Department of Computer Science

Memory Consistency in the Haswell multi-core architecture

EDT621 - Datorarkitektur med operativsystem

Robin Skafte - dat14rsk@student.lu.se

8 december 2015

LUND UNIVERSITY

Table of Contents

1. Abstract

As the amount of cores on each processor increases, so does the complexity of implementing a efficient memory structure. Most of todays System-on-chip processors supports shared memory, this means that every thread in every core may read and/or write to the same specific memory adress location. When more than one unit can modify a value on the same adress location problems may arise with inconsistency is the memory hierarchy, which could lead to all kinds of incorrect execution. In order to keep the memory consistent two important disciplines are usually implemented: a *memory consistency model* and a *cache coherence protocol*.

Today, there are many diferent types of memory models and these are often categorised under *release* or *strong* memory models (like the x86 TSO Memory Model, implemented in the Haswell architecture).

In Chapter 2 the concepts of memory consistency and cache coherence will be explained. Chapter 3 deals the basics of the Haswell memory architecture and possible data conflicts that may arise in the processor system. Memory consistency will be further discussed in Chapter 4, where the difference between *sequential consistency* and the *x86 TSO* memory model will be explained. In Chapter 5, the implementation of the cache coherence protocol, MESIF, in the x86 microarchitecture is illustrated.

2. Introduction to Memory Consistency and Coherence

The increasing level of intregration, in the context of semiconductors and many-core SoC-processor's, introduce various valueable properties such as: better performance (faster execution time) but also a lower cost due to lower power consumption. While new designs can improve the reliability and performance in a given system, it can cause problems with inconsistency and memory stalls. Memory corectnsess are often separated into two sub-issues: *coherence* and *consistency*.

2.1 Memory Consistency and Shared Memory Models

A shared memory system is defined by each processor core being able to read and write to a shared memory adress space. Writing correct and effective parallel programs requires a specification of memory semantics, referred to as a *memory consistency model*. Memory models are visible to both software and hardware, in a compiler and in lock-up free caches processor may reorder memory accesses if it can determine it will not affect the program correctness. In a lock-up free cache compilers may also prefetch data into processor registers, which is important when optimizing memory systems.

The most intuitive consistency model is ***sequential consistency***, this memory model were used in Intel's early unicore processors. From the view of sequential consistency, multiprocessors has no cache meories, and all memory accesses goes to memory, thus each program order from one processor can be regarded as *not* overlapping. Sequential consistency requires that memory operations appers to execute atomically, where the order is consistent with the program order of each process and where a read returns the value deposited by the latest write to the requested memory adress location. In order of a sequientially consistenct memory working correctly, the cache memories needs to *not* be lock-up free caches.

The memory consistency model implemented in Intel's x86 architecture is ***total store order*** (TSO), which is a strong memory model that uses the core of sequential consistency. In a system that implements SPARC TSO, a read can complete before a write to a different address, but a read cannot return return the value of a write by another core or processor before all processors in the system are aware of the write. See chapter 4.2.

A memory consistency model is a set of rules which specifies how reads and writes in a processor will be managed. Consistency models define decorous behaviour in a multi-threaded processor system in terms of loads and stores. In order to get some actual intruition, we will consider two different cases: Suppose that the the value of X is replicated on the nodes M and N, client *A* writes a new value to X in node M, after a period of time, the value of row X in node N is read by client *B*. The memory consistensy model is determined to find out whether or not client *B* is aware of the write from client *A*.To get some actual

real-word intruition, let's consider another scenario: Suppose that a ballet course is scheduled to be in room 123. A week before the class starts, the universiry registrar dicides to reschedule the class to Room 321, the registrar asks the web site administrator to post the udpdate online, after which he sends an e-mail to all entrants regarding the room update – all students but one recieves the e-mail. Since the web site administrator accidentaly forgot to update the web site in time, one student still observes the old classroom, resulting in him missing the class. A memory consistency model is supposed to determine wherher a behaviour in a process is correct or incorrect.

2.2 Cache Coherence

A single computing component with two or more independent processing units (i.e. cores) is often referred to as a multi-core processor. The multi-core architecture may therefore be defined as a technology where one physical chip, contains the core logic of two or more processing units. By increasing the number of CPUs on a die, the overall speed for programs amenable to parallel computing will decrease, since the multiple cores can fetch several instructions from the memory simultaneously.

In a shared memory multiprocessor system, when clients maintain caches of a common memory resource, problems may arise with inconsistent data. The view of memory held by two different processors is through their individual caches, meaning that two different processors could end up seeing two different values on the same adress location in the memory. If the value of an operand changes before a processor fetches the data, the processor could end up obtaining the old value rather than the new. Should this occur, the processor is said to have seen stale data. The discipline intended to manage inconsistency in the memory, i.e. maintaining consistency between the cache- and primary memory is cache coherence. Cache coherene ensures that changes in the values are propagated throughout the system.

A key problem of many-core processors is to provide a consistent view of memory in a processor-system with various cache hierarchies. Even though cores logically access the same memory location, SoC-cache memories are essential in achieving a high performance for the memory accesses in a processor. *Cache coherence* is a design point for supporting memory models. The cache coherence mechanisms determine how data shall be transfered between processors, caches and the primary memory.

A cache coherence protocol mainly has three tasks:

1. At a write, the cache coherence protocol should remove all intances of the written data, or send the newly written value to all update all instances.
2. Detect if the current write is completed, so that the processor can execute the next instruction, i.e. perform the next memory access.
3. Maintaining the illusion of instructions being executed atomically.

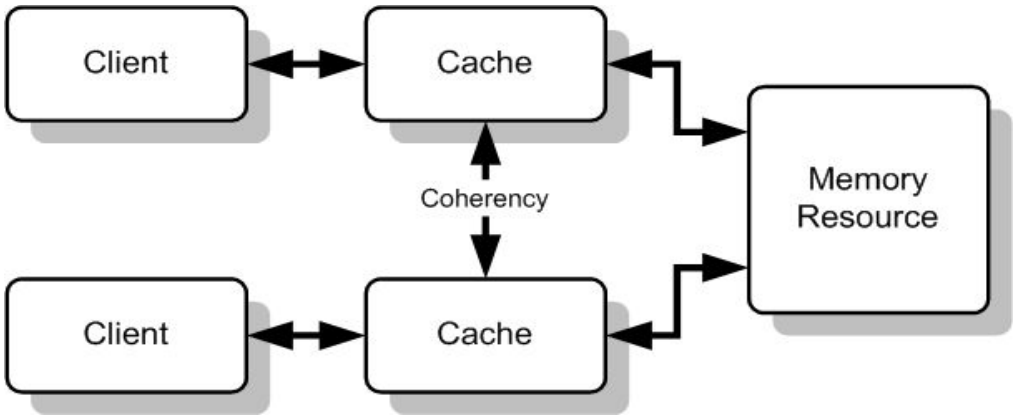

Figure 1 - A illustration of how coherence between the cache memories in a multi-processor system is achieved.

In the illustration above, if the client on top holds the same value in a memory block as the client on the bottom, the system is said to be coherent. If the client on the bottom changes the value held in the memory block a inconsistency will occur between the two client, since the client on top holds the *old* value. The coherence protocol used in the Haswell microarchitecture is the MESIF-protocol, see Chapter 5.

Detecting write completion
A request is sent to the memory where the requested data is located. Upon recieving the request, the memory knows in which cache's copies of the data are present, the memory either sends the updated value to all the cache's, or invalidates the data on all other locations. The recieving cache's then mush acknowledge that they recieved the correct data, the acknowledgement is sent back to the requesting cache, thus the process is done and the processor can continue to execute it's instructions.

2.3 Sequential Consistency vs. Cache Coherence

The assumption that cache coherence defines memory consistency is not true since, (1) coherence ensures that values written to a particular memory adress location are read by all other processor, and (2) consistency ensures that all values written to *different* memory adress locations are will be seen by all processors in a correct, and given, program order. It would seem, intuitively, that cache coherence defines consistency in shared memory systems, there are several reason to why it does not do that. One of the key goals in implementing a correct cache coherence in multi-core processors is to make all caches invisible to each other while maintaining coherence, and since coherence protocols only deal with one cache block at the time, problems may arise with inconsistency because most real programs access more than one cache block simultaneously. Thus, implementing sequntial consistency in multi-core processors can be very challenging, which is the reason to why *relaxed memory models* are used in Intel's x86-architecture.

3. The Haswell Architecture

Intel's Haswell microarchitecture is built on the same 22nm technology as it's prodecessor Ivy Bridge. Features carried over from the Ivy Bridge microarchitecture are, amongst others, the 6 kB micro-operation cache, the 64- and 256 kB L1 and L2 cache, aswell as the 3D tri-gate transistor.

The increasing level of intregration (Moore's Law), in the context of semiconductors, forces developers to find new solutions to minimize the size of the die as the amount of transistors on it increases. The processors in the Haswell microarchitecture is the first SoC's to take advantage of Intel's new process technology, the 22nm FinFET tri-gate transistor.

The new microarchtecture introduces several new features, such as a wider core (4 ALU's, three AGU's, a second branch execution unit, higher cache bandwidth, and a improved memory controller), a expanded instruction set and a updated instruction decode queue (allocation queue), which is no longer static but partioned between the two threads in each core.

3.1 Memory Hierarchy

In the core architecture for the Nehalem processor each core had a local L1 cache and a L2 cache which was shared between each pair of cores. This allowed two cores to coherently and efficiently communicate with each other, as the amount of cores started to increase on each single processor die, problems arose with implementing efficient and coherent communication links between the pairs of cores.

In the 4th generation Intel Core processors core contains of both a L1 D- and L1 I-cache, and a L2 D- and L2 I-cache. The first level cache is 64 kB large (32 kB D-cache and 32 kB I-cache), and the second level cache is 256 kB large (128 kB D-cache and 128 kB I-cache) All local and independent cache memories are write-back and non-inclusive. Meaning that they will not duplicate data stored in another core's cache memory. The cores in the Haswell architecture also shares a cache, implemented as an inclusive L3 cache, ranging from 2 to 20 MB. Iris Pro models of the Haswell architecture even have a level 4 cache of 128 MB. Since the L3 cache is inclusive, this means that it duplicates all the data stored in each core's individual L1 and L2 cache. Thus, greatly improving the inter-core communication efficiency. If a processor requests data from core one, which cant be found in any level of the core's cache memory, it knows that the data requested is not present in any other core's cache memory either. I.e. a given core will never need to locate data in another core's cache memory.

3.1.1 Sharing of Modified Data and False-Sharing

When a software thread running on one core tries to read or write data that is currently present in another core's local cache, in a modified state, some degree of performance penalty will happen, resulting in eviciton of the modified cache line back into memory. After which, the cache line will be read into the L1 cache of the other core. This latency, of cache line transfer, is higher than writing or reading data from the immediate first level- or even second level cache.

The term false sharing applies when two threads in a processor unwittingly modifies variables that shares the same cache line. The caching protocol may force the first thread to, despite a lack of logical necessity, reload the entire unit. If two core's in a processor operates simultaneously on independent data in the same memory adress region, the line may be forced across the bus, by the cache coherency mechanisms, with every new data write. This will result in wasted system bandwidth and time penalty through memory stalls. Write conflicts in shared cache lines/on the same memory adress region is the most limiting factor on obtaining scalability for parallel threads in a symmetric multiprocessing system.

3.1.2 Intel's Transactional Memory

The memory hierarchy carried over from Ivy Bridge is largely unchanged, the most substantial benefit is in the implementation of Intel's transactional memory hardware, TSX. Haswell's transactional memory uses the first level cache to store reference data. The extension is a part of the x86 instruction set archtecture and adds both the transactional memory support, an interface that provides greater flexibility for concurrent programming, and the hardware lock elision, which speeds up the execution time of multi-threaded software. Transactional memory attempts to allow groups of load and store instructions to be executed atomically, it is a mechanism in concurrent computing meant to control the acsess to shared memory. In this case, the L3 cache.

If the transaction requested through the processors transactional memory fits in the first level data cache, it should be able to execute successfully. In practice, this means that the first level data cache contains a bit of extra meta-data ("data about data", that acts as a flag) to track if a cache line in the memory have been read, written or modified, in order to detect data conflicts. If a data conflict is detected through the cache coherence protocol, in this case the MESIF protocol, the thread in which the conflict was detected will abort the data.

3.1.3 Conflicting Data and Transactional Aborts

A primary cause to a processor doing a transactional abort is due to conflicting data accesses between two logical cores. If a logical processor reads a location that is part of the write-set or writes to a location that is either part of the read- or write-set in a transactional region, a data conflict will occur, thus resulting in a transactional abort by the transactionally executing

logical core. Data conflicts are detected in the granularity of a cache line Thus, unrelated variables in the same cache line therefore appears to have the same adress as the conflicting data in the transactional region, resulting in unnecessary transactional aborts. If the amount of data accesed in a transactional region happens to exceed a specific capacity, the concerned thread will make a transactional abort.

In most Haswell processors, decluding Iris Pro models, the L3 cache is often reffered to as the LLC (last level cache). It is the highest level cache and is called before accesing the primary memory. The LLC in the Haswell processors is shared between multiple cores, thus increasing the level of parallelism between the cores. Since the L3 cache is sliced into multiple pieces (the number of slices is equal to the number of IA cores), memory addresses can be accessed independently. Each slice in the L3 cache has it's own logic portion that handles data coherency, memory ordering, L3 misses, and write-back to the main memory.

To manage data conflicts and to insure cache coherency, the L3 cache has flags to keep track of the source of a specifik data, the cache will be able to track the data and tell from which core the data came. If an operand modifies a data value in the L3 cache, then the flags in the L3 cache will be able to tell in which core the data will need to be updated to keep the coherency between the cores. By doing so, the "snooping" coherency between the cores greatly decreases.

The cache coherence protocol used to organize the memory hierarchy in the Haswell architecture is a modification of the popular MESI protocol, called the **MESIF** protocol.

4. Memory Consistency and the x86-architecture

4.2 Strong Memory Models and the x86 TSO Memory Model

In a *strong memory model* machine instructions comes implicity with release and acquire semantics. This means that, when a thread in a processor performs a sequence of writes, every other core in the processor observes the values in the same order in which they were written in by the thread.

MSDN, Microsoft, defines acquire and release semantics as:[1]

> **Definition:** *"A read-acquire executes before all reads and writes by the same thread that follow it in program order."*
>
> **Definition: "***A write-release executes after all reads and writes by the same thread that precede it in program order."*

In a strong memory model the follow reorders are allowed. Reads may not move ahead of reads, writes may not move ahead of writes, and writes may not move ahead of reads. Allowed reorder's are reads moving ahead of writes. Thus:

Reordering actions	**x86/x64**
R → R	False
W → W	False
W → R	False
R → W	True

Table 1 - Allowed reorderigs in a strong memory model

[1] https://msdn.microsoft.com/en-us/library/windows/desktop/ee418650.aspx

4.2.1 Sequential Consistency

Since there can be multiple copies of a variable in a memory hierarchy, sequential consistency requires some kind of control mechanism is implemented, the control mechanism is supposed to determine whether or not a cached copy of data is allowed to be read, or if another processor has modified the data recently. This control mechanism is refered to as the cache coherence protocol.

Leslie Lamport defines *Sequential Consistency* as:[2]

> **Definition:** *"A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."*

Sequential consistency is the strongest memory model and a correct execution requires that, (1) a system that implements the *sequential consistency* memory model respects each core's program order, regarding their loads and stores into memory, and (2), memory requests to any given memory adress location are serviced from a FIFO-queue, thus a request returns the last value written in the requested adress location.

Consistency example 1:

Process A	Process B
data_value = 64;	while (flag == 0) { … }
flag = 1;	memory[k] = data_value;

In a system where sequential consistency is implemented the following rules are requierd: (1) the write to data_value must complete before the write to flag can begin, and (2), the read of flag must complete before the read of data_value can begin. As mentioned above, multi-core processors requires their cache's to be lock-up free, a system implemented as a sequential consistent system requires that one process can't execute before the precedent process is done, thus requiring the cache's to be managed via locks. I.e. sequential consistency violates the rules of cache coherene in a multi-core processor system, and therefore lowers the performance in the processor.

[2] Lamport, Leslie (1979). "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Program".*IEEE Trans. Comput.* **28** (9): 690–691.doi:10.1109/TC.1979.1675439. ISSN 0018-9340

4.2.2 The x86 TSO Memory Model

Total store order is a strong memory model and is built upon the sequential consistency memory model. Processor systems in Intel's x86 architecture agrees that, operations to all data form a single total order, and that the order is consistent with the order of operations in each individual processor in the regarded system.

Write buffers are used to hold commited stores until they can be processed by the rest of the memory system. In a uni-core processor the view of write buffers can be architecturally invisible by ensuring that a load to a specific memory location returns the most recent deposited value to the same location, even if there are several instances of the same memory location in the write buffer. In a multi-core processor system a store enters the buffer when the write commits, and a write exits the buffer when the to regarded block in the cache is in a read-write coherent cache state. A important key part in implementing a correct memory model is to avoid memory stalling and thus avoiding time penalty in the processor. A write can enter the write buffer even before the regarded cache has obtained permission for the block to be written. Thus, the TSO model allows a read to return the value of it's own processor's write before the write is acknowledged by the other processor's.

5. Implementation in the Haswell-architecture

5.1 The MESIF Protocol

The MESIF (Modified-Exclusive-Shared-Invalidated-Forward) is a snooping cache coherence protocol, which is based on the MESI (Modified-Exclusive-Shared-Invalidated) protocol, and is developed by Intel for cache coherence in the NUMA-architecture (non-uniform memory architecture, a memory design used in multiprocessing). A snooping cache coherence protocol propagates

In the new protocol, Intel included an additional state, the Forwarding (F) state, to prevent multiple caches holding a shared line to respond when a cache line is requested, but also changed to role of the Shared (S) state. In the MESIF protocol, the only instance that may be duplicated is the cache line in the Forward state, this cache line is used to respond to read requests. Other caches may hold the same cache line, but it will be held in the Shared state. Shared state cache lines are now silent, which makes the Forward state cache line first amongst equals, when responding to requests.

In the MESI protocol, when a processor requests a cache line, if the cache line is stored in multiple memory locations, every location will respond to the request. Thus, unnecessarily many redundant messages will be forwarded across the bus, often resulting in a high latency. If the processor is able to designate one single cache line to respond to the request, the contemporaneous traffic will be reduced. The advantages of implementing the MESIF protocol instead of the MESI protocol is illustrated in Figure 1 below, by marking one cache (Peer F in the image below) as a "**F**orwarding cache" redundant messages in the system are avoided since only one cache memory will answer to each request. thus the forwarding state facilitates the rapid responce of a cached copy.

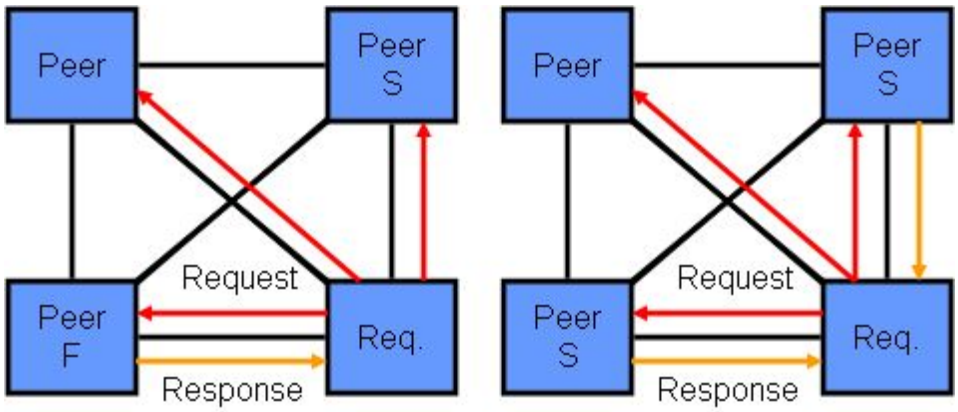

Figure 2 - MESIF protocol vs. MESI protocol[3]

[3] http://www.realworldtech.com/common-system-interface/5/

In a cache system employing the MESIF protocol requests will be responded with cache-to-cache speed, instead of waiting for a respond from a “slow” main memory or the responds of all locations holding the cache line.

Each cache line in the cache is in one of the five states (**M**odified, **E**xclusive, **S**hared, **I**nvalid or **F**orward):

> **Modified** - The cache line is dirty, the memory adress in the cache and in the main memory does not hold the same value, and it must be written back to main memory before other reads or writes to the regarded memory adress can be done.
>
> **Exclusive** - A cache line held in the Exclusive state is unique for that cache and the main memory, i.e. it is not present in any other cache’s.
>
> **Shared** - The Shared state indicates that a cache line held in one cache may also be present in other cache’s. The other cache’s can update the cache line and thus leaving a inconsistency, the L3-cache deals with inconsistency by using flags to keep track of which cache’s that holds the regarded cache line.
>
> **Invalid** - All cache lines are *invalid* initially, when a read miss occurs, the requested cache lines is moved to the requesting cache and the cache line is marked as *valid*. The Invalid state marks whether or not a cache line is used.
>
> **Forward** - The forward state is unique for the MESIF protocol, this state is designated to respond and update all cache that are sharing a requested cache line.

When a thread is writing to a shared memory location, the MESIF protocol requires that all other copies of the memory location is invalidated. If this occurs, all cores in the processor sends a RFO-request (read-for-ownership request) to the LLC (in this case the L3-cache) that checks the snoop-filter and then sends out invalidations to all cache’s that holds a copy of the cache line.

As mention in Chapter 4, MESIF does support sequential consistency, but it does not dictate it. If all processors perpretate instructions that modifies the value in a shared memory location, in the rules of sequential consistency (write atomicity and in-order instructions) then sequential consistency in a system implementing the MESIF protocol is achieved.

6. References

Intel, september 2015 (last accessed 2015-12-08)
http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

Intel, september 2015 (last accessed 2015-12-08)
http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

David Kanter, November 2012 (last accessed 2015-12-08)
http://www.realworldtech.com/haswell-cpu/
http://www.realworldtech.com/common-system-interface/5/

International Journal of Computer Science and Information Technologies, 2013
(last accessed 2015-12-08)
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.310.7541&rep=rep1&type=pdf

Mark D. Hill, published in Morgan Claypool Publishers, 2011 (last accessed 2015-12-08)
https://lagunita.stanford.edu/c4x/Engineering/CS316/asset/A_Primer_on_Memory_Consistency_and_Coherence.pdf

Michael R. Marty, 2008, (last accessed 2015-12-08)
http://research.cs.wisc.edu/multifacet/theses/michael_marty_phd.pdf

Mahesh Neupane (last accessed 2015-12-08)
https://web.archive.org/web/20100620091706/http://cse.csusb.edu/schubert/tutorials/csci610/w04/MN_Cache_Coherence.pdf

Jeff Preshing, Preshing on Programming, July 2012 (last accessed 2015-12-08)
http://preshing.com/20120710/memory-barriers-are-like-source-control-operations/
http://preshing.com/20120930/weak-vs-strong-memory-models/
http://preshing.com/20120913/acquire-and-release-semantics/

Sarita V. Adve, 1993 (last accessed 2015-12-08)
http://pages.cs.wisc.edu/~markhill/theses/sarita_adve.pdf

Scott Owens, Susmit Sarkar, Peter Sewell (last accessed 2015-12-08)
https://www.cl.cam.ac.uk/~pes20/weakmemory/x86tso-paper.tphols.pdf
Extended version:
http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-745.pdf

MSDN, Microsoft (last accessed 2015-12-08)
https://msdn.microsoft.com/en-us/library/windows/desktop/ee418650.aspx

J.R. Goodman, University of Auckland, H.H.J. Hum, Intel Corporation, 2009 (last accessed 2015-12-08)
https://researchspace.auckland.ac.nz/bitstream/handle/2292/11594/MESIF-2009.pdf?sequence=6