Introduction to multiprocessors

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Javier García Blas

Computer Architecture
ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid

1. Module structure

This module is structured in four lessons:

- **Symmetric shared memory**. It introduces the concept of multiprocessor architecture and the design alternatives in shared memory machines. It presents the problem of cache coherence and the available alternatives for solving it. Besides, it offers some details on *snooping* protocols.
- **Memory consistency models**. It introduces the concept of memory consistency and the sequential consistency model. It also presents more relaxed consistency models, including the release/acquire model. Besides, it offers some details on the memory model from Intel processors.
- **Synchronization**. It presents the synchronization problem in shared memory machines. It also presents the different alternatives for hardware primitives that can be used to implement synchronization. It introduces the concepts of *lock* and *barrier* as well as their design alternatives.
- **Distributed shared memory**. It offers additional details on the concept of distributed shared memory and presents the directory based protocols.

2. Symmetric shared memory

This lesso has the following general structure:

1. Introduction to multiprocessor architecture.
2. Centralized shared memory architectures.
3. Cache coherence alternatives.
4. Snooping protocols.
5. Performance in SMPs.

2.1. Introduction to multiprocessor architectures

In recent years, multiple reasons have contributed to increase the interest in multiprocessor machines. On one hand, the diminishing benefits from instruction level parallelism and the increase in energy consumption make it necessary alternative mechanisms to improve computers performance. Additionally, new market segments, as cloud based services and software as a service, have placed more and more demand on high performance servers. Finally, the increase of data intensive applications require the ability to process greater data volume.

To be able to exploit the parallelism offered by architectures with several processors it is necessary to exploit thread level parallelism. This assumes the existence of multiple program counters. While this programming model has been widely used in the scientific computing domain, its use outside that domain is something relatively recent.

A multiprocessor is a computer consisting of several highly coupled processors with two important characteristics. On one hand, the processors coordination and use is under a single operating system. On the other hand, all memory is shared, and consequently, there is a single global address space shared by all processors.

To make use of a multiprocessor programming models most frequently used are parallel processing, request processing, and multiprogramming. From a physical point of view, alternatives go from CMP (*chip multi-processors* or *multi-core*) to the use of multiple chips or the use of multicomputers.

To be able to adequately exploit a multiprocessor with **n** processors it is necessary to be able to decompose the program to be executed into **n** threads of execution. Those execution threads can be identified explicitly by the programmer, be created by the operating system from received requests, or be generated by the compiler. It is important to remark that, in any case, the number of instructions to be executed by every thread must be sufficiently high to compensate the cost of thread creation and destruction.

From the memory point of view, a multiprocessor may follow two differentiated designs. The use of centralized shared memory leads to SMP (*symmetric multiprocessor*) machines using UMA (*Uniform Memory Access*) memory. The use of distributed shared memory leads to DSM (*Distributed Shared Memory*) machines using NUMA (*Non-Uniform Memory Access*) memory.

2.2. Centrilized Shared Memory Architectures

One reason for using centralized shared memory is the existence of greater and greater cache memories within de processor, decreasing the effective demand on main memory.

In a cache memory we may distinguish to kind of data: private data and shared data. Problems arise due to the existence of shared data, as they can be present in more than one cache memory at the same time opening the opportunity for different copies to have different values. This problem is known as cache *coherence* (or *incoherence*).

A memory system is said to be *coherent* if any read from a memory location returns the value that has been most recently written in that memory location.

We need to distinguish between to problems that ar distinct, but highly related. *Coherence* determines which value is returned in case of a read. On the other hand, *consistency* determines in which moment in time is the write made visible to the rest of processors. In this way, coherence defines the behaviour of reads and writes on the same memory location, while consistency define the behaviour of reads and writes on one memory loction with respect to another memory location.

To be able to guarantee consistency, three conditions must hold: program order preservation, a coherent view of memory, and write serialization.

2.3. Alternatives to cache coherence

One coherent processor must offer two fundamental properties for performance. On one hand shared data migration and on the other hand data shared read data replication. Those properties may be integrated into a cache coherence protocol

There are two families of cache coherence protocols: directory based protocols and snooping protocols. In directory based protocols the sharing state is kept in a data structure (the *directory*), that may be centralized or distributed. In snooping protocols each cache keeps information of sharing state for each block it stores.

2.4. Snooping protocols

In a snooping protocol tow strategies can be used: write invalidation and write update. In invalidation, the protocol guarantees that a processos has exclusive access to a block before performing the write over it. Previously, the rest of copies must be *invalidated*. In contrast, in write update each time a write is performed on a memory location, this must be broadcasted to all caches, which would eventually modify their own block copy. As this strategy makes use of more bandwidth, the invalidation strategy is usually preferred.

In invalidation, whe an processor needs to invalidate a datum, it must acquire the memory bus and broadcast the address to be invalidated. All processors will be observing (*snooping*) the bus to check whether they have that address, and consequently, a copy of the invalidated block. In this way, the bus is used as a serialization mechanism, excluding the possibility that simultaneous writes can happen. In case a cache miss happens, the behaviour is different depending on whether the write strategy is write-through or write-back. With write-through, the main memory always holds the latest write that has been performed and the datum must be taken from there. However, in the case of write-back, if a processor has a modified copy from that block, that processor cache will answer the cache miss from the other processor.

The most basic protocol based on snooping and invalidation is the MSI protocol. This protocol uses a state machine with three states for each block in cache memory. State transitions may generate requests to processor cache or request to the bus. The three basic states are defined in terms of the block state. The block may have been modified (M), be shared (S), or have been invalidated (I):

There are serval extensions to the basic protocol. The MESI protocol adds a fourth exclusive state to signal that the block resides in a single cache memory but it has not been modified. Other alternatives are the MESIF protocol (used in Intel Core i7) and the MOESI procol (sued in AMD Opteron).

2.5. Performance in SMPs

The use of cache coherency policies has impact on the miss rate and, consequently, on performance. Two new kind of misses emerge. The true sharing misses and false sharing misses. The latter, are due to two different accesses from different processors to words belonging to the same block.

3. Memory consistency models

This lesson has the following general structure:

1. Memory model.
2. Sequential consistency.
3. Other consistency models.

4. Use case: Intel.

3.1. Memory model

A consistency memory model defines the set of rules that define how processor generate read and write operations are processed by the memory system. Thus, such model can be seen as a contract between the programmer and the system.

In general, any memory consisntency model determines validity of possible optimizations that can be performed on correct programs. Consequently, the memory model defines the interface betwee a programmer and its transformers (as the compiler or the very hardware on which the program is executed).

In case of a single processor system, memory operations happen in program order. That is, semantics is defined by the sequential program order.

3.2. Sequential consistency

In case of multiprocessors, the simplest memory model to reason about is the sequential consistency model. The concept of sequential consistency was defined by Leslie Lamport in 1979.

Sequential consistency establishes two constraints: program order and atomicity. On one hand, memory operations from a program must be visible to all processes in *program order*. On the other hand, all memory operations must be *atomics* requiring that nothing that a processors does after of having seen the new value from a write is made visible to other processes before they have seen the value of that write.

Sequential consistency constrains order of memory operations and it is the simplest model to reason about prallel programs. However, even simple reorderings that are valid in the context of a mono-processor, are no longer valid.

There is a set of sufficient conditions that guarantee the existence of sequential consistency. However, there might be conditions less demanding.

3.3. Other consistency models

There are other consistency models less constrained than the sequential consistency model. Those models can be defined in terms of the operations that can bypass other operations.

In weak ordering, operations performed on memory are divided into two kinds: data operations and synchronization operations. Synchronization operations act as barriers. In this way, all synchronization operations must be executed in program order and data operations admit reorderings as long as those do not bypass a barrier imposed by synchronization operation.

Other more relaxed model than weak ordering is the release/acquire consistency. In such case, synchronization operations may be from two types: release or acquire. An acquire operation must complete before all subsequent memory accesses. In the case of release, all previous memory accesses must complete before starting that release.

3.4. Use case: Intel

In the case of the Intel processor family, the memory model has been defined trhough a series of processor generations.

On one hand, the model defines in which cases can the memory operations be considered atomics. It als defines the different conditions for a memory operation to be able to establish memory bus blocking.

On the other hand, the model also defines synchronization instructions (barrier instructions).

Finally, it defines the rules from the memory model within the processor as well as the memory model ruls for accesses among multiple processors.

4. Synchronization

This lesson has the following genral structure:

1. Introduction.
2. Hardware primitives.
3. Locks.
4. Barriers.

4.1. Introduction

In shared memory systems, communication is performed through reads and writes on that shared memory. To avoid problems with the concurrent access to memory it is necessary to establish synchronization mechanisms for shared variables access. We may distinguish two cases: 1-1 communication (communication between two processes) and collective communication (communication between an arbitrary number of processors.

In 1-1 communication it is necessary to ensure that reception happens after sending. Generally, it is necessary to ensure that memory accesses are performed guaranteeing *mutual exclusion*. In explanation, only one of the processes may access at teh same time to a shared variable.

In collective communication, it is necessary to coordinate multiple accesses to variables guaranteeing that writes are performed without interferences and reads wait until data is available. Again, it is necessary that accesses to shared variables are performed with *mutual exclusion*. In that case, it is possible to guarantee that a result is not read until all processes have executed their own critical section.

4.2. Hardware primitives

The consistency model many be insufficient and complex. Consequently, it is usually complemented with hardware primitives for read-modify-write. Those can be from different types as:

- Test-and-set instruction.
- Exchange instructions.
- Fetch and operation instruction.
- Compare and exchange instruction.
- Store conditional instruction.

4.3. Locks

A lock is a mechanism to ensure mutual exclusion. It has two associated operations: the acquisition operation (*lock*) and the release operation (*unlock*).

The acquisition operation may imply a wait. There are two alternatives to implement the wait mechanism: busy waiting and blocking. With busy waiting, the process remains in a loop which is constantly querying for the value of the variable. This is the basic underlying idea behind a *spin-lock*. With blocking, the process remains suspended and processor is given to another process. Consequently, blocking requires the participation of a process scheduler, which is typically a part of the operating system or from a run-time support system. While selecting the alternatives it is necessary to take into account the run-time cost trade-offs.

In general, in the design of a lock mechanisms there are three design elements: the acquisition method, the waiting method and the release method.

The simplest implementation of a lock is based on using a shared variable that can take two values (open and closed). To reduce the wait mechanism memory accesses, a technique that can be used is the exponential delay. Another optimization, that can be used in some cases, is using the same variable to synchronize and communicate.

Last but not leaset, another problem for the simplest implementations is that they do not fix the lock acquisition order. This leads to starvation problems. The solution to this problem is to make the lock to be acquired by request age guaranteeing FIFO order. To achieve that goal, tagged locks or queue based locks can be used.

4.4. Barriers

A barrier allows to synchronize several processes at some point and guarantees that no process passes the barrier until all have arrived. Thus, barriers are used to synchronize stages in a program. The simplest implementation for a barrier is a centralized barrier were barrier is associated to a counter for the processes that have arrived to it. To avoid problems derived from barrier reusing in the case of loops, barriers with way inversion may be used. Finally, to avoid scalability and contention problems when accessing shared variable, hierarchical barriers can be used where arriving and released processes are structured as a tree.

5. Distributed shared memory

This lesson has the following general structure:

1. Introduction to distributed shared memory.
2. Directory based protocols basics.
3. Directory based protocol.

5.1. Introduction to distributed shared memory

Snooping protocols require communication with all caches both in every cache miss and in each shared data write. However, its use is due to the absence of a centralized data structure. This results in a low implementation cpost. When the processors number increases, the snooping protocol scalability problems also increase.

In distributed shared memory (DSM) machines, coherence traffic on bus may become a bottleneck. The other alternative is the use of directory based protocols keeping the sharing state. Those may be

used at two levels. In SMP processors, a centralized directory in the last level cache may be used. In DSM machines, a distributed directory may be used, avoiding bottlenecks derived from bus traffic.

In general, the basic idea for a directory based protocol is to keep information about the state for each cache block. This information includes a list of caches containing a copy of that block, as well as the status bits for the block.

Within a multi-core processor, this information may be represented using one bit per core. Thus, only invalidations to to caches marked in this bitmap are sent, avoiding broadcasting messages. While, those messages are avoided, when the number of processor increases, the centralized directory causes scalability problems.

The distributed directory solves those scalability problems distributing the directory with the memory. Consequently, each directory has information on the blocks from the associated local memory.

5.2. Directory based protocol basics

A directory based protocol keeps the state for each block. This state can be: shared (one or more nodes have the block in cache and the value in memory is up to date), non cached (no node has a copy from the block) or modified (only one node has a block copy in cache and it has overwritten it). In particular, the protocol tries both read misses and writes in clean shared blocks.

5.3. Directory based protocol

In multi-core chips, internal coherence is kept through a centralized directory. That directory may act as local directory in the case of distributed shared memory. In that case, the protocol implementation requires handling local and distributed transitions.