Shared memory multiprocessors

Leonid Ryzhyk
<leonidr@cse.unsw.edu.au>

April 21, 2006

1 Introduction

The hardware evolution has reached the point where it becomes extremely difficult to further improve the performance of superscalar processors by either exploiting more instruction-level parallelism (ILP) or using new semiconductor technologies. The effort to increase processor performance by exploiting ILP follows the law of diminishing returns: new, more complex optimisations tend to cost more in terms of silicon as well as design effort and provide smaller and smaller performance gains. In addition, aggressive use of speculative caching and execution techniques in modern superscalars leads to poor energy efficiency — an important concern in both embedded systems with limited battery capacity and in server systems, where heat dissipation is a problem of growing importance.

The natural solution is to rely on *thread-level parallelism* (TLP) rather than ILP to further increase the computational power of computer systems. The following forms of TLP are currently being used: *explicit multithreading*, *chip-level multiprocessing* (CMP), *symmetric multiprocessing* (SMP), *asymmetric multiprocessing* (ASMP), *non-uniform memory access multiprocessing* (NUMA), and *clustered multiprocessing*.

With the exception of clustered multiprocessors, all of the above architectures provide all cores in the system with access to a shared physical address space. The shared memory organisation has three major advantages over simpler private memory organisation. First, because in shared-memory systems communication does not have to interfere with computation and because access to shared memory can be streamlined using hardware caching, shared memory provides an extremely efficient low-latency high-bandwidth communication mechanism. Second, shared memory provides a natural communication abstraction well understood by most developers. Third, the shared memory organisation allows multithreaded or multiprocess applications developed for uniprocessors to run on shared-memory multiprocessors with minimal or no modifications.

The goal of this report in to give an overview of issues and tradeoffs involved in memory hierarchy design for shared memory multiprocessors.

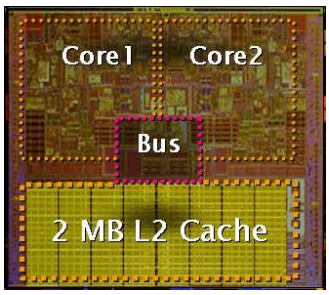


Figure 1: The Intel Core Duo processor layout.

The report is largely based on the material from Hennessy and Patterson [HP03], Culler, Singh, and Gupta [CSG99], and Adve and Gharachorloo [AG95]. Other sources are referenced throughout the report.

2 Memory hierarchy organisation: a high-level overview

This section presents a high-level overview of alternatives involved in memory hierarchy design of a shared memory multiprocessor. The most fundamental question to be answered by the memory system designer is: On what level of the memory hierarchy should physical sharing of memory occur? Moving sharing closer to processor cores enables faster inter-processor communication but typically slows down regular non-shared memory accesses, while moving sharing to lower levels of the memory hierarchy allows faster non-shared accesses at the cost of more expensive inter-processor communication. Another general trend is that architectures that share fewer layers of the memory hierarchy scale better. The available design alternatives are:

- **Shared L1 cache.** This approach is only used in chips with explicit multithreading, where all logical processors share a single pipeline.
- **Shared L2 cache.** Some CMP systems are built with shared L2 caches. This design minimises on-chip data replication and makes more efficient use of cache capacity. For example, Figure 1 shows the physical layout of the Intel Core Duo processor with shared L2 cache. Unfortunately, shared L2 caches have longer access times; therefore, many existing CMP systems use private L2 caches.
- **Shared main memory.** In this memory system organisation, every processor or core has its own private L1 and L2 caches, but all processors share the common main memory. For example, in the dual-core Itanium 2 Montecito processor (Figure 2), every core has 3 levels of private on-chip cache connected to a shared off-chip memory controller. Until recently, this was

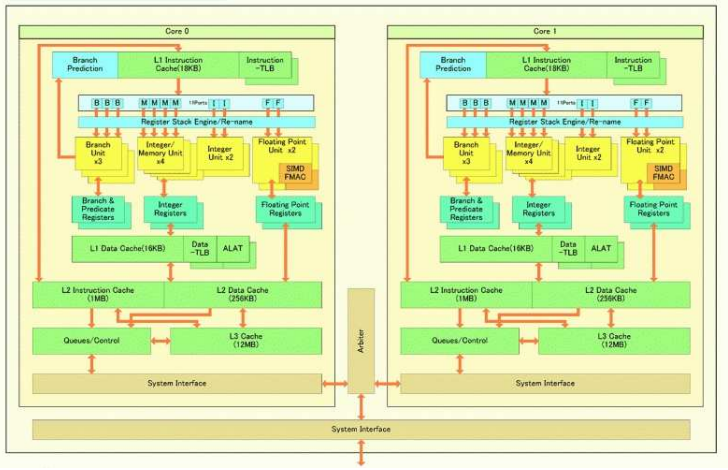


Figure 2: The Montecito processor with 3 levels of private on-chip cache.

the dominating architecture for small-scale multiprocessors. However, as the gap between processor and memory speeds is increasing and as the number of threads per chip is growing, memory bandwidth is becoming a bottleneck even for small-scale systems. Therefore, some of the recent architectures, such as AMD64, abandoned the shared memory organisation and switched to the NUMA organisation (see Figure 3).

- **No physical sharing.** In this memory system organisation, every processor (or node consisting of more than one processors) has it own private main memory and can access remote memory connected to other nodes through interconnection fabric. Of course, the cost of access to local memory is much lower than for remote memory access; therefore this architecture is known as the non-uniform memory access architecture or NUMA. One example of the NUMA architecture are Sun WildFire servers (Figure 4) that can include up to 4 symmetric multiprocessors, each consisting of up to 28 UltraSparc 4 processors.

Once the overall memory hierarchy layout is fixed, the next step is design of interconnects between different levels of the hierarchy. The two major types of interconnects are bus-based and network-based interconnects. The main difference is that a bus is a shared resource that can be used by a single client at a time, and thus supports a single flow of data, while an interconnection network provides larger bandwidth by allowing multiple simultaneous flows of data. Buses connect communicating entities directly and therefore are generally faster under moderate loads. In contrast, messages in an interconnection network travel from the source to the destination through a number of switches, which leads to higher latencies but allows much better scalability. Section 3 discusses interconnection networks design in more detail.

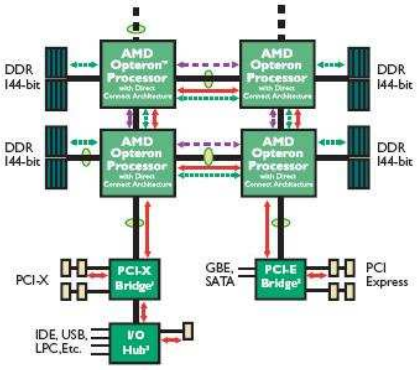


Figure 3: A 4-way AMD64 multiprocessor with private memories.

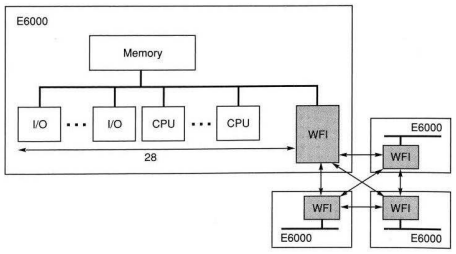


Figure 4: The Sun WildFire architecture.

The next major issue in memory hierarchy design is selection of a consistency model. In a multiprocessor system data can be replicated across multiple locations, including the main memory, private and shared caches, memory write buffers, and processor registers. As a result, the problem of maintaining data consistency arises. In addition, modern processors execute instructions out of order. This does not constitute a problem in a uniprocessor system, as long as no data or control dependencies are violated. For example, in a uniprocessor system, reads and writes to different memory locations can be safely reordered. In contrast, as we will see in Section 5, reordering memory accesses to shared memory locations in a multiprocessor system can have dramatic impact on program behaviour. A memory consistency model defines what properties should be enforced among reads and writes issued by different processors in the system. Two related notions are used to reason about memory access consistency in multiprocessor systems. *Cache coherence* describes ordering of memory reads and writes to the same location, while *memory consistency* describes ordering of memory accesses to different locations. Section 4 will describe algorithms for enforcing cache coherence, while Section 5 will study the tradeoffs involved in the design of a memory consistency model.

3 Interconnection networks

The interconnection network design space has four dimensions:

- **Topology** defines the physical interconnection structure of the network graph. Some standard topologies are described below.
- **Routing algorithm** determines which routes messages can follow through the network graph.
- **Switching strategy** determines how the data in a message traverses its route. The switching strategy can be either *circuit switching*, in which a path from the source to the destination is established and reserved before the actual data transfer, or *packet switching*, in which packets are individually routed from source to destination.
- **Flow control mechanism** determines how the network deals with multiple data flows requiring access to the same network resource (e.g., a channel).

From the performance point of view, two most important characteristics of interconnection networks are message latency and bandwidth. Latency is the time required to transfer n bytes of information from its source to the destination. The following formula describes the four components of message latency:

$$Time(n)_{S-D} = Overhead + RoutingDelay + ChannelOccupancy + Contention$$

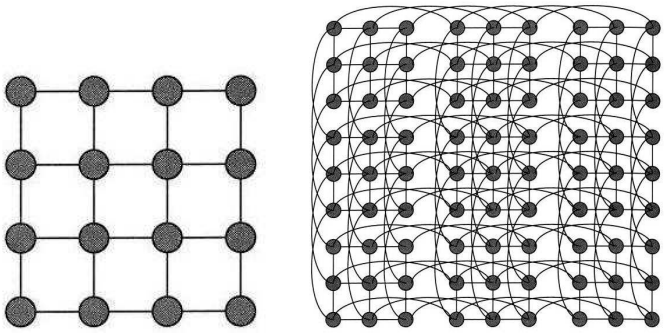


Figure 5: The hypercube topology.

Overhead is the time of getting the message into and out of the network on the ends of the transmission. *Channel occupancy* is the actual time used to transfer the packet along communication channels within the network. *Routing delay* is the sum of delays that occur at each switch on the way of the packet when selecting the right output port for the packet. *Contention delay* is the time the packet spends queued at each switch waiting for the output channel to become available.

Network bandwidth can be studied from two points of view. *Local bandwidth* is the bandwidth available to individual node, while *global bandwidth* characterises the aggregate amount of data that can be transmitted through the network in a unit of time. The most common measure of aggregate bandwidth is *bisection bandwidth* — the sum of the bandwidths of the minimum set of channels which, if removed, partition the network into two equal unconnected sets of nodes.

The primary factor that determines network performance and scalability is the topology of the network. In particular, the network *diameter* and the *average routing distance* determine message latencies, while the number of channels connecting two halves of the network determines the bisection bandwidth. Some of the most popular network topologies are *crossbar*, *hypercube*, *torus*, and *fat tree*.

A crossbar is a fully connected network that directly connects all its inputs to all outputs. Of course, this topology does not scale very well. Therefore, crossbars are mainly used to connect systems with small numbers of nodes and as building blocks for larger networks.

A d-cube (or a d-dimensional hypercube) is a d-dimensional array of nodes, where every node is directly connected to its grid neighbours that differ by one in precisely one coordinate (see Figure 5). A d-cube with k elements along each dimension has the diameter of $d(k-1)$ and the bisection of k^{d-1}. A torus is a hypercube whose faces are connected in such a way that every node in the network has $2d$ edges coming out of it (Figure 6).

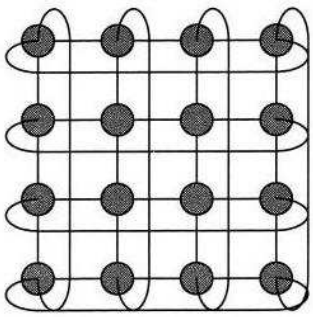


Figure 6: 2D torus topology.

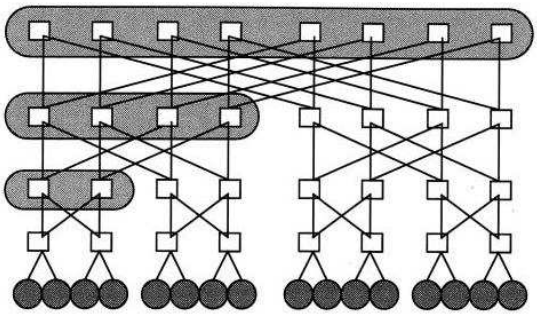


Figure 7: The fat tree topology.

A fat tree is an interposition of multiple regular trees, each of which contains all computational nodes in the system in its leaves (see Figure 7). The diameter of a fat tree is equal to $2(h-1)$, where h is the height of the tree. The bisection is equal to the number of regular trees interposed to form the fat tree.

4 Cache coherence protocols

The term "cache coherence" refers to the ability of a memory system to synchronise access to individual memory locations by different processors. The cache coherence property adds two additional constraints to conventional uniprocessor memory ordering semantics. First, it requires that all processors in the system observe all writes in the same order. This property is known as *write serialisation*. Second, it requires that all writes eventually become visible to all processors. However, it does not define any constraints on when this should happen. The issue of exactly when a written value must be seen by a reader is defined by a memory consistency model.

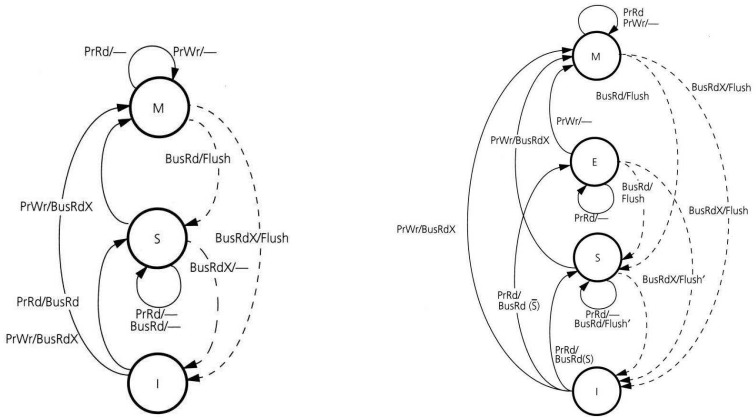
In bus-based and some small-scale interconnect-based shared memory systems, cache coherence is implemented using a technique known as *bus snooping*:

each processor on the bus monitors bus transactions issued by other processors and updates the status of data in its cache according to this information. This involves looking for a cache line on every memory access issued by any processor in the system. Performing this lookup in the L1 cache could severely interfere with processor operation. Therefore, cache coherence controller is usually inserted between the memory bus and the L2 cache, so cache coherence is enforced across L2 caches. For this to work properly, the L2 cache must be *inclusive*, that is it should contain L1 cache as its subset.

There exist 2 major types of snooping protocols: *update-based* and *invalidate-based* protocols. Update protocols immediately put writes performed by a processor on the bus, thus updating all copies of the data. In invalidate protocols, a processor performs an invalidate bus transaction before writing the data in order to ensure that it has the only valid copy of the data block. After exclusive access to the data is obtained, subsequent writes can be safely performed locally. Obviously, update protocols raise much more bus traffic, while the lazy nature of invalidate protocols allows batching multiple writes in a single bus transaction. Given that the memory bus is a very limited resource, invalidate protocols are much more popular nowadays.

A cache coherence protocols can be specified as a finite state machines (FSM) defined for each individual cache line. Figure 8a shows the FSM for the simplest snooping protocol with 3 states: *modified*, *shared*, and *invalid*. The modified state means that the only valid copy of the data block is stored in the local cache. The local processor is the exclusive owner of the data and can freely read and write it locally, until another processor requests access to the same block. The shared state means that a valid copy of the data is contained in the local cache and, probably, in some of the other processors' caches. This data can be read locally, but before modifying the data, it must be upgraded to the shared state by issuing an invalidate request on the bus. Finally, the invalid state means that the local cache does not have a valid copy of the data. The data can be either in the main memory or in other processors' caches. To access the data, the appropriate request must be issued on the bus. Note that this protocol relies on the atomicity of bus transactions to enforce strict ordering of memory accesses. That is, write serialisation is enforced by the bus arbitration mechanism.

Numerous improvements to this simple protocol have been developed. One simple improvement known as the MESI protocol is illustrated in Figure 8b. The basic observation behind this protocol is that, in the MSI protocol, if a processor first reads a data block from memory and then writes it, two bus transactions are required even if there is no sharing: one to upgrade the block from invalid to shared state and another one to upgrade from shared to modified. To overcome this inefficiency, a new state is introduced to indicate that the cache line is exclusively owned by the local processor and is unmodified. This state is called *exclusive clean*. The advantage of having this additional state is that it can be upgraded to the modified state locally, without issuing additional bus requests. Alternatively, it can be downgraded to shared or invalid if another processor issues read or write request to this block. In order to determine whether a



(a) The FSM of the MSI protocol (b) The FSM of the MESI protocol

Figure 8: Snooping protocols for cache coherence.

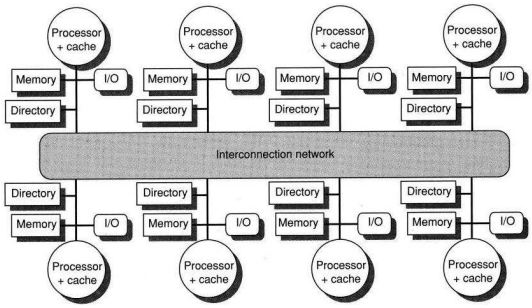


Figure 9: The layout of a multiprocessor with directory-based cache coherence.

cache line must be in shared or exclusive clean state, a new line is added to the memory bus.

In a large-scale multiprocessor, broadcasting memory accesses to all nodes becomes impractical; therefore a different mechanism for maintaining cache coherence is used. *Directory-based* protocols maintain information about sharing status of every memory line in a directory. Instead of broadcasting memory accesses to all nodes in the system, processors send corresponding requests to the directory. The directory "knows" which nodes store the memory line in their caches and forwards the request to those nodes only. The directory acts as the point that orders memory accesses and thus enforces serialisation of writes. The directory is located in the same node as the memory that it describes (see Figure 9). Clients can compute the home processor of a memory block from its address in order to forward their requests to this node.

The basic FSM of directory-based protocols is the same as the one in Figure 8a, except now it is replicated across individual processor caches and the

directory. Caches contain only the status bits that describe whether the cache line is invalid, shared or modified. The directory also stores a bit vector describing which processors have copies of the memory line. Therefore, the directory size grows linearly with the memory size and the number of nodes in the system and can become very large for large-scale multiprocessors. Such systems use various techniques for directory size reduction. One popular technique is to replace the directory with a cache that describes the status of the most recently used memory blocks only, rather than the entire memory.

5 Memory consistency models

The cache coherence property described in the previous section constraints possible orders in which different processors in the system observe writes to a single memory location. A memory consistency model is a broader definition of the memory system behaviour, which specifies ordering constraints imposed on all memory accesses in the system.

The strictest memory consistency model used in practice is *sequential consistency* proposed by Lamport [Lam79]. According to Lamport, 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. The main benefit of this model is that it allows a programmer to reason about a multiprocessor system as if it was a uniprocessor system, which enormously simplifies implementation of concurrent algorithms.

The problem with sequential consistency is that it disallows most of hardware and software (compiler) optimisations that exploit instruction-level parallelism. In a system that executes all instructions strictly in the program order and blocks on all memory accesses, cache coherence protocols described in the previous section are sufficient to ensure sequential consistency. However, most existing systems do not follow these restrictions. Below, I enumerate possible types of sequential consistency violations and give examples of ILP optimisations that can lead to these violations. The "X→Y" notation refers to a program order relaxation which leads to operation Y that follows X in the program order being executed before X.

W→R relaxation

Relaxation of the program order between a write and a following read *from a different location* can violate sequential consistency by returning a read value that would be stale in *any* global sequential order. For example, if in the program in Figure 10 both processors perform flag reads before writing new flag values, then both reads can return zeros and, as a result, both processors will enter the critical section simultaneously, which would not be possible in a sequentially consistent system. In the majority of modern processors, such reordering can happen for several reasons. First, from the point of view of an individual

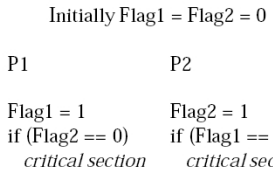


Figure 10: Example for sequential consistency.

processor, memory accesses to different locations do not constitute a data hazard and can be safely reordered to achieve better pipeline utilisation. Even if two operations are separated by a branch instruction, they can still be reordered during speculative execution. Second, most modern processors use write buffers, which means that the write operation can complete before the value actually reaches the memory bus. The following read operation can bypass the queued write, which effectively reorders the effects of the two operations. A similar effect can be observed in systems with non-blocking caches: the write operation can complete before the corresponding request is put on the memory bus, and the following read can be satisfied from the cache. Finally, reordering can be introduced statically by an optimising uniprocessor compiler that assumes the sequential memory model.

W→W relaxation

Reordering writes performed by a processor to different memory locations can lead to another processor observing the order of writes that is different from the one defined by the programmer and hence violates sequential consistency. For example, if the program executed by processor P1 in Figure 11 is reordered in such a way that the `Head` is modified before the `Data` field of the `Task` structure is initialised, than one of the other processors may observe an invalid state of the `Task` structure. Similarly to the previous case, such reordering can be caused by either out-of-order pipeline or can be introduced by a compiler. In a bus-based system, writes are typically executed in the order in which they are issued by the pipeline. However, in a system based on an interconnection network, writes can be reordered by the network. For example, a write to a nearby memory module can complete before a write to a remote module even if the latter was issued first.

R→W and R→R relaxations

Reordering a read with one of the following reads or writes can result in reading stale or otherwise incorrect data. For example, if processor P2 in Figure 11 reads the `Data` field before reading `Head`, the obtained value of `Data` can be stale. In this example, reordering can happen as a result of overlapping of the two reads operations in a system with multiple memory modules.

```
P1                                      P2, P3, ..., Pn

while (there are more tasks) {          while (MyTask == null) {
  Task = GetFromFreeList();               Begin Critical Section
  Task → Data = ...;                      if (Head != null) {
  insert Task in task queue                 MyTask = Head;
}                                           Head = Head → Next;
Head = head of task queue;                }
                                          End Critical Section
                                        }
                                        ...  = MyTask → Data;
```

Figure 11: Example for sequential consistency.

```
                Initially A = B = 0

P1      P2              P3

A = 1
        if (A ==1)
          B = 1
                        if (B==1)
                          register1 = A
```

Figure 12: Example for sequential consistency.

Reading own writes early

A system with non-blocking caches can return a value of a write in a subsequent read before the write propagates to all processors in the system. Luckily, this important optimisation does not compromise sequential consistency.

Reading others' writes early

A more serious problem is that of maintaining a global ordering of writes issued by different processors. While writes to the same memory location are serialised by a cache coherence protocols, writes issued to different addresses can arrive to different processors in different orders in a multiprocessor with an interconnection network. For example, if in Figure 12 the write of variable A reaches P3 after the write of B by P2, then the value read by P3 will be stale. This problem does not exist in systems that use invalidate-based cache coherence protocols, where the new data value can only be sent to a remote processor after the invalidation has been acknowledged by all processors in the system that stored a copy of the memory block in their caches.

As can be seen from the above discussion, a straightforward implementation of sequential consistency would defeat many optimisations used in modern processors and thus destroy the performance of the resulting system. Two solutions to this problem have been proposed. The first solution implemented in the MIPS R10000 architecture is to maintain sequential consistency but hide the resulting latency using speculative execution. The processor executes the

Table 1: Relaxed consistency models.

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

program using usual ILP optimisations but does not commit the results until it becomes clear that they do not violate sequential consistency. If the processor detects a memory consistency violation, it rolls back and restarts the uncommitted computation.

The second solution is to sacrifice the convenient sequential view of memory in order to enable some of the ILP optimisations. Such a tradeoff is provided by *relaxed consistency models*. Relaxed consistency models rely on programmer's knowledge of the program logic to determine situations where the consistency guarantees supported by the model are insufficient for correct program execution and to explicitly issue serialisation instructions to enforce stronger memory ordering at these specific points in the instruction stream. Table 1 summarises relaxations supported by several consistency models.

The set of serialisation instructions supported by a specific architecture is called the *safety net* of this architecture. Serialisation instructions (or memory barriers) can be either fine grained or coarse grained. One example of a coarse grained barrier is the PPC `sync` instruction, which guarantees that all instructions preceding it complete their execution before any of the following instructions starts executing. In contrast, fine-grained barriers enforce only partial ordering of instructions. For example, the IA-32 `lfence` instruction ensures that any memory loads issued before it will complete their execution before any of the following instructions starts executing. A memory barrier may require global synchronisation among all processors in the system. Therefore, barrier instructions are quite expensive and should be used with care, especially in large-scale systems.

6 Operating systems for shared-memory multiprocessors

This section highlights some interesting interactions between shared memory organisation and operating system architecture. In a multiprocessor computer,

an operating system is a shared resource and, therefore, a potential bottleneck. There are three main sources of contention that can be found in a multiprocessor operating system:

- **Locks.** In order to provide safe access to resources shared among multiple processors, they need to be protected by locks. The purpose of a lock is to serialise accesses to the protected resource by multiple processors. Undisciplined use of locking can severely degrade performance of the system or even cause a livelock. This form of contention can be reduced by using fine-grained locks, avoiding long critical sections, replacing locks with lock-free algorithms, or, when possible, avoiding sharing altogether.

- **Shared data.** Accesses to a shared data item by multiple processors (with one or more of them modifying the data) are serialised by the cache coherence protocol. Even in a moderate-scale system, serialisation delays can have significant impact on the system performance. In addition, bursts of cache coherence traffic saturate the memory bus or the interconnection network, which also slows down the entire system. This form of contention can be eliminated by either avoiding sharing or, when this is not possible, by using replication techniques to reduce the rate of write accesses to the shared data. One example of such techniques is the scalable spin lock algorithm proposed in [MCS91]. This approach has been generalised in the Tornado system, which supports a general-purpose clustered objects mode [GKAS99].

- **False sharing.** This form of contention arises when unrelated data items used by different processors are located next to each other in memory and, therefore, share a single cache line. The effect of false sharing is the same as that of regular sharing — bouncing of the cache line among several processors. Fortunately, once it is identified, false sharing can be easily eliminated by padding each data item to the cache line size or otherwise adjusting the memory layout of non-shared data.

A perfectly *scalable operating system* should increase its throughput linearly with the number of processors. There exist two conceptual approaches to developing a scalable operating system. The first, evolutionary, approach is to take an existing OS and modify it to support a large number of processors by identifying and eliminating all points of contention. However, scaling an existing system turned out to be much more difficult than it may appear [CW05, BH03], which prompted some researchers to design a scalable OS from scratch. This approach was followed by Hurricane, Tornado, and K42 systems [GKAS99]. These systems use techniques like replication and per-CPU data structures to achieve near-linear scalability.

Apart from eliminating bottlenecks in the system itself, a multiprocessor operating system developer should provide support for efficiently running user applications on the multiprocessor. Some of the aspects of such support include

mechanisms for task placement and migration across processors, physical memory placement in NUMA systems (the OS should guarantee that most of the memory pages used by an application are located in the local memory), and scalable multiprocessor synchronisation primitives.

References

[AG95] Sarita Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. Technical Report 9512, Rice University, Electrical and Computer Engineering Department, September 1995.

[BH03] Ray Bryant and John Hawkes. Linux scalability for large NUMA systems. In *Proceedings of the Ottawa Linux Symposium*, 2003.

[CSG99] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: a Hardware/Software Approach*. Morgan Kaufmann, 1999.

[CW05] Peter Chubb and Darren Williams. Linux scalability — from the micro to the HUGE. In *Proceedings of the 6th Linux.Conf.Au*, Canberra, ACT, April 2005.

[GKAS99] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximising locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 87–100, New Orleans, LA, USA, February 1999. USENIX.

[HP03] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28:690–1, 1979.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronisation on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9:21–65, 1991.