bit that is - is also dependent on appropriate use of this cache. So, if it's that important, we need to define what cache really is and what can be done to better utilize it. We'll start by outlining the concepts underlying cache and then discuss how to optimize for it.

Looking at the importance of cache another way, these processors are capable of remarkable levels of fine-grain parallelism. Just focusing on the typically executed instructions, recent processors are capable of executing up to **5 instructions per cycle**. To do so requires that the data and instruction stream reside in the core's cache. But rather than executing at this rapid rate, it is far more likely to measure programs executing slower than **5 cycles per instruction**. A large portion of this over 25X performance difference is directly attributable to the latency involved in priming the cache from slower storage. Clearly, the opportunity for performance response time and capacity improvement can be very large with knowledge of cache optimization techniques.

In order to provide suggestions to speed your applications, the following sections will provide an overview of cache-related technologies and describe related techniques for optimizing processor performance.

CEC Storage Technologies Driving Performance

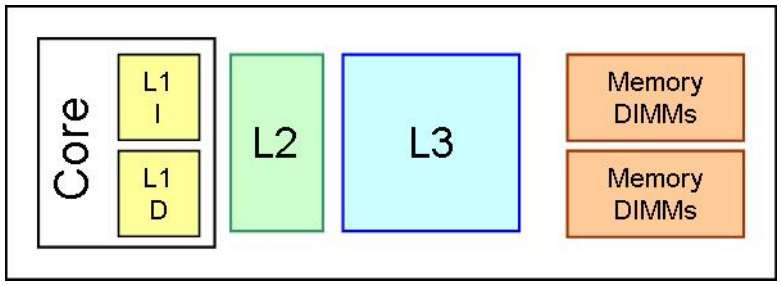
The Cache Line

The place to start is with the notion of a **cache line**. On Power-based server systems, a cache line holds the contents of an arbitrary block of storage, 128 bytes in size and on a 128-byte boundary. Your program may, for example, request one byte and it will be quickly accessed from the cache line holding the block containing that byte. But when that byte is not in the cache, main storage is often accessed and the entire 128-byte aligned 128-byte block of main storage containing it gets pulled into that processor's cache. If this same byte – or any bytes in the same block - is soon accessed again, the access will be very rapid. Fortunately, such repeated and nearby accesses really are frequent. For example, consider instruction streams, if your program executes one 4-byte instruction of a 128-byte block, the program is very likely to execute at least a few more of the 32 instructions contained in this same block.

The Cache Array

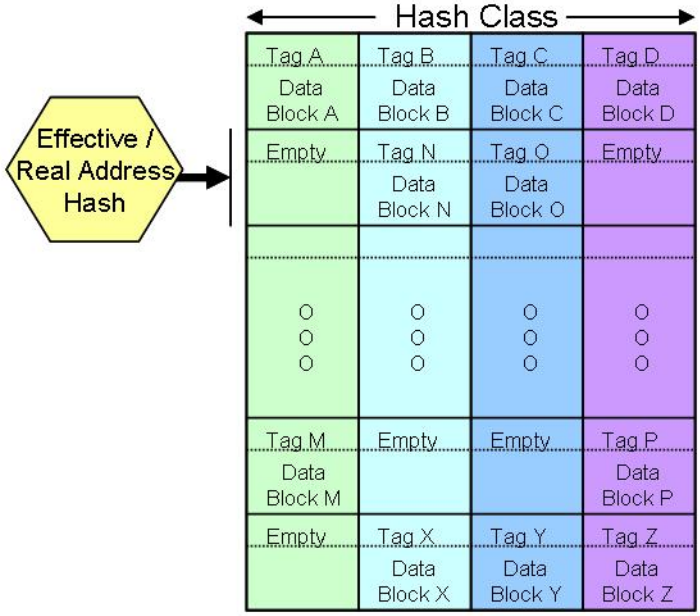
Processor cache holds sets of the most recently accessed 128-byte blocks. You can sort of think of each cache as just a bucket of these storage blocks, but actually it is organized as an array, typically a two-dimension array. As with the contents of the cache line itself, the cache array assumes that what was accessed once will be soon accessed again. That assumption tends to also dictate the size and organization of each cache. On the POWER7 processors, each of the processor chips cores (up to eight of them) have three levels of cache:

1. An L1 cache accessible at processor frequency speeds, split into separate data and instruction caches, and each being 32 Kbyte in size.
2. An L2 cache, with blocks accessible in just a few cycles, and 256 Kbytes in size, and
3. An L3 cache, with blocks accessible in over 50 cycles, and each 4 Mbytes in size. POWER7+ processors have still larger L3 caches, 10 Mbytes.



Mentioned above is the fact that the cache is organized as a two-dimension array. One dimension – its columns within a row if you like – exists partly as an aging mechanism. As each new block is brought into the cache, one block of each row needs to leave the cache; the cache line chosen is based on this aging algorithm within each row.

When accessing cache, the other dimension – the row – is based upon an index generated from a hash of the high order address bits. The L1 cache's row index is generated from the program's effective address of the byte being accessed. The L2 and L3 cache's row index is generated from the real (i.e., physical memory address) of the byte being accessed. Once the row is chosen – which chooses the "hash class" or "associativity class", the set of cache lines at this column – the cache line within this hash class is found by comparing the address which tags each one of these cache lines while the cache holds its contents.



Store-Back Cache

So far we've outlined the notion of a block of storage being "cache filled" into a cache line of a cache. Clearly, when doing store instructions, there is a need to write the contents of some cache lines back to memory as well. We'll touch on this notion next.

When a store instruction is executed on the processor core, its ultimate intent is to change the contents of main storage. But if each store were to update main storage immediately, we'd be talking about hundreds of processor cycles to complete each store instruction. To avoid this, if the target block of storage is already in the local core's cache, the store instruction quickly updates the contents of the cache line only; it does not update main storage at this time. In fact, on the POWER7 processors series, store instructions update the contents of both the L1 and L2 caches. [*Technical note: Just as with load instructions initiating cache fills in order to provide the accessed byte(s), a store instruction to a block not already in the local core's cache also initiates cache fills. Once complete, the store updates the L2.]*

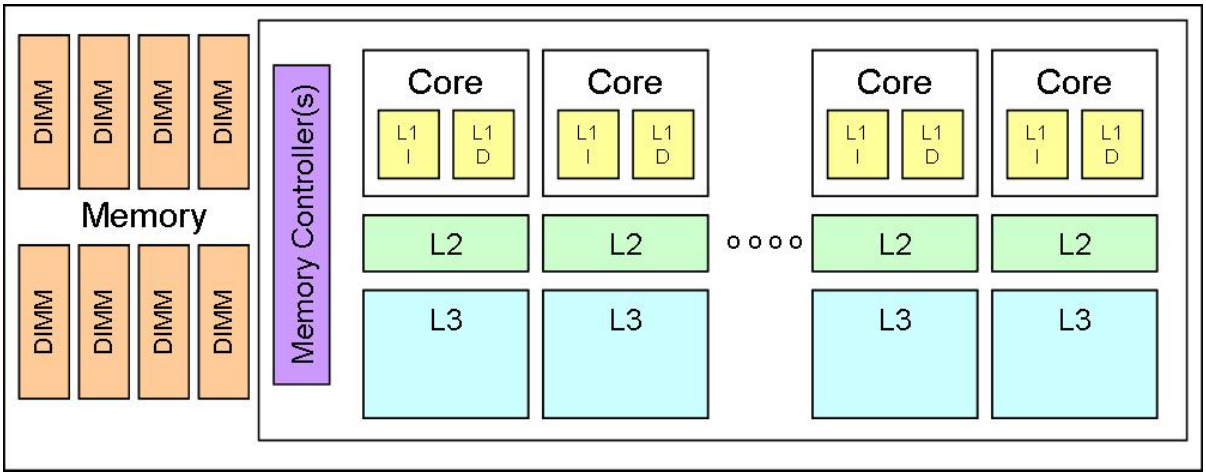
This changed storage block can remain in this processor's cache indefinitely. But again, with the cache being much smaller than main storage, any block filled into the cache also means a block is required to be removed from the cache. Since cache lines can contain the most recent (changed) state of the data – something different than the block's contents in main storage – as a cache fill occurs into a particular cache line, its current "changed" contents may need to be written back to memory. (More detail will be provided on this notion later.) From the point of view of the core owning this cache line, the process of writing back to memory is done in an asynchronous manner.

The L3 "Cast-Out" Cache

For POWER7 processors, a storage access fills a cache line of an L2 cache (and often an L1 cache line). And from there the needed data can be very quickly accessed. But the L1/L2 cache(s) are actually relatively small. [*Technical Note: The L2 of each POWER7 core only has about 2000 cache lines.*] And we'd rather like to keep such blocks residing close to the core as long as possible. So as blocks are filled into the L2 cache, replacing blocks already there, the contents of the replaced L2 are "cast-out" from there into the L3. It takes a bit longer to subsequently re-access the blocks from the L3, but it is still much faster than having to re-access the block from main storage. Such a cast-out occurs whether the content of the L2's cache line is in a changed state or not. If changed, though, as with any cache line. the state of that changed block can remain here in this L3 indefinitely. However, as with any cache, as blocks are cast into the L3, cache associatively-based aging occurs here as well, and ultimately a changed cache line will be written back into memory.

Of Multi-Processors and Cache Coherence

So far we've been discussing the concepts of L1, L2, and L3 cache associated with a single core along with some notion of a connection to main storage. Each processor chip of a POWER7 system has multiple of these cores and caches. Each processor chip also supports one or more controllers for accessing main storage. Such a single POWER7 chip's block diagram is as shown below.

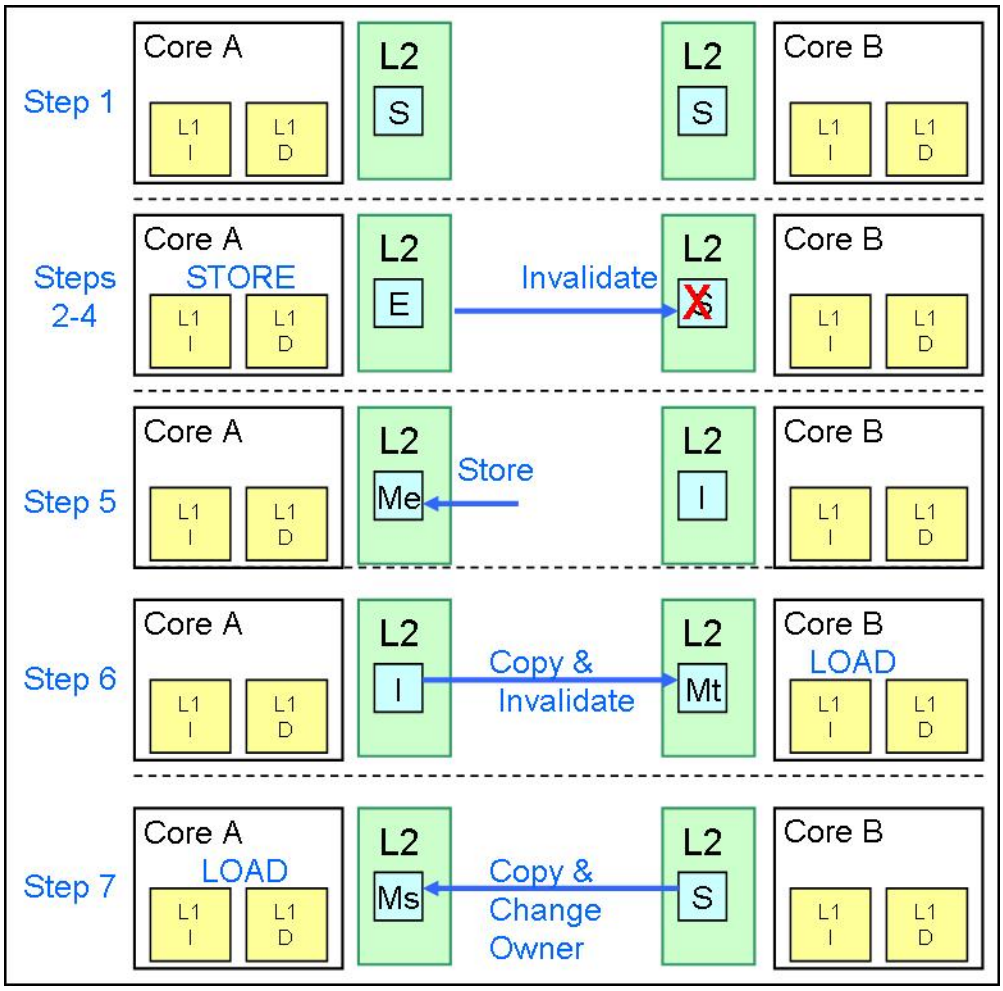


This is a Symmetric Multi-processor (SMP). Within such multi-core and multi-chip systems, all memory is accessible from all of the cores, no matter the location of the core or memory. In addition, all cache is what is called "coherent"; a cache fill from any core in the whole of the system is able to find the most recent changed block of storage, even if the block exists in another core's cache. The cache exists, but the hardware maintains the illusion for the software that all accesses are from and to main storage.

One key to the notion of "cache coherence" within an SMP is that the cache(s) of multiple cores can hold the contents of the same 128-byte memory block. Such sharing is completely automatic when multiple tasks on different cores are addressing the same object; they'll all get a copy the same data as it exists in physical memory. During such periods of data sharing, each core can also fill its cache from another core's cache, allowing even these cache misses to be done quite rapidly. Such cache lines, all containing the same storage block, are tagged as being "shared".

With that in mind, let's now have one of the cores execute a store instruction into this shared data block. The programs that the tasks are executing don't know anything about this sharing feature of cache – in fact they know nothing about cache – so any other task subsequently accessing that changed data block is going to expect to see the change, seeing it just as though the store instruction had changed physical memory. Picture it like this (see figure below):

1. Core A and Core B each have a copy of the same block of memory in their cache.
2. Core A executes a store instruction to this block, with the intent that the change stays in its cache alone.
3. Core B's next access of that data expects to see the change.
4. To ensure that Core B accessing that same variable does not see its old state, the hardware invalidates Core B's cache line containing the memory block.
5. Core A, now having an exclusive version of the memory block, alters that memory block in its cache per the store instruction.
6. Core B, subsequently accessing the memory block, requests a copy of that block from Core A. It happens on such a case, Core B gets an exclusive copy.
7. Core A, subsequently accessing the same memory block, gets a copy back from Core B.



Notice the key to this? Any change to a storage block means first having an exclusive copy of the current state of that block. Accessed frequently enough, such changed blocks can move between the caches of multiple cores without actually making its way back into memory proper for quite a while. That fact actually tells you a lot about caches in a multi-processor. It tells you that

- In order to make a change to a block of memory, only one core's cache can have that block of storage in its cache. This implies that part of such store processing can require an invalidation signal being sent to other cores. This requires some time to request and then know that the local core's cache line has the only copy.
- When other cores want to see – and perhaps also change - that changed block, the other cores must do a cache fill from the cache of the core where the changed block exists. This means that, rather than filling a cache line from memory, the cache fill is from another core's cache. This inter-core cache fill is faster than an access from memory, but not as fast as a cache fill from a core-local L3.

Notice again that it is the hardware that is managing this, just as though all accesses were from memory proper. And that it just exactly what the programming model used by all software within an SMP is expecting. It just works. But we'll also be seeing later that if your program design is at least indirectly aware of what is actually going on, it can also execute faster and system throughput can also improve.

[*Technical note: In order to maintain a program's illusion that all accesses are made to/from memory, there is a complex hardware architecture for keeping track of the various states that a cache line can be in. For example, consider that even though a block of storage has been changed relative to its contents in memory proper, that block might also reside in multiple cores' cache; even so, one core remains responsible for writing its contents back to memory. There is a considerable amount of inter-core traffic*

required to maintain this data block state information. For more see http://en.wikipedia.org/wiki/Cache_coherence]

Recapping what you have learned so far about cache theory:

1. A data or instruction stream access from a core's L1 cache can be very rapid, allowing the instruction stream to continue executing at its high frequency without pausing.
2. In the event of an L1 cache miss on a Load or Branch instruction, the core looks into its local L2, which on POWER7 is less than 10 processor cycles away, potentially filling the L1 cache from there.
3. In the event of an L1 cache miss on a Store instruction, the stored data is stored into the L2 cache line (if present), without filling the L1. If the data block is also in the L1 cache, the stored data updates both the L1 and L2 cache lines. Although it may take a few cycles to update the L2 cache line, the store instruction's execution does not delay; the store is actually done into a store queue in the core whose content is ultimately staged for writing into the L2 cache. [*Technical note: If the store queue is full, store instruction processing pauses pending store queue entries becoming available after their contents is written to the L2 cache.]*
4. In the event of an L2 cache miss, you can think of instruction processing as pausing until the cache fill completes of the needed data into an L2 cache line. But it's a bit more complex than that. For example, a cache fill on a store still allows the store to complete, but the data stored in the store queue gets held there until the cache fill is complete. For now, consider this as simply delaying processing until the L2 cache miss is complete.
5. If the needed storage block is present in this core's L3, a cache fill into the local L2 from the local L3 is initiated. This is roughly 25 processor cycles away.
6. If the needed storage block is not present within the reference core's cache (including a local L3 miss as well), a request to the location where the block does reside - checking first other chip-local core's cache and then memory - is initiated.
7. If the needed storage block is found in some other core's cache, that core initiates an access from its cache, and passes it onto an internal chip's bus as though from memory. Such a cache fill is roughly 150 processor cycles away.
8. If not found in a core's cache, a request for this block in memory is initiated. Such a cache fill takes roughly 400 processor cycles.
9. Just to complete the picture, the needed data might not even reside in memory when accessed. This results in a page fault interrupt; the page (typically 4096 bytes in size) containing the 128-byte block needs to be brought into memory. Where the 128-byte cache fill from memory takes well less than a microsecond, the page fault processing typically takes a handful of milliseconds to complete. In this context, even main storage is managed by OS software a bit like a cache.

Now, recall that in the ideal there would be no delay in instruction processing due to cache misses; that ideal requires all L1 cache accesses – both instruction and data – to succeed. But failing on hitting on the core's L1 caches, and then also on the core's L2 and L3 caches, your next best option is for the data to happen to reside in some cache nearby this core. We'll cover this more shortly. But first, two more items of prelude.

L3's "Lateral Cast-Out"

We had outlined previously how the contents of a core's L2 cache lines get cast-out into the same core's larger and slower L3 cache. POWER7's chips also support a notion called "lateral cast-out" where the contents of data also cast out of a core's L3 can be written into another core's L3. If a core has a cache miss on its own L3, it can often find the needed data block in another local core's L3. This has the useful effect of slightly increasing the length of time that a storage block gets to stay in a chip's cache, providing a performance boost as a result.

This becomes more useful when one or more cores on a chip are not doing much processing (and adding much cache pressure to their own L3), implying that little is being written into a Core B's own L3 by

Core B's L2 cast-outs. As a result, the hardware detecting the aging cache lines in Core B's L3 can allow cast outs from Core A's L3 to be temporarily written into Core B's L3.

The TurboCore Effect: This basic notion is what TurboCore Mode is based upon. A chip, which nominally has eight functional cores – and so eight functional L3 caches – is set up in TurboCore mode to use only four of the eight cores, maintaining the use of all eight L3 caches. The now inactive four core's L3 act as the cast-out targets (16 Mbytes in total) for the still active four cores. It's like having an extra, slightly slower and larger, level of the cache available in TurboCore mode.

Simultaneous Multi-Threading (SMT) and Cache Pressure

Recall the sizes of a POWER7's L1s (32K), L2 (256K), and L3 (4M). As you think about a single application executed by a single task, this can seem like a lot of cache. POWER7 cores, though, are also capable of executing the instruction stream of up to four tasks, each with their own cache state. Whether two, three, or four tasks, each has their own working set of data. These additional tasks with their additional data are sharing the cache, cache otherwise - and occasionally - used alone by one task. When these tasks happen to truly share data, all the better, but such might not frequently occur. As compared to a single task executing on a core, think of the core's cache as perceived by each task as being on average ¼ the size when there are four tasks executing, 1/3 the size with three tasks, and ½ the size with two tasks executing. You can see that there can be some considerable pressure for a core's cache as each additional task is added to the core.

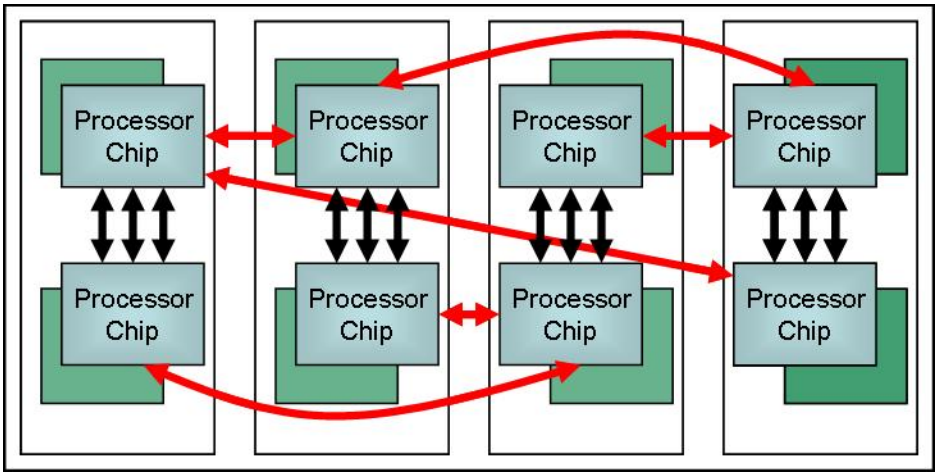
The concept to come away with here is that this cache contention has a way of slowing all tasks on a core when additional tasks are added to a core. As a result, when a partition is normally concurrently executing a significant number of tasks (something which can be perceived as executing with relatively high utilization) – when multiple tasks are executing on a core – it becomes even more important to be thinking in terms of minimizing the number of 128-byte blocks of data and instruction stream concurrently used by each task.

Also observe, though, that the concept of SMT exists largely because the core is often waiting on cache misses. With SMT, in the mean time, the core can allow another task's instructions full use of a core when one or more other tasks are waiting on cache misses. In terms of capacity, it is better to have SMT than not. But that tradeoff only works up to a point. The core can concurrently handle only so many cache fills. And, after all, even with SMT, the POWER7 core can only execute instructions at a maximum rate of roughly five instructions per cycle.

Multi-Chip SMPs and the Cache Effects

A multi-core but single chip-based system can make for a sizeable SMP, but far larger SMPs can be built from multiple such chips as in the following figure. This POWER7-based system happens to support up to 64 cores; other topologies support up to 256 cores. It happens that in these topologies, a portion of the system's main storage can be found behind every processor chip.

A key point to keep in mind here is that this entire multi-chip complex is a single cache-coherent SMP; all of main storage in this entire system is accessible from all cores on any chip. Being cache-coherent, it follows that the most recent updates of data originally in main storage but now residing in some core's cache is also accessible from any core on any chip. The cache is just a transparent extension of main storage; its contents can not be addressed as being separate from main storage. Again, any task executing on any core can access its needed blocks of instruction stream and data, no matter where they reside – cache or memory – throughout this entire complex.
(*Technical Note: The hardware allows full access to all storage from all cores. But this is true only if the software is allowed to generate a real address into that storage. Higher level addressing protection techniques, of which there are many, ensure isolation to only that storage – to those portions of real address space - which that software is allowed access.)*



Another point to notice here, though, is that from a performance point of view this system is not just a series of processors hung on a common link/switch to a whole set of memory DIMMs. From the point of view of some reference core on some reference chip, that core can functionally access any main storage, but an access to the portion of main storage directly attached to the chip – a "local" access - is faster than an access to any other chip's memory. To make an access to "remote" memory means requesting another chip – perhaps via still other chip(s) - to actually drive the access on behalf of the reference core.

Further, every single one of these chips have multiple buses to their local memory DIMMs. This starts to add up to some massive concurrently accessible bandwidth to the memory of this system. Rather than each request having to wait to use the linkage to memory, this approach provides considerable parallelism for concurrent memory accesses.

Both the relative access latency and the massive bandwidth are the basic attributes of a NUMA-based (Non-Uniform Memory Access, http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access) system. All memory is accessible quite quickly, but some memory is accessible more quickly. Where an access to local memory can take roughly 400 processor cycles, consider a remote access as adding roughly 100-120 processor cycles to that for each chip hop required to make the access. In POWER7, some chips are directly linked, others require two or three hops.

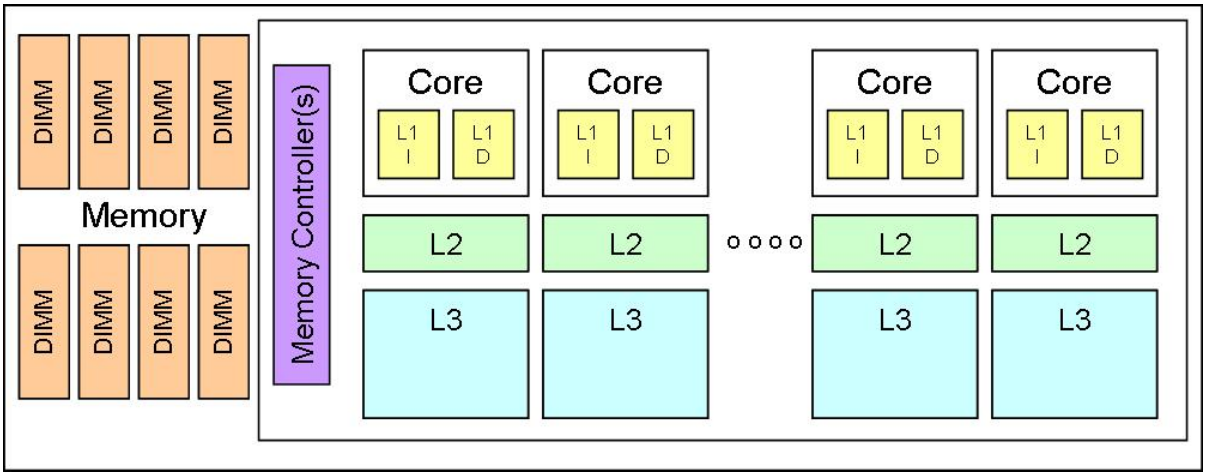
These “local” and “remote” accesses are largely just the cache fills and cache store backs that we’ve been referring to all along. The only additional concept is that some cache fills have different latencies – something that you have seen is true for even single chip systems – but here also depending upon the chip where the accessed block of storage happens to reside. Don’t worry, IBM i OS and hypervisor software know of this architecture and now so do you.

So, in order to speed application execution and boost system capacity, we’d all like to make the local accesses more probable than any form of remote access. And there are some straightforward steps used to achieve this:

1. Allocate storage from the main storage local to core(s) where the work is likely to execute and
2. Assign work to those cores local to the location where the data being accessed is likely to reside.
3. Allow tasks sharing data to execute on cores relatively close to each other.

Fortunately, both the hypervisor and the partition’s OS attempt to do just that. Doing so does not necessarily guarantee only local accesses (that’s simply not required), but it does have the effect of improving the probability of local memory access.

There is another key aspect to this as well. Recall that in the single-chip SMP figure (reproduced below), a reference core can access the cache of any core on that chip, this being done to either access the most recent state of a changed block of storage or to simply speed the access of – say – a shared block of storage. Within these multi-chip SMPs we’ve increased the number of cores, with those cores now residing on multiple chips, but the basic required capability still remains. All cached storage of this entire SMP is coherently maintained, whether within a single partition or across multiple partitions. Any reference core in this entire system can see – even if still in a cache of another chip’s core - changes made to any block of memory. Similarly, a reference core can alter a block of storage – making it a changed/exclusive version in the reference core’s cache – and expect that no other core will see stale data.



The previous paragraph outlines the functional capability, but – relative to performance - the chip boundary introduces some cached storage access latency as well. Recall that a cache fill from a core local to the same chip takes roughly 130 processor cycles. Just as with the remote memory accesses, a similar cache fill from a core on a remote chip requires roughly 100-120 additional processor cycles to cross chip boundaries as well.

I/O and Processor Cache Effects

We all tend to forget that processors are not the only entities that access a system's main storage. I/O devices and coprocessors do so as well. And at the time that they do, the block(s) of storage associated with data that the device is reading from main storage or writing into main storage might happen to reside in some core's cache line. The reads done by these devices must see the same state of the data as a processor requesting a cache fill.

So a few observations:

1. If the I/O DMA write into main storage is altering an entire 128-byte aligned 128-byte block, all that the I/O DMA needs to do is first arrange for the invalidation of any core's cache line containing this block.
2. If the I/O DMA write is partially altering a 128-byte block which happens to reside in a processor's cache …..
    a. If it happens to exist in the cache in an unchanged state, the cache line in the core is merely invalidated.
    b. If it resides in a core's cache in a changed state, the changed cache lines contents is written to memory before the DMA write can occur.
3. If the I/O DMA read is of a block residing in a core's cache in a changed state (relative to main storage), the read of all or a portion of that block must come from the core's cache.

An I/O DMA on POWER7 (and some of its predecessors) enters the CEC through a GX++ controller on each of the processor chips. These share the same SMP fabric as all of the processor cores and memory controllers. So, in a manner of speaking, these controllers are acting as surrogates into the SMP fabric in much the same way as processor cores.

Why does this matter? Just as with a processor core's cache fill latency being partly a function of the relative distance of the core from the storage block, I/O latencies are – at least partially - as well. For example, given a GX++ (I/O) controller reading a storage block which just happens to then reside in the core of a cache on the same chip, the data to be DMAed can be rapidly delivered to the controller from that core's cache. At the far other end of latency, that GX++ controller might also reside on a processor chip multiple hops away from the chip whose attached memory is actually being accessed. This is the SMP's storage access component of the I/O latency. But you also know that I/O latencies are a function of a lot more than just the storage access within the SMP. So, if you are after every last microsecond of I/O response time, and looking to maximize the capacity of the SMP proper, you might want to also keep this component of it in mind.

Optimization Tips, Techniques, and Best Practices

Data Packaging

128 bytes on a 128-byte boundary … That is what the processor always requests in the event of a cache miss. It does not matter whether the application is accessing just one byte from that block or all 128, the hardware is going to do the same thing either way. The same is true with the instruction stream; a 128-byte block might contain only one 4-byte instruction that happens to execute or it might contain 32 executed instructions (i.e., 32 * 4 = 128 bytes). Either way, the hardware is going to pull the same 128-byte block into the cache and then execute whatever instructions residing there that it needs to execute. Where in the ideal you'd want to have the hardware do one cache fill and have every byte there accessed, it would be unfortunate to have to instead have 128 bytes reside in 128 distinct blocks and then have to wait on 128 cache fills. Similarly, given a program needs to execute exactly the same number of instructions no matter how the program's instructions were packaged, it would be similarly unfortunate if processing of those instructions required quite a few cache fills – rather than one - to bring those instructions into the L1 cache and so consumed a large fraction of the processing time to do it.

Avoiding the processing delays – each taking hundreds of processing cycles – by more densely packaging data and instruction streams is what we are addressing here.

The Program Stack

Consider your thread's program stack for example. As your program does some number of routine calls and returns, the stack frames used are repeatedly working their way up and down over the same effective address space. This seems like an ideal situation; this means that the same real address space is repeatedly being used by different stack frames. This also means that it is using the very same cache lines as long as this stack's task continues executing on a given core. But is it really? Are these stack frames using the same physical storage? In the most general sense, yes. But suppose a stack frame allocated a large buffer (e.g., an exception handler object) which typically does not get accessed. In doing so the set of cache lines corresponding to that real address space are also not being used (and so may age out of the cache). Subsequent stack frames may start using effective and real address space corresponding to memory not already in the cache, meaning that such will require additional cache fills.

- **Tip:** Think about what gets allocated in each stack frame as a routine gets called. If that storage is sizeable and not likely to be accessed, consider breaking that storage out as part of another routine's stack frame or even heap.

We just noted that these same cache lines can get reused by the stack frames when the task stays executing on a core. But now let's introduce short waits to this task, resulting in this task leaving a core and then being dispatched again. Ideally, the task returns to the same core, but in the mean time, another task might have been using that core. (With SMT, other tasks could have been concurrently using that core.) So where are those stack frames now, and what will it take to pull their blocks of storage back into the cache?

- **Tip:** The fewer 128-byte storage blocks being used by the stack, the fewer there are to return to the cache, the faster that task returns to processing at full speed.
- **Tip:** Consider inlining of routines to avoid the creation of new stack frames.

The Program Heap

Consider your program's heap. When it allocates an object (e.g., malloc, new), from where in physical memory did the heap manager allocate that storage? If using a vanilla heap manager, your program only knows that it came somewhere from that heap. So, how is the heap manager choosing the storage it is providing and what is it doing with the heap allocations that your program is freeing? You know now

that the cache is just holding the contents of the most recently accessed 128-byte blocks of physical memory.

- Is the heap manager doing any recently accessed storage reuse?
- If the heap allocation is small enough, did it allocate from the same 128-byte block (or the same page) as a recent heap allocation?
- If frequently allocated and freed, does a subsequent heap allocation use the same storage as was recently freed?

Still thinking about heap storage, what about the organization of your objects built there? Certainly one can create nice but complex objects based on pointers (even in Java) to heap allocated for more primitive objects, but does each reference to its more primitive parts result in a cache miss? (Or as problematic, an access into to another page?) An object's organization can make a great deal of sense without even thinking about the cache, but can it be reorganized and still be equally maintainable by also thinking about its organization from a 128-byte block point of view?

- **Tip:** Instead of making many heap allocations from essentially arbitrary locations in effective address space to construct a complex object, can that same object be allocated from contiguous effective address space?

Consider a linked list of objects, with each element potentially having been allocated from the heap. Where does each element of the list really reside in effective and real address space?

- **Tip:** Can the linked list and appropriate search criteria instead be allocated as an array with each entry addressing the actual location of the object? The intent here is to avoid scanning through a link list which is effectively also a set of cache lines/misses and replace what gets actually accessed with just a few contiguous cache lines.
- **Tip:** Consider using a structure which is a linked list of array entries, where the array entries now represent a set of the objects which would have otherwise been the objects comprising the linked list; if the objects had been small enough, multiple of them are now packed into single or contiguous 128-byte blocks of memory as opposed to each being in its own cache line.

Instruction Packaging

We alluded above to instruction stream cache fills. For a certain number of instructions of a program that will be executed, how many instruction stream cache fills were needed to make that happen? We want to minimize that. Package the instructions most likely to be executed within the fewest number of 128-byte blocks – and the fewest number of pages - as is reasonable. Take a look at most any source code and then realize that the compiler will generate instructions on behalf of all of it, often roughly corresponding to the way that the source code itself is laid out. Now take another look and realize that a fair amount of that code is less likely to be executed; a fair amount of it is getting skipped. But also notice that this less frequently executed code's instructions are still packaged in the same 128-byte, 128-byte aligned blocks along with instructions which are being executed. As a result, your program needed more instruction cache fills than were really needed.

- **Tip:** What you would like to do is somehow influence the compiler to package the infrequently executed instructions at some distance from the instructions which are likely to execute. There is a way via profiling of an executing program. An example of such a tool is FDPR (Feedback Directed Program Restructuring), documentation of which can be found at
  https://www.research.ibm.com/haifa/projects/systems/cot/fdpr/
  or for IBM i at
  http://www.systeminetwork.com/article/other-languages/topics-in-ile-optimization-part-1-65827

Or consider subroutines and their size. Picture now how that is really being packaged. At each call site there are instructions to marshal parameters, the call itself, and then code to handle the returned results. At the called routine you have a prolog, epilog, and, of course, the routine itself. All that takes space (cache lines). So, how large was that routine, especially if it did not have a need for a prolog and epilog?

- **Tip:** Would the overall size have been improved with inlining of the routine?

But more important in this context, would inlining decrease the number of cache lines needed to execute this routine? No marshalling, calling, prolog, and epilog, and the instructions supporting the routine's functions are packed inline with the instructions already at the call site. The hardware's instruction execution just flows right into the instructions of what would have otherwise been an external subroutine call. The hardware does not necessarily know how large a program really is; it only knows how many cache lines were filled and are being currently consumed to support the instructions executed by this thread on this core.

We can continue on with many more examples. But the key thought process is to think about the relationship of data and instruction stream being accessed to its allocation in effective and real address space, and then realize that what the processor really sees is just accesses of 128-byte blocks.

- **The BIG Tip:** The intent is to pack that data which actually gets frequently accessed into fewer 128-byte blocks, thereby deceasing the number of cache fills, speeding the time required to load same, and allow the remainder of the cache(s) to be used to continue to hold other data and instruction stream.

Multi-Threading Data Packaging

In the previous section we discussed packaging of data largely accessed by a single task. But a fair amount of storage is being frequently modified by multiple threads. The rules for such multi-threaded modified-data sharing are a bit different.

To explain, let's have a variable – say an 32-bit atomic counter – be rapidly advanced by quite a few threads, each executing on a different core. As with all other updates, the atomic counter resides within a 128-byte block and any change means that at some moment in time only one core's cache contains an exclusive copy of that block. Certainly, that storage block is being moved from one core's cache to another. Especially if those cores happen to reside on a single chip, such rapid updates are not necessarily a problem. Only the thread making the update sees the cache miss delay.

But now let's add a wrinkle. Let's have that same 128-byte block (one containing the relatively rapidly advanced atomic counter) also contain variables which are much more frequently read and much less frequently modified. The frequent reads would normally allow the block to be cache filled into multiple cores caches with the contents of that block repeatedly accessed from there. Let's say that soon thereafter the atomic update is attempted. Initially the core doing the atomic update also pulls a copy of the same storage block into a Core A's cache; then the actual update is attempted, requiring that an invalidation request be sent to all other cores to remove that block from their cache. Once invalidated elsewhere and held exclusively in Core A's cache, the update occurs. Being also frequently read, this update is followed very quickly thereafter by threads on other cores, continuing their access of their read-mostly variables. This reload, like all cache misses, takes time. But notice that this resulting delay is for <u>all</u> of these threads after each such update. Now also picture this set of operations being done frequently;

1. the block gets replicated across quite a few cores,
2. then it gets invalidated on all and updated on one,
3. with the above being done repeatedly.

We've got a lot of time being spent on cores just waiting on cache fills to reload the read-mostly variables.

This effect is called "False Sharing" if you would like to look for further information….
http://en.wikipedia.org/wiki/False_sharing

The effects on SMP scalability from such as this can be serious. When this occurs, it is not uncommon for a design which should have scaled up linearly with the number of threads to actually produce results which are well less than linear. In the extreme, increasing the number of threads produces less throughput with each additional thread.

So let's consider an optimizing change with this effect in mind……

Let's move all of the frequently accessed read-mostly variables into a 128-byte block different from the atomic variable. Having done that, the atomic variable updates still result in the block moving between cores. But the block containing the read-mostly variables is not. The frequent reload of this block ceases and only the one core actually doing each atomic update perceives the cache miss delay. This is much cleaner.

What is particularly sinister about False Sharing is that all that you see in your code is a structure of variables. It is not until you look at it again as overlaid by a set of 128-byte blocks and realize the relationship of this to the cache and SMPs that the performance effects become more obvious.
And worse … Recalling that small heap allocations (e.g., two allocations, each of size 64 bytes), which also means multiple independent small objects, could reside in the same 128-byte block. If both are intended as thread-local, but nonetheless rapidly updated, the heap allocations are falsely sharing the same storage block.

Here is another interesting False Sharing variation wherein a number of threads happen to be rapidly advancing their own private counter in an array which happens to be in the same cache line.
http://www.drdobbs.com/parallel/217500206
It happens that an easy solution to this would have been to keep the counters as a thread-local variable, and then update the array just once at the end of the operation.

A variation on this is based on temporal updates. Suppose that you have a routine, the routine updating a set of variables in a single 128-byte block, but taking a while in doing so; the updates are spread out over time. Let's further have a large number of tasks concurrently executing this routine, accessing and updating this same 128-byte block. While doing so, with each access, this previously modified/exclusively held block is copied into another core's cache, potentially tagged as being shared. Each update then also again makes the block exclusive to enable the modification (i.e. repeatedly cache filled as shared and then invalidated elsewhere for the modification). This continues throughout the period of concurrent updates of this block. (Picture the cores fighting over the block to get their own updates done.) Suppose instead, though, that the period of updating the block is decreased considerably, that the updates were made in a short burst. Rather than each update producing an invalidation and refill by all, by grouping the updates in time, the grouped updates have the effect of often invalidating just once. Further, because the updates are done so quickly, the probability of any other core requesting the block decreases and the updater thread makes all of its updates without needing to do repeated cache fill (with intended modify) itself.

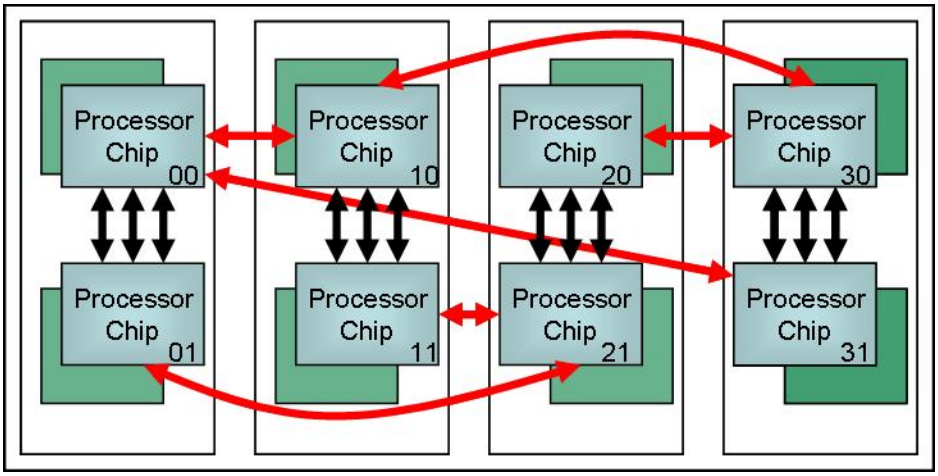
Still another approach to managing a single entity like an atomic variable is – perhaps oddly – not to have one, but still have one. To explain, suppose that your atomic and rapidly changing variable is also only queried for its results only occasionally. It is going to be read and updated frequently, but with its current results "used" considerably less frequently. You know that each 128-byte block containing a variable can stay in a core's cache in a changed state even indefinitely. In the ideal, you'd also like such variables to remain there as well. Perhaps only slightly less ideal, you'd just like that block to be moving less frequently between core's caches, potentially being updated multiple times while in the same core's cache. So consider breaking up the variable to have separate instances in different 128-byte blocks. Each updater is aware of the instance of its variable. But more importantly, the less frequent query of the results is aware of all of these locations, pulling each of the associated cache lines into its cache only for the period needed to see the aggregated variable's current state. True atomicity can be maintained by having the query code first flag its intent to read all to the actual modified variables.

Memory Affinity Management within Partitions

You've seen that as long as a partition's core and memory resources have been assigned from a single processor chip, from the point of view of this partition, the multi-chip system in which it resides is not much like NUMA at all. There is then also no need for this effect to be managed. Good. That, though, had required the hypervisor to have placed this partition to reside on this single chip and that, in turn, required that there be core and memory resources there sufficient to meet the needs of this partition.

[*Technical Note: The resources required by other partitions sharing the same system might have the effect of forcing what could otherwise have been a single-chip partition to span multiple chips. Taking care with the definitions of all partitions and/or having excess processor core and memory resource to allow clean packaging can minimize this effect*]

For the remainder of this section, let's have a partition with core and memory resources residing on multiple chips, say on chips 00 and 01 of the following figure. We'll start with dedicated-processor partitions and then add some caveats about shared-processor partitions. We'll also be making some observations concerning those partitions which have cores on chips with no local memory and partition memory behind chips that don't also have any cores assigned there.



As outlined earlier, the basic rules associated with affinity management are
1. Allocate storage from the main storage local to core(s) where the work is likely to execute and
2. Assign work to those cores local to the location where the data being accessed is likely to reside.

But, realizing that cache coherency is also an affinity issue, a third rule adds
3. Attempt to have tasks sharing common data reside as close together as possible, ideally on the same processor chip.

Within an IBM i partition, the Memory Affinity Management support is built on a notion called a Home Node. As each new task is created – say at the time that a single-threaded job is initiated – that task is assigned a Home Node ID, identifying which chip's cores – and so memory behind it – that this task would prefer to use. When that task becomes dispatchable, the OS' Task Dispatcher will attempt to assign that task to a "processor" (i.e., an SMT hardware thread) of a core of the chip identified by the Home Node ID.

[*Technical Note: The Home Node ID is a logical, not physical, concept since the number and location and each node can change over time.*]

Similarly, when this task needs to allocate a page in main storage – say as part of a simple DASD page read – the IBM i Main Storage Manager attempts to allocate a page from this partition's memory residing

behind this Home Node's chip. These are the preferred locations, but failing there, any core or memory (the closer the better) is used.

Each task's Home Node ID is generally assigned – and potentially altered later – based on the partition's current notion of balance. Different nodes might have differing numbers of cores; as a result, balance is based on the available capacity in each node. As work comes and goes, as some tasks are found to consume more capacity than others, Home Node IDs are slowly rebalanced. This balance also includes an assumption that the partition has enough memory available behind each node; ideally, nodal memory available is proportional with the node's compute capacity. Notice that if a node does not happen to have any memory assigned there, the node's cores don't tend to be included in determining balance for Home Nodes; the IBM i Task dispatcher will use the extra cores, but only when capacity needs dictate that it must.

These are the general concepts, but you can produce variations on this theme. For example,

1. By default, a task might prefer to be dispatched to its Home Node (or chips in the same drawer/book as this Home Node), but any chip's cores are acceptable. This can be strengthened on a partition or job basis to say that this task must use its own Home Node.
2. It may be that the threads of a multi-threaded job are cooperating and sharing the same data and instruction stream. By default, the Memory Affinity support does not make such a distinction, but a partition or job can say that it wants all of these threads to share the same Home Node ID.
3. It may also be that a set of jobs/tasks are cooperating, also sharing the same data. These too can arrange to share the same Home Node ID by having the jobs share the same Routing Entry associated with the same subsystem.
4. An odd variation on this theme is done via Workload Groups (a.k.a., Workload Capping). The workload capping function can be used to limit the processing capacity of a workload to a subset of processor cores in a partition. Jobs can then be assigned to the workload capping group either via a subsystem description or individually via a CHGJOB command.

Memory Affinity Management by Hypervisor

The hypervisor is also well aware of the topology of these systems and the relative storage access latencies. For small enough partitions, the hypervisor does indeed attempt to place a dedicated-processor partition's specified number of cores and memory in some nice manner. Even for shared-processor partitions where the partition's virtual processors could potentially execute anywhere, the hypervisor does attempt to allocate such a partition's virtual processors on cores close to where the partition's memory also resides.

But, for the hypervisor, it is a bit like a puzzle, attempting to nicely place every partitions cores, virtual processors and memory. And sometimes the pieces can be made to fit truly nicely, sometimes not. This is both a function of the specified partition resources and the physical nodal topology of these systems. Now that you have seen the topology of these systems, you can see the challenge for the hypervisor. You can also see that making adjustments to these resource specifications can have the effect of allowing any given partition to be nicely packaged. But you can also see that any non-optimal placement of one can have the effect of rippling into less than optimal placement of others.