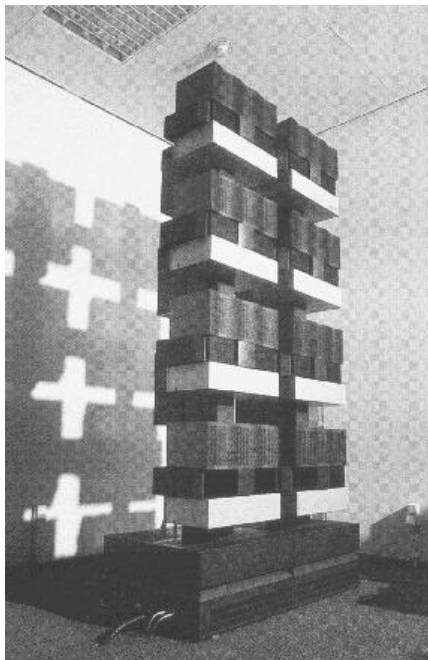


Introduction to Parallel Architecture

Andy Pimentel

andy@science.uva.nl



Contents

Preface	1
1 Processors in parallel systems	3
1.1 High performance processors	3
1.1.1 Very Large Instruction Word (VLIW) processors	5
1.1.2 Vector processors	7
1.2 Caching	7
1.2.1 Direct mapped cache	8
1.2.2 Fully associative cache	9
1.2.3 Set-associative cache	9
1.2.4 Cache strategies	9
2 Interconnection networks	11
2.1 Direct connection networks	12
2.2 Indirect connection networks	14
2.2.1 Busses	14
2.2.2 Multistage networks	16
2.2.3 Crossbar switches	18
2.3 Direct versus Indirect networks	20
2.4 Packet switching	21
2.5 Routing techniques	23
2.5.1 Local routing techniques	25
2.6 Complex communication support	29
3 Multicomputers	31
3.1 VSM and SVM	32
3.2 Real multicomputers	33
3.2.1 The IBM SP2	34
3.2.2 The Parsytec CC	35
4 Multiprocessors	37
4.1 Cache coherency in shared memory machines	37
4.2 Uniform Memory Access (UMA)	40
4.3 Non Uniform Memory Access (NUMA)	42
4.3.1 Latency hiding	42
4.3.2 Disadvantages of CC-NUMA	44
4.4 Cache Only Memory Architecture (COMA)	44
4.5 COMA versus CC-NUMA	45
4.6 Simple COMA (S-COMA)	45
4.6.1 Reactive NUMA (R-NUMA)	46
4.7 Cache coherency revisited	46
4.7.1 States and state transitions	48
4.7.2 Cache coherency in a cache hierarchy	51
4.7.3 Scalable Coherent Interface (SCI)	52

4.8 Synchronization	52
4.8.1 Barrier synchronization	54
4.9 Disk storage considerations	54
4.10 Real multiprocessors	55
4.10.1 The SGI Origin 2000	55
4.10.2 The Cray T3D	56
4.10.3 The Cray MTA	58
5 Other parallel machines	61
5.1 Supercomputers	61
5.2 SIMD array architectures	62
Appendix: example examinations	65

Preface

This is the syllabus for the Architecture part of the course Introduction Parallel Computing & Architecture. It gives an overview of the hardware and low-level software that enable high-performance parallel computing. Moreover, problems and possible solutions related to these parallel computer architectures will also be discussed. Essentially, this syllabus addresses a broad range of topics but can do this only with a limited amount of detail (due to time constraints since there are only six lectures). For more details on specific topics, the interested reader is referred to one of the books listed below.

1. D. Sima, T. Fountain and P. Kacsuk, “Advanced Computer Architecture, A Design Space Approach”
2. J. L. Hennessy and D. A. Patterson, “Computer Architecture, A Quantitative Approach”, 2nd ed.
3. D.E. Culler, J. P. Singh and A. Gupta, “Parallel Computer Architecture, A Hardware/Software Approach”

Book 1 (Sima et al.) gives a nice overview of single-processor and parallel architectures. With respect to parallel computer architecture, it provides a bit more detail than this syllabus. Book 2 is the bible for single-processor architecture. Regarding parallelism, this book mostly deals with instruction-level parallelism. However, this book also contains one or two chapters on parallel systems. Especially when you consider graduating in the ‘computer systems’ variant of the UvA’s computer science study, this book is a must (you will need it, for example, for the Advanced Computer Architecture course). Book 3 is a very good and detailed book on parallel computer architecture. Possibly, this book will become the bible in this area. So, if you are in need of details, then this book will most probably provide them to you.

Outline of the syllabus

The syllabus roughly consists of three parts, which refer to the main building blocks of parallel systems:

- Processors
- Networks
- Memory hierarchy

In the first part, some of the topics treated in the Computer Organization course will be revisited as we will discuss the types of processors that are or can be used in today’s parallel systems. We will also have a look at caches again, since basic understanding of caching is of eminent importance when we are going to discuss shared-memory parallel architectures.

The second part of the syllabus deals with network technology, which is of course needed to connect multiple processors together in a parallel system. Issues such as network topology, switching and routing are key ingredients that will be discussed.

In the third part, different scenarios for the memory hierarchy of a parallel system will be presented. This part basically can be subdivided into two smaller parts: the first one discussing distributed-memory parallel machines (multicomputers) and the other one dealing with shared-memory parallel machines (multiprocessors). Both parts are concluded with the description of several ‘real-world’ commercial machines that apply the previously discussed methods and techniques.

Regarding the parallel machine architectures we will describe in this syllabus, the focus will be on MIMD (Multiple-Instructions, Multiple-Data) architectures. During the past decade, this class of parallel architectures has been by far the most popular and influential one. The class of traditional SIMD (Single-Instruction, Multiple-Data) machines, which was popular during the seventies and eighties, more or less vanished from the arena and has evolved into the SPMD paradigm for MIMD machines.

Another remark that needs to be made is that the author will interchangeably use the terms ‘processor’ and ‘node’ when referring to a processing element of a parallel computer. Unless specified otherwise, one can assume that a ‘node’ is identical to a single processor (as we will see later on, in some parallel machines a node may also consist of multiple processors).

This is the very first edition¹ of this syllabus. This means that there still may be some typos and other small errors in the text. You are encouraged to report any typos/bugs/etc. to the author. Questions or other types of comments are, of course, also welcome. You can reach the author at andy@science.uva.nl.

¹In previous years, a syllabus was deemed unnecessary because there was no written examination for this course. Instead, there was paper assignment.

Chapter 1

Processors in parallel systems

From the Parallel Computing part of this course you should have learned that sequential performance is not to be forgotten. This is because Amdahl's law implicates that programs contain code segments which are inherently sequential. This has influenced to choice of the processors used in todays parallel machines. Where in the past parallel computers were often made out of special-purpose processors specifically designed for certain a parallel machine, current parallel computers typically use commodity microprocessors. This chapter will shed some light on why this move to commodity processors happened.

A good example of a special-purpose processor which was very popular in the eighties for building multicomputers (distributed-memory machines) is the Transputer. A Transputer consists of a processor-core, a small SRAM memory, four communication channels and a DRAM main-memory interface, all on a single chip. In those days, it was the perfect building block for building parallel computers: simply connect the communication channels together to form a network of Transputers. The reason why the Transputer vanished from the parallel computing field is the fact that it lacked the computational power to cope with increasingly demanding parallel applications. In the late eighties and early nineties, commodity RISC processors *were* able to provide such computational power. Moreover, commodity processors are relatively cheap (they are produced in large numbers) and usually have a large design team behind them which implies that these processors are well-tested and hard to beat performance-wise. In other words, by using commodity processors one can "ride the wave" of state-of-the-art microprocessor technology. So, enough reasons for designers of parallel machines to choose commodity processors as a building block for their machines.

Nowadays, there are not many companies (left) that produce parallel machines. This is because the user community of these machines is still relatively small and does not seem to be growing. Many producers of parallel computers also produce commodity microprocessors. Evidently, these companies use their own microprocessors in their parallel machines. For example, SGI uses MIPS microprocessors, IBM uses its own POWER processors, Sun uses Sparc processors and Cray uses DEC (now Intel) Alphas.

As modern parallel computers use commodity microprocessors, they exploit parallelism at several levels. Naturally, coarse-grain task/process-level parallelism is exploited since they have multiple processors operating in parallel. In addition, within a single processor instruction-level parallelism and sometimes fine-grain data parallelism is also exploited. In the (near) future, even thread-level parallelism may also be exploited within commodity processors. We will come back to each of these types of parallelism later on.

1.1 High performance processors

Todays microprocessor market, and therefore also the parallel machine market, is dominated by RISC (Reduced Instruction Set Computer) and 'RISCy' processors. With the latter, we refer to microprocessors which have a CISC (Complex Instruction Set Computer) instruction set but internally operate like a RISC processor. Intel Pentiums are, of course, the best example of RISCy processors.

Let us shortly summarize the characteristics of (traditional) RISC processors. They

- have a few addressing modes.
- have a fixed instruction format, usually 32 or 64 bits.

- have a load-store architecture: there are dedicated load/store instructions to load data from memory to a register and store data from a register to memory. Arithmetic operations are always performed on registers.
- take heavily advantage of pipelining.

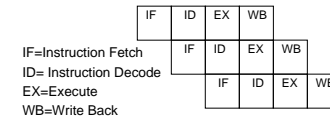


Figure 1.1.

- (almost) have no microcode. Increasingly, this is less true since modern RISC processors often have complex media instructions which are typically implemented using microcode.
- need caching (reducing the performance gap with the main memory) to keep the pipeline(s) filled: new instructions should enter a pipeline quickly enough and memory accesses should not stall the pipeline (i.e., take one cycle).

An instruction pipeline distinguishes between several stages through which the execution of an instruction can propagate. The most straightforward pipeline (see Figure 1.1) consists of four stages:

1. Fetch: fetch an instruction from cache/memory
2. Decode: decode the opcode of the instruction
3. Execute: actually execute the instruction (e.g., addition, load from memory, etc.)
4. Writeback: write the result back to a register/memory

Most of todays microprocessors are *superscalar*. This means that there is not a single instruction pipeline but there are multiple ones operating in parallel (see Figure 1.2). Superscalar processors can therefore execute more than one instruction at the same time (provided that they are independent of each other). So, where a single pipeline may reach an optimum of finishing one instruction per cycle, a superscalar processor may finish n instructions/cycle when it has n pipelines. The effectiveness of superscalar processors is bound by

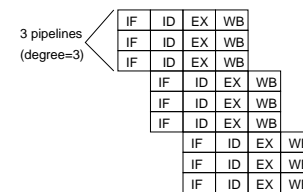


Figure 1.2.

the amount of instruction-level parallelism (ILP) available in applications. To keep the pipelines filled, these processors become increasingly 'aggressive' by allowing out-of-order execution of instructions. This means that (independent) instructions may be executed in a different order than the program order at the hardware level, and this is done completely transparent to the user (he/she does not notice this reordering).

Another method applied by many modern microprocessors in order to increase the throughput of instruction pipelines, and which is often used in combination with the superscalar approach, is *superpipelining*. In superpipelining the work done within a pipeline stage is reduced and the number of pipeline stages is increased (a stage in the normal pipeline is subdivided into a number of sub-stages as illustrated in Figure 1.3).

This allows for a higher clock frequency (since less work is done per sub-stage). Clearly, superpipelining is a popular technique a lot of processor manufacturers are currently applying to boost the clock frequencies of their processors (e.g., a Pentium 4 has over 20 pipeline stages!). These high clock frequencies may look impressive to ignorant customers but they do not say much about actual performance.

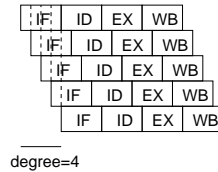


Figure 1.3.

The superpipelining technique may also allow for a better matching of the work actually done per stage. For example, when in a normal pipeline the work performed in a stage takes less than a cycle, the result of the stage will still be available after one clock cycle. So, valuable time is lost. In a superpipelined processor, with a smaller cycle time, the smaller sub-stages used in the pipeline can better match the actual work that needs to be performed. A big disadvantage of superpipelining is that when the pipeline needs to be flushed (e.g., due to a branch instruction), this causes a substantial performance penalty since the states of a large number of instructions that are active in the pipeline need to be cleared. In Figure 1.3, a 4-way superpipelined version of the pipeline in Figure 1.1 is shown. The stages of the original pipeline have been divided into four sub-stages and the clock frequency has been quadrupled.

In the table below, several modern superscalar microprocessors are listed together with the number of parallel execution stages (called 'execution units') they have in their pipelines, the issue-rate and their performance measured using the Spec2000 benchmark (Int = Integer performance, FP = Floating Point performance). The number of execution units denote the ILP that can be exploited by a processor, while the issue-rate relates to the number of instructions that can enter one of the execution units (= stages) per cycle. Normally the issue-rate is less than the number of execution units since some execution units may take longer than 1 cycle to produce their results.

Processor	Clock (MHz)	Execution units	Issue rate	Spec2000 Int	Spec2000 FP
Intel Pentium 4	1800	7	6	619	631
Sparc Ultra III	900	9	4	467	482
Dec Alpha 21264	1001	6	4	621	756
HP PA8700	750	10	4	603	581
MIPS R14000	500	5	4	427	463
Power3-II	450	8	4	316	409

For a much more detailed discussion on these superscalar processors and how they obtain such remarkable processing speeds, you are referred to the Advanced Computer Architecture course (which also happens to be lectured by the author of this syllabus :-).

The previously discussed (superscalar) microprocessors are general purpose processors which are capable of efficiently processing a wide range of applications. There are, however, other processor types which target a less broad application domain but which are capable of obtaining better performance for these applications. Examples are digital signal processors (DSPs), Very Large Instruction Word (VLIW) processors and vector processors. We will briefly treat the latter two of these processor types.

1.1.1 Very Large Instruction Word (VLIW) processors

Very Large Instruction Word (VLIW) processors are derived from horizontal microprogramming and super-scalar processing. A generic VLIW processor architecture is shown at the top of Figure 1.4. It consists of

multiple functional units (integer ALUs, floating-point ALUs, Load/Store units, Branch units, etc.) which resemble the execution stages (units) in superscalar processors. The functional units are connected to a large register file (> 128 registers!). Instructions in VLIW processors are, as the name already suggests, very large; they contain a number of operation slots in which RISC-like instructions (called operations) can be placed (see the picture in the middle of Figure 1.4). The operations within a single instruction can be performed in parallel and are routed towards the appropriate functional units for execution. So each part of an instruction controls different functional units (which is similar to how horizontal microprogramming works). Typical for the VLIW approach is that the compiler fills the instructions with operations. It therefore needs to find out which operations can be performed in parallel. The filling of instructions with operations in such a way that instructions are filled as much as possible (i.e., minimizing the number of unused operation slots) is called *scheduling*.

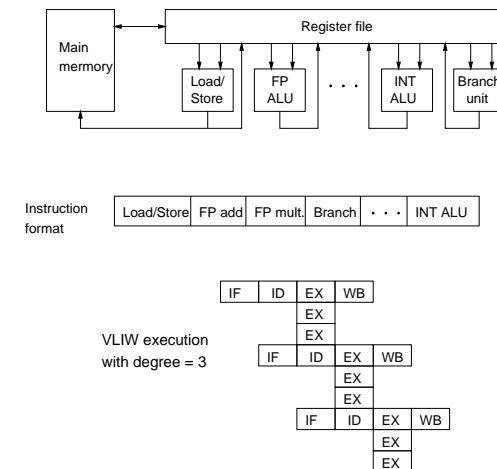


Figure 1.4.

At the bottom of Figure 1.4, the pipeline parallelism obtained by the VLIW approach is illustrated (assuming 3 operation slots per instruction). After a VLIW instruction has been fetched and its operations have been decoded, the operations are dispatched to the functional units in which they are executed in parallel.

Since the VLIW approach determines parallelism at compile-time (static scheduling), the success heavily depends on the quality of the compiler. The advantage is that it does not require complex hardware (implying low costs) to find parallelism at run-time like superscalar processors do. On the other hand, the compiler tries to find as much as possible parallelism in applications and therefore needs to make predictions on their run-time behavior (e.g., which branch will be taken and which one not). For this reason, the VLIW approach only performs well on code that is predictable at compile-time. Examples of such applications can be found in the domains of scientific and multimedia computing. These applications contain many predictable loops in which computations on arrays/matrices are performed.

A drawback of VLIW processors is that their object code is less compact since No-Ops (No-operation instructions) are placed in operation slots which cannot be filled due to a lack of parallelism. In addition, the object code is hard to keep compatible between processor updates, e.g., when the number of functional units is changed in the processor architecture this affects the instruction format.

A well-known example of a VLIW processor is the Philips TriMedia which targets multimedia applications (e.g., it is used in set-top boxes for TVs). Another example is the new IA64 architecture from Intel. This processor architecture is a mix of superscalar RISC and VLIW.

1.1.2 Vector processors

Vector processors generally are not stand-alone processors but are co-processors used next to a general-purpose microprocessor. Unlike normal processors which work with scalars (integers, floats, etc.), a vector processor can operate on whole vectors (an array of scalars). Vector processors are either register-register (vectors are retrieved from special vector registers and written back to one as well) or memory-memory (vectors are retrieved and written back from/to memory). Two examples of a register-register vector operation are shown at the top of Figure 1.5. At the bottom of the same figure, the pipeline parallelism is illustrated. A vector instruction is fetched and decoded after which for each element of the operand vectors a certain operation is performed (each vector element gets the same treatment). So, whereas in normal processors a vector operation requires a loop structure in the code, a vector processor can perform the operation using a single instruction. This means that the loop overhead is reduced by the ‘doing-it-all-in-hardware’ approach of vector processors. To make it even more efficient, vector processors are often able to chain several vector operations together. This means that the results from one vector operation are streamed into another one as an operand.

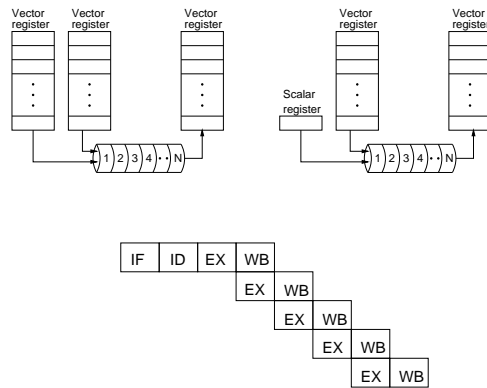


Figure 1.5.

Needless to say is that vector processors target applications in which a lot of vector operations are available. Traditional scientific applications belong to this application class. Multimedia applications also contain many vector operations, albeit operations on short vectors only. Currently, quite some research is performed to make proven vector technology with its traditional focus on long vectors (since it was driven by scientific applications) also work for multimedia applications with their short vectors.

1.2 Caching

Caches are an essential element of today's high-performance microprocessors. While microprocessors roughly become twice as fast every 18 months, DRAM chips for the main memory cannot keep up with this pace. Therefore, caches were introduced in order to bridge the speed gap between processor and memory.

A cache is a fast and small SRAM memory which exploits *locality*. There are two types of locality:

- Temporal locality : a referenced item tends to be referenced soon again (e.g., instructions: loop structures)
- Spatial locality : items close to a referenced item tend to be referenced soon (data: arrays, instructions: instruction_{i+1} is most likely executed after instruction_i)

There are several types of caches which are being applied in modern processors. For example, Translation Look-aside Buffers (TLBs) are caches for storing recent virtual to physical address translations. These trans-

lations can be quite expensive in terms of performance as they often need multiple memory accesses to read the page-table. Luckily, TLBs are able to obtain hit ratios of over the 90% which prevents that virtual to physical address translation becomes a performance bottleneck.

Another well-known class of caches are the instruction and data caches. To this class belong the caches that only store instructions or data and those that are unified and store both instructions and data in a single cache. For the sake of discussion, we will focus on this class of data/instruction caches in the remainder of this section. The presented implementation methods are, however, also applicable to other types of caches.

Roughly speaking, there are three ways to implement the mapping of data located in the main memory to a location in the (much smaller) cache: direct mapped, fully associative and set-associative. In all implementations, the data granularity that is cached (and thus is mapped into the cache as a single entity) is called a *cache block*. Typically, cache blocks have a size of 16 to 256 bytes.

1.2.1 Direct mapped cache

In direct mapped caches, there is a one-to-one mapping of addresses in the main memory to cache locations using a modulo function. To illustrate this, let's assume the example cache in Figure 1.6. This cache has 8 entries which each can hold a cache block and the cache block size is 16 bytes. In a direct mapped implementation of our example cache, the first 16-byte block in main memory maps onto the first entry in the cache, the second block in main memory to the second cache entry, and so on. This scheme implies that main memory block 8 again maps to the first entry in the cache.

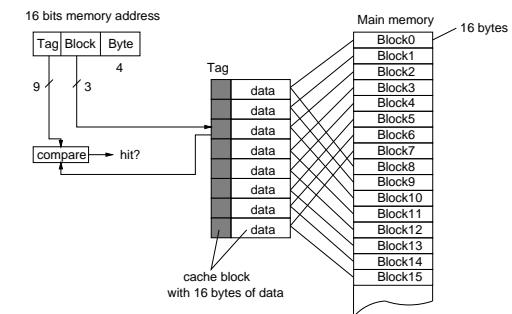


Figure 1.6.

Since multiple main memory blocks can be mapped to the same cache entry, the processor must be able to determine whether a data block in the cache is the data block that is actually needed. This is done by storing a *tag* together with a cache block. This tag constitutes the most-significant address-bits of the memory location the cache block in question stores. Consider, for example, Figure 1.6 again. Here, we assume 16-bit addresses. Accessing the direct mapped cache with these 16-bit addresses is done as follows. The 4 least-significant bits are used to address the right byte in the 16-byte cache blocks. The next 3 bits determine in which cache entry the processor should look for the required data. So, these are the mapping bits. Finally, the remaining 9 most-significant bits are used for the tag. These tag bits are compared with the tag bits stored in the cache at the selected cache entry (selected by the 3 mapping bits). If the tags are identical, then there is a cache hit, implying that the data in the cache block is the data actually needed by the processor.

Direct mapped caches are, because of the straightforward mapping, relatively simple to build and allow for high-speed access. However, because this mapping is also very rigid, direct mapped caches may yield poor hit ratios. For example, if you are accessing an array by element numbers 0, 8, 16, etc., then every array access maps to the same cache location, thereby removing the data from the previous access. By making a direct cache very large, these effects can be reduced. For this reason, direct mapped caches are often used as 2nd-level caches (which are often placed off-chip and can therefore be large) in processors.

1.2.2 Fully associative cache

A fully associative mapping is more or less the opposite of a direct mapping. It allows for placing a cache block anywhere in the cache. According to some replacement policy, the cache determines a cache entry in which it stores a cache block. Consider Figure 1.7 to illustrate how such a cache is accessed. Again, we assume 16-bit addresses, a cache with 8 entries and cache blocks of 16 bytes. The latter means that the 4 least-significant bits of an address are again used to address the right byte in a cache block. There is no fixed mapping, so there are no mapping bits. Instead, because a required cache block can reside anywhere in the cache, all cache entries need to be searched. This means that the remaining 12 bits of an address are tag-bits and these tag-bits are compared in parallel with all tags stored in the cache. If there is a tag match, then there is a cache hit. In that case, the data from the cache block with the matching tag can be used by the processor.

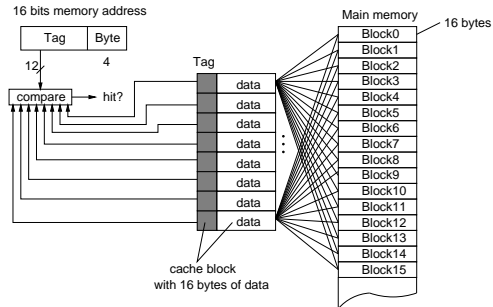


Figure 1.7.

Clearly, fully associative caches have a flexible mapping which minimizes the number of cache-entry conflicts (i.e., they may yield high hit ratios). However, a price must be paid since a fully associative implementation is expensive (e.g., it must be capable of doing a large number of comparisons in parallel). For this reason, appliances of fully associative caches never use large cache sizes. Examples of these appliances are TLBs and branch history tables (of which the explanation is beyond the scope of this syllabus).

1.2.3 Set-associative cache

A set-associative mapping can be seen as a combination of a direct mapping and a fully associative mapping. In a set-associative cache, the cache entries are subdivided into cache sets. Similar to a direct mapping, there is a fixed mapping of memory blocks to a set in the cache. But inside a cache set, which can hold several cache blocks, a memory block is mapped in a fully associative manner. To illustrate this, consider Figure 1.8. It depicts a 2-way set-associative cache with 16-byte blocks. The "2-way" means that each set contains 2 cache blocks. This means that the 8-entry cache is subdivided into 4 sets. Memory block 0 maps to the first set, memory block 1 to the second set, and so on. Memory block 4 again maps to the first set in the cache.

In the example of Figure 1.8, the least-significant bits are still used to address the byte in the cache block. The next 2 bits determine in which cache set the data should reside. To find out if there is a cache hit, the tags from all cache blocks in the set are compared in parallel with the 10 tag-bits provided by the processor. So, compared to a fully associative solution, the set-associative approach reduces the number of comparisons that need to be performed in parallel as it now equals to the number of cache blocks inside a set. Studies have shown that 4-way set-associative caches already approach the performance of fully associative caches while being less expensive. Set-associative caches are therefore a popular choice for 1st-level cache implementations (these on-chip caches can only be of moderate size and must yield a good hit ratio).

1.2.4 Cache strategies

Besides a mapping mechanism, caches also need a range of strategies that specify what should happen in the case of certain events. In this section, we discuss several of these cache strategies.

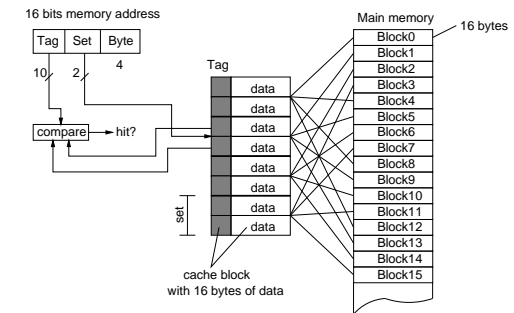


Figure 1.8.

In (set-)associative caches, the cache must determine which cache block is replaced by a new block entering the cache. There are several well-known replacement strategies, which are also applied in other fields such as in operating systems: Random, First-In First Out (FIFO) and Least Recently Used (LRU). Many caches apply some sort of LRU replacement (sometimes a type of pseudo-LRU instead of real LRU). Studies have shown that in most cases the LRU strategy performs best.

Several other cache strategies relate to write actions. For example, when the processor performs a store instruction, the cache can follow one of two write policies. First, it can write the value of the store instruction into the cache *and* into the main memory. This is called a *write-through* cache since it writes the value through the cache to the memory. Alternatively, the cache can only update the cache at a write action. This is called a *write-back* cache.

The advantage of a write-through cache is that the main memory is always consistent with the cache (later on we will see why this may be useful). The disadvantage is, however, that it increases bus/memory traffic since every write action is propagated to the main memory. This is not done in a write-back cache, which is therefore more efficient in terms of memory traffic. However, such a cache needs an extra status bit per cache block specifying whether a cache block is dirty (written to) or not. In the event a dirty cache block is removed from the cache (e.g., it has been selected by the replacement strategy), it cannot be simply discarded but it needs to be written back to memory. Another potential drawback of write-back caches is that context switches of the OS may be slower than is the case with a write-through cache. Because switching to a new process often implies that a different range of addresses is used, a context switch may trigger a large number of replacements in the cache. With a write-back cache, this means that these replacements may lead to heavy memory traffic when many of the replaced blocks are dirty and need to be written back to memory. In a write-through cache, on the other hand, replacements are cheap: simply discard the cache block.

Another strategy related to write events is the "write-miss strategy". In other words, what should the cache do when a cache miss for a store instruction occurs? In the *allocate-on-write* strategy, the cache allocates a cache block in the cache after which it performs the write action on this block. When the cache implements the *fetch-on-write* strategy, the cache allocates a cache block *and* fetches the data that belongs to this cache block from main memory before performing the write action. Write-back caches often use one of these two write-miss strategies. Write-through caches may apply the *no-allocate-on-write* strategy, in which nothing is done on a write miss. The value is simply written to the memory and the cache contents remain unchanged.

In this chapter, we have presented a brief overview of (issues relating to) high-performance processors applied in modern parallel systems. Again, if you are interested in a more in-depth discussion on microprocessor architecture, then you are referred to the Advanced Computer Architecture course. The next building block of parallel systems we are going to discuss, is the interconnection network which ties the processors together. The network determines the connectivity between processors and therefore has significant impact on the number of processors that can be accommodated by a parallel system.

2.1 Direct connection networks

In direct networks, there are point-to-point connections between neighboring nodes. These networks typically are static, which implies that the point-to-point connections are fixed. Well-known examples of direct networks are rings, meshes, tori and cubes. In Figures 2.1 and 2.2, several example direct networks are shown. Let us briefly discuss the ones which are less self-explanatory.

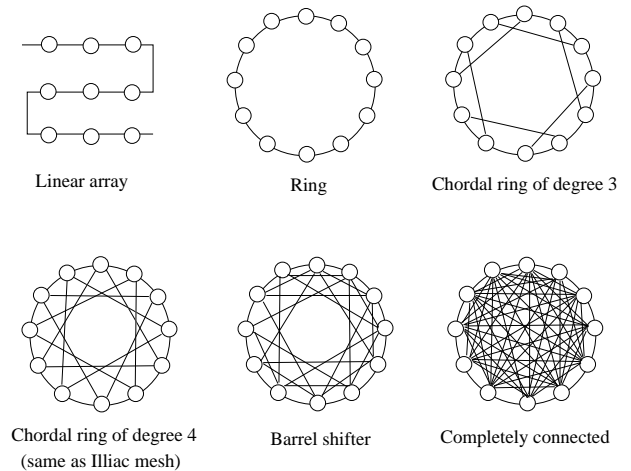


Figure 2.1.

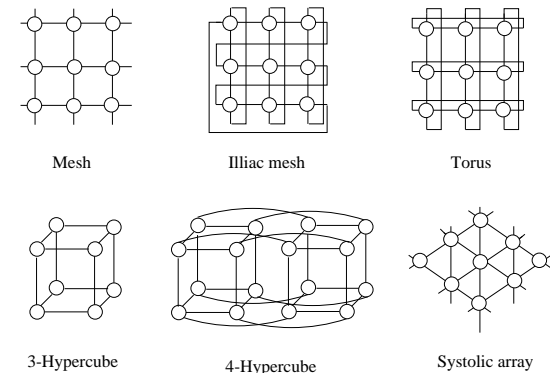


Figure 2.2.

The degree of a Chordal ring refers to the node degree in such a network. Using different degrees for a Chordal ring, a range of well-known networks can be made: a normal ring, an Illiac mesh (named after the Illiac parallel machine which used this network, see also Figure 2.2), up to a completely connected network. A barrel shifter has connections between nodes that have a distance of a power of two. A torus is a mesh with ‘wrap-around’ connections. An n -hypercube contains 2^n nodes connected in n dimensions. Finally, systolic

Chapter 2

Interconnection networks

To make the discussion on networks more easy, we will first familiarize the reader with a number of network properties and definitions.

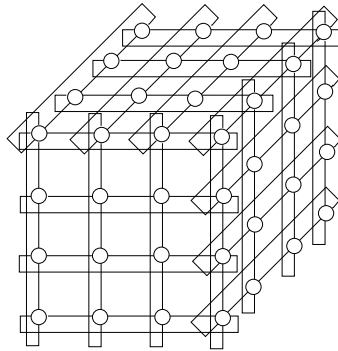
The purpose of a network is to allow for exchanging data between processors in the parallel system. Regarding this data exchange, two important terms need to be introduced: *network switching* and *network routing*. Network switching refers to the method of transportation of data between processors in the network. There are roughly two classes of network switching: *circuit switching* and *packet switching*. In circuit switching, a connection is made between the source and destination processors which is kept intact during the entire data transmission. During this communication, no other processors can use the allocated communication channel(s). This is like a traditional telephone works. Some of the early parallel machines used this switching method, but nowadays they mainly use packet switching. In packet switching, data is divided into relatively small packets and a communication channel is allocated only for the transmission of a single packet. Thereafter, the channel may be freely used for another data transmission or for a next packet of the same transmission.

Network routing refers to the method of *steering* data (messages) through the network. In other words, it defines the route data (messages) should follow through the network in order to reach the destination. Note that it does not specify how the data is transferred along the chosen route (this is defined by the network switching method). Network routing is, of course, highly influenced by the lay-out, or *topology*, of the network. A distinction can be made between direct networks and indirect networks. As will be discussed in the next sections, direct networks have point-to-point communication links between neighboring processors, while indirect networks use shared communication channels.

We end this introduction with listing several other important network definitions/parameters accompanied with a short explanation. In addition, between parentheses we note (if appropriate) whether we should strive for maximizing (\gg) or minimizing (\ll) a given network parameter for performance reasons.

- **Node degree:** number of channels connected to one node (\gg).
- **Diameter of network:** maximum shortest path between two nodes (\ll).
- **Bisection width:** when the network is cut into two equal halves, the minimum number of channels along the cut (\gg). This specifies the bandwidth from one half of the network to the other half.
- **Network redundancy (fault tolerance):** amount of alternative paths between two nodes (\gg). Routing and redundancy are coupled: high redundancy \Rightarrow many routing possibilities.
- **Network scalability:** measure for expandability of the network (\gg).
- **Network throughput :** $\frac{\text{Amount of transferred data}}{\text{time units}}$
- **Network latency:** worst case delay for transfer of a unit (empty) message through the network.
- **Hot-spots:** nodes in the network that account for a disproportionately amount of network traffic. Hot-spots can degrade the network performance by causing congestion.

arrays are networks designed for implementing specific algorithms (they match the communication structure of an algorithm). The systolic array in Figure 2.2 can be used for performing a matrix-matrix multiplication.



K-ary N-cube with K = 4 (radix, # of nodes along side) and N = 3 (dimension)

Figure 2.3.

In Figure 2.3, a k-ary n-cube network is shown. Note that the hidden nodes and connections are not shown in Figure 2.3. The class of k-ary n-cube networks is topologically isomorphic with rings, meshes, tori and hypercubes. The parameter n is the dimension and k the radix, or the number of nodes along each dimension. For example, a k-ary 1-cube network creates a ring, a k-ary 2-cube creates a torus and a 2-ary n-cube creates a hypercube. Because the k-ary n-cube network is isomorphic with so many other networks, it is often used in networking performance studies.

Network	Node degree	Diameter	Bisection width
Linear array	2	$N - 1$	1
Ring	2	$\lfloor \frac{N}{2} \rfloor$	2
Completely conn.	$N - 1$	1	$(\frac{N}{2})^2$
Binary tree	3	$2(\log_2 N - 1)$	1
2D-mesh	4	$2(\sqrt{N} - 1)$	\sqrt{N}
2D-torus	4	$2\lfloor \frac{\sqrt{N}}{2} \rfloor$	$2\sqrt{N}$
Hypercube	$\log_2 N$	$\log_2 N$	$\frac{N}{2}$

Table 2.1.

Table 2.1 shows three network characteristics for several common direct networks. In this table, N equals to the number of nodes in the network. From this table can be seen that linear arrays and rings have a poor diameter and bisection width. For this reason, these networks are hardly applied as the main network in parallel machines. Completely connected networks have very good characteristics but are extremely expensive and hard to realize (especially when scaling to a large number of nodes: the node degree grows with $O(N)$). A binary tree has a node degree of 3 (two children and a parent) and a moderate diameter. However, it has a poor bisection width of only 1 since there is only one root node connecting the two halves of the tree. Meshes and tori have fixed node degrees and moderate diameters and bisection widths. Tori have slightly better network characteristics than meshes because of the wrap-around channels. Finally, hypercubes have a relatively good diameter and bisection width but their node degree grows with $O(\log(N))$. The latter means that higher dimensional hypercubes can become expensive to realize. Looking at Table 2.1, it will not be a

surprise that many parallel computer architectures use a mesh, torus or hypercube topology for their network.

2.2 Indirect connection networks

Indirect networks are also often called dynamic networks since, unlike in point-to-point networks, there are no fixed neighbors. Essentially, the communication topology can be changed dynamically based on the application demands. Indirect networks can be subdivided into three types: bus networks, multistage networks and crossbar switches. We will treat each of these networks in the following sub-sections.

2.2.1 Busses

A generic bus network in the context of a parallel machine is shown in Figure 2.4. It consists of a number of bitlines onto which a number of resources (e.g., processors, memories, I/O-devices, etc.) are attached. Regarding the bitlines, one can distinguish between address lines, data lines and control lines. Sometimes, busses use the same physical lines for addresses and data. In that case, the address and data lines are time-multiplexed. When there are multiple bus-masters (i.e., resources that may initiate a bus transaction) attached to the bus, an arbiter is required. We return to the topic of arbitration later on.

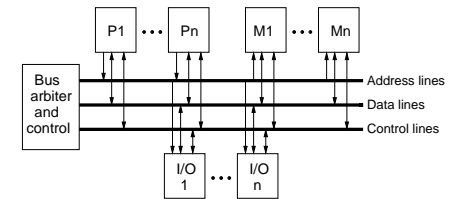
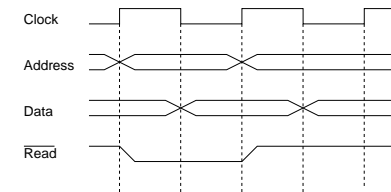


Figure 2.4.

There are synchronous and asynchronous busses. In a synchronous bus, all actions are synchronized using a clock. Consider, for example, Figure 2.5 which shows a bus protocol for a synchronous bus when performing a read transaction. First, an address is put on the address lines and the \overline{read} control signal is pulled down to specify a read transaction. On the next clock edge, when the address has been seen by the resource from which is read, this resource responds by putting the required data on the data lines. On the next clock edge, this data is read from the bus by the reading resource. Naturally, this example is oversimplified since it implies that data can be fetched and put on the bus in a single cycle which in reality is hard or impossible to accomplish.

In an asynchronous bus, the actions are not as predictable (e.g., data is available in n clock cycles) as in a synchronous bus. Therefore, it uses a handshake protocol to indicate when certain events occur. To give an example in the context of Figure 2.5, an extra control would be needed to indicate the validity of the data on the data lines (i.e., signaling when data is put on the data lines). In general, asynchronous busses



Typical bus read transaction

Figure 2.5.

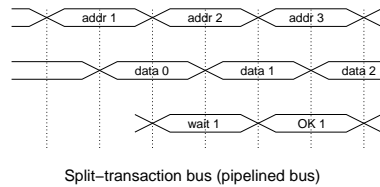


Figure 2.6.

are more complex, more expensive and often less efficient than synchronous busses because of the explicit handshaking protocol, but they are more flexible. It is more easy to attach resources with different speeds on an asynchronous bus than on a synchronous bus. Processor to memory busses generally are synchronous. Well-known examples of asynchronous busses are the VME and SCSI busses.

In a traditional bus, the bus stays occupied during an entire bus transaction, even when the bus is idle because it is waiting for data from a resource. Clearly, this is quite inefficient. To increase the throughput of busses, split-transaction (or pipelined) busses were introduced. In a split-transaction bus, multiple bus transactions can be active on the bus at the same time. Consider Figure 2.6 to illustrate this for a read transaction on a synchronous split-transaction bus. First, an address is put on the address lines. When this address has been picked from the bus by the destination resource, the address lines are ready again to receive a next address. When the resource is ready with fetching the required data, it places the data on the data lines and specifies, using a *tag*, the identity of the resource which requested this data. So, extra control lines are needed to signal this tag information. Although a split-transaction bus improves the throughput, it may also slightly increase the latency of transactions because the bus is now shared between several transactions at the same time.

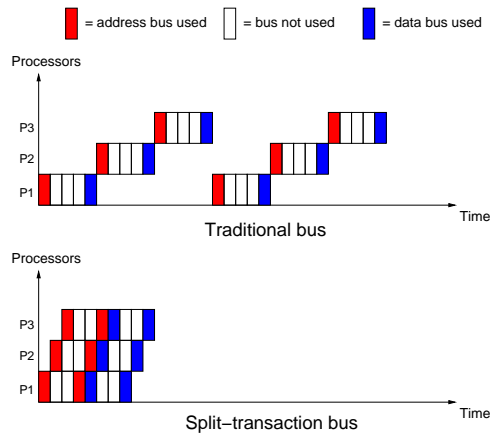


Figure 2.7.

In Figure 2.7 the performance difference between read transactions on a traditional and a split-transaction bus is shown. For both busses we assume that after putting the address on the bus, it takes three cycles before the data is placed on the bus. Moreover, it is assumed that requests from one processor do not need to wait for each other and can therefore be overlapped. This assumption is not entirely unrealistic since in current superscalar processors there usually are multiple load/store units operating in parallel. Looking at Figure 2.7, it is evident that there can be a large performance gain when pipelining bus transactions.

We already mentioned that arbitration is needed when there are multiple masters on a bus. In the case more that one of these masters would like to perform a transaction at the same time, the arbiter should decide which of the masters is granted access to the bus. In Figure 2.8, a centralized arbitration scheme with independent request/grant lines is shown. Whenever a master wants to use the bus, it signals the arbiter using its request line. If it is the only master which requested the bus, then it immediately is granted access to the bus (signaled using the grant line). In the case multiple masters want to use the bus, the arbiter applies some selection policy to determine which master is granted access to the bus. If a master is granted access to the bus, then the 'bus busy' line is activated. As long as this line is activated, the arbiter cannot grant access to another master. This scheme with independent request/grant lines is flexible (allows for different kinds of selection/priority policies) and efficient, but also is expensive due to the many control lines.

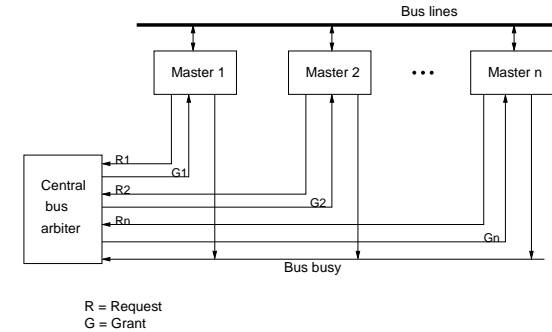


Figure 2.8.

Another centralized arbitration scheme is shown in Figure 2.9. This is so-called *daisy-chained* arbitration. In daisy-chaining, a grant signal is propagated through a chain of masters. When a master receives such a grant signal it may request the bus by activating the request line. When it does not need the bus, it simply forwards the grant signal to the next master in the chain. This scheme is less expensive since it requires less control lines, but because of the (slow) propagation of the grant signal it may take longer before a master can be granted access to the bus. Moreover, this scheme is less fair since the masters in front of the chain have a higher priority than those in the end of the chain.

As one would expect, there also are distributed arbitration schemes, but we do not consider them in this syllabus.

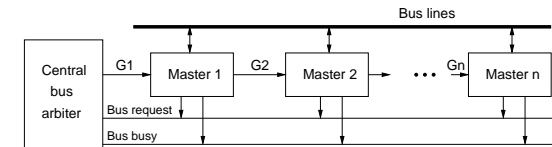
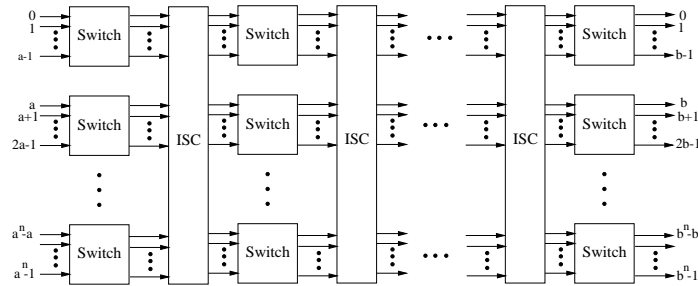


Figure 2.9.

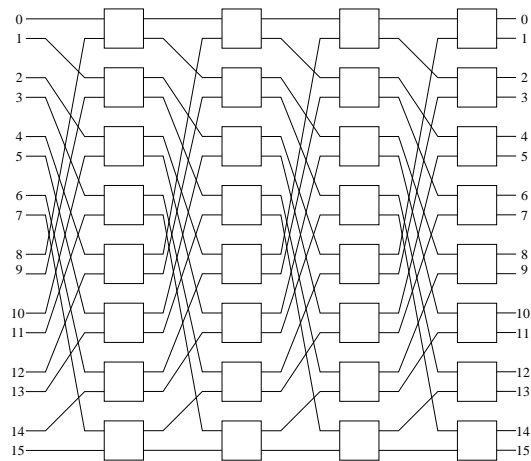
2.2.2 Multistage networks

As its name already suggests, a multistage network consists of multiple stages of switches. A generalized multistage network is shown in Figure 2.10. It consists of $a \times b$ switches which are connected using a specific interstage connection pattern (ISC). Small 2×2 switch elements are a popular choice for many multistage networks. If the multistage network has N inputs, then it needs $\log_2 N$ stages of 2×2 switches, where each stage consists of $\frac{N}{2}$ switch elements, to reach all N outputs. In a multistage network, the number of stages



A generalized Multistage Interconnection Network (MIN) built with a x b switches and a specific InterStage Connection (ISC) pattern

Figure 2.10.



A 16x16 Omega network of 2x2 switches

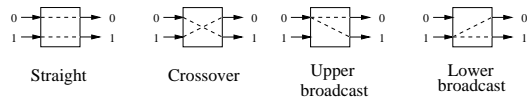
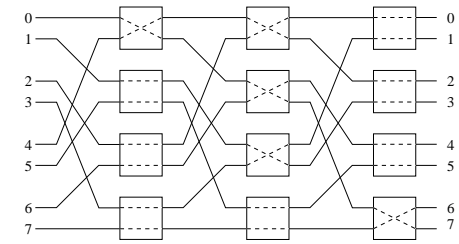
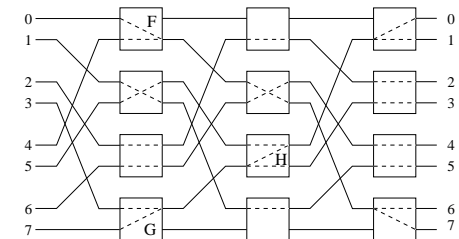


Figure 2.11.



Permutation (0,7,6,4,2)(1,3)(5) without blocking



Permutation (0,6,4,7,3)(1,5)(2) blocked at switches F, G and H

Figure 2.12.

determines the delay of the network. Moreover, by choosing different interstage connection patterns, various types of multistage network can be realized. Well-known examples are Omega networks, baseline networks and Clos networks. Let us look at one of these: the Omega network.

In Figure 2.11, a 16×16 Omega network with 2×2 switches is shown. The switch elements can switch incoming data to its outgoing communication ports in four ways, which is also shown in Figure 2.11. The interstage connection pattern in an Omega network is called a *perfect shuffle*. This is because it works like shuffling a deck of playing cards where the top half of the deck is perfectly interleaved with the bottom half.

Multistage networks can either be *blocking* or *non-blocking*. In a blocking network, not all communication permutations can be performed without a switching conflict. A non-blocking multistage network supports all communication permutations without switching conflicts. To illustrate this, consider the Omega network again. Figure 2.12 shows why the Omega network is blocking. At the top of Figure 2.12, the switch settings to realize the communication permutation (0,7,6,4,2)(1,3)(5) in an 8×8 Omega network are shown. This permutation means that input 0 communicates with output 7, input 7 communicates with output 6, and so on. Finally, input 5 communicates with output 5 in this permutation. Looking at the switch settings, there are no conflicts. This means that this permutation can be performed without blocking. However, at the bottom of Figure 2.12, the switch settings for the permutation (0,6,4,7,3)(1,5)(3) are shown. In this case, there are switch conflicts at switches F, G and H. This means that for some of the incoming communication channels at these switches the communication must be blocked until the required outgoing channel is available again.

2.2.3 Crossbar switches

Conceptually, a crossbar switch consists of a matrix of simple switch elements that can switch on and off to create or break a connection. This is illustrated in Figure 2.13, in which a number of processors are connected to multiple memory banks via a crossbar switch. By turning on a certain switch element in the

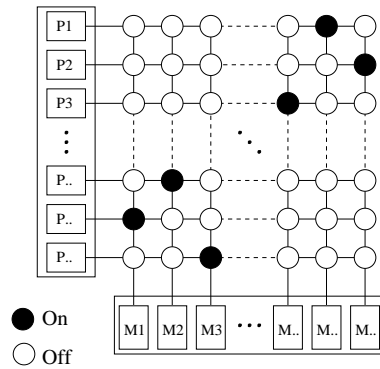


Figure 2.13.

matrix, a connection between a processor and a memory can be made. Of course, multiple switches on a single vertical line cannot be turned on at the same time as this will cause a conflict (i.e., multiple processors cannot communicate with a single memory bank at the same time). On the other hand, by turning on multiple switches for a single horizontal line one can realize a multicast or broadcast operation. Crossbar switches are non-blocking, implying that all communication permutations can be performed without blocking.

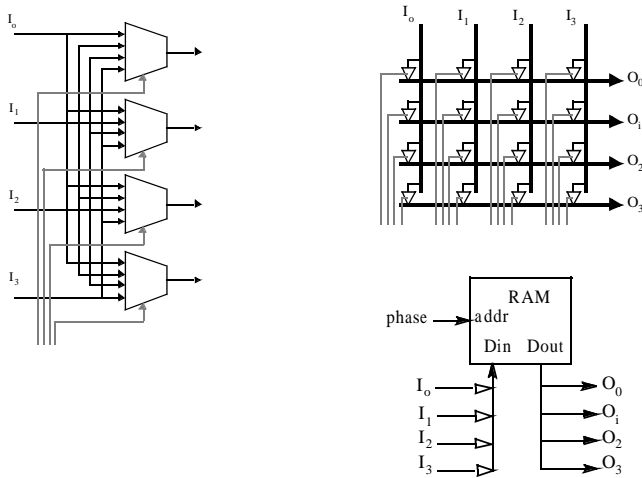


Figure 2.14.

In Figure 2.14, three possible implementations of a 4×4 crossbar are shown. On the left, it shows an implementation using four multiplexers. A selection channel per multiplexer is used to specify which input signal is selected as output signal. The upper right implementation of a 4×4 crossbar is similar to the one shown in Figure 2.13. It consists of horizontal and vertical ‘busses’ together with tri-state drivers (the ∇ -s). These tri-state drivers can switch between the data bitvalues 0 and 1, and a special state: no-value. The

latter is needed to indicate that a switch is turned off. Finally, the lower right picture in Figure 2.14 shows a memory-based implementation of a 4×4 crossbar. By reading/writing from/to specific memory locations, the functionality of a crossbar switch can be established.

In Table 2.2, we summarize some of the characteristics of the three types of indirect networks we have described in the previous sections. Here, we assume n processors on a bus of width w , an $n \times n$ multistage network using $k \times k$ switches with line width w and an $n \times n$ crossbar with line width w . With respect to the performance characteristics, we also assume that the networks do not suffer from contention.

Network characteristics	Bus	Multistage network	Crossbar switch
Min. latency	constant	$O(\log_k n)$	constant
Bandwidth	$O(w)$	$O(w)$ to $O(nw)$	$O(w)$ to $O(nw)$
Wiring complexity	$O(w)$	$O(nw \log_k n)$	$O(n^2 w)$
Switching complexity	$O(n)$	$O(n \log_k n)$	$O(n^2)$
Connectivity and routing capability	One to one and multi-/broadcasts (one at the time)	Non-blocking permutations and multi-/broadcasts	All permutations and multi-/broadcasts

Table 2.2.

The communication latency for a bus and crossbar are constant¹, whereas for a multistage network it is dependent on the number of stages that needs to be traversed (and the number of stages equals to $\log_k n$). Regarding the bandwidth of a bus, this scales with its width. Additionally, the bandwidth of a bus is also affected by its implementation: traditional v.s. split-transaction. For the multistage and crossbar networks, the bandwidth can range from $O(w)$ to $O(nw)$. It is $O(w)$ when there is blocking (multistage network) or when multiple inputs want to communicate with a single output (crossbar). For non-blocking permutations, it is $O(nw)$ since n communication lines of width w are used in parallel. The wiring complexity of a bus scales with the bus-width. For a multistage network, it scales with $n \times w \times \log(n)$: the number of stages times the number of switches per stage times the line width. For a crossbar, the wiring complexity (which includes control lines to turn on/off switches) scales with n^2 : see the non-memory-based implementations in Figure 2.14. The switching complexity of a bus scales with n . Of course, a bus does not have switches but here we take the increasing arbitration complexity into account. For a multistage network, the switching complexity equals to the number of stages times the number of switches per stage. And the switching complexity for a crossbar clearly scales with the number of inputs times the number of outputs. Regarding the connectivity and routing capabilities of indirect networks, a bus offers one-to-one and multi-/broadcast communication at a time (although pipelining allows for overlapping some parts of a transaction). A multistage network provides parallel communication between resources when the permutation is non-blocking. It also allows for multi-/broadcasts. Finally, the crossbar supports non-blocking communication for all permutations and also allows for multi-/broadcasts.

2.3 Direct versus Indirect networks

Let us conclude the discussion on direct and indirect networks with a comparison between the two. In direct networks, there are dedicated communication paths between neighboring processors, whereas in indirect networks there are no point-to-point connections between processors. Communication with a neighbor in a direct network usually is efficient, while the communication with a non-neighboring node is slower since multiple ‘hops’ need to be made through the network (later on, we will see that so-called wormhole routing minimizes this performance degradation). To make these hops through the network (in order to find the destination node), a routing algorithm is needed. Regarding indirect networks, only multistage networks require routing as the switches in the network must be switched correctly. It is obvious that a bus does not need routing and a crossbar only requires a single switch to be turned on (we do not call this routing).

¹Dependent on arbitration delay and protocol for a bus, and on the time to allocate a channel for a crossbar

Direct networks typically scale well to a large number of nodes, while indirect networks are less scalable. A bus, for example, is going to be saturated when attaching a large number of processors to it which all contend for bus access. Scaling multistage networks and, especially, crossbar switches is expensive because of wiring constraints (see Table 2.2).

If the communication traffic is not uniformly distributed over the network, then both direct and indirect networks suffer from congestion. However, the worst case behavior of indirect networks is worse than that of direct networks because of the fact that they use shared communication links. Indirect networks are therefore more prone to tree saturation which is a phenomena that will be introduced in the next section.

To exchange data between processors, direct networks typically apply packet switching. In the next section, we will elaborate on this popular switching method. Indirect networks apply either circuit or packet switching. Crossbars are by definition circuit switched. A traditional bus is also circuit switched, but a split-transactions bus can be regarded as packet switched. In a multistage network, all switches can be set before the communication takes place, thereby creating a circuit switched communication path. Alternatively, the switches may locally decide how to switch dependent on an incoming packet (packet switching).

2.4 Packet switching

In today's packet switched networks, there usually are dedicated (hardware) switch elements that perform the packet switching. So, these switches forward a packet until it has reached its destination node. In the case of a direct network, this means that a switch is part of a node in the network. Such a switch either forwards a packet (when it has not yet reached its destination) or copies it to the node's memory (when it has reached its destination). For indirect networks, we have already seen these dedicated switch elements in multistage networks as they form the stages in such networks.

There are three common packet switching techniques: store & forward switching, wormhole routing², and virtual cut-through switching. In store & forward switching a packet is the smallest entity and the switches in the network require packet buffers to locally store incoming/outgoing packets. A packet that needs to make several hops along switches in the network first *entirely* makes the first hop, then entirely the second, and so on. So, an incoming packet at a switch is not forwarded to the next switch before the entire packet has been received in the local packet buffer. This technique is rather obsolete as it is not very efficient (we will demonstrate this in a little while). In the eighties, a substantial number of parallel machines used store & forward switching because it is relatively simple to implement.

In wormhole routing, a *flit* (FLoW control digIT) is the smallest entity. Typically, flits have a size of one to a few bytes. Data that needs to be transferred, is first separated into packets and these packets are again separated into flits. Like the name wormhole routing suggests, the flits of a packet are transferred like a worm through the network, or in other words, in a pipelined manner. A network switch, after having received a flit, immediately forwards the flit to the next switch and, at the same time, receives the next flit from the packet. This scheme has several consequences. First, only small flitbuffers are needed at switches to store incoming/outgoing flits. When the first flit (the head flit) of a packet is stalled because it needs to wait for a communication channel to become available, all flits in the worm behind it are locally stored in a flitbuffer in the intermediate switch at which they reside. Since flitbuffers are small, wormhole routing is efficient to implement in hardware (compare with store & forward switching in which packet buffers are used which may need to be kilobytes in size). Second, as a packet (consisting of multiple flits) is transferred through the network in a pipelined manner, it typically occupies multiple communication channels. During this occupation, these channels stay reserved for this packet until a channel is explicitly freed by the tail flit (the last flit of the packet). The reason for this channel reservation is that packets may not be split into multiple parts during transmission (i.e., a packet always must be transferred as a single worm). This is because the route information of a packet specifying the path the packet should follow, is located in the first few flits at the head of the worm. By splitting a worm into two halves, the last part would lose its route information and would therefore be unable to find its destination.

In Figures 2.15 and 2.16, the performance of store & forward switching and wormhole routing is illustrated. The figures show a packet transmission from a source (S), along intermediate switches I1, I2, etc. The gray part of the packet refers to the header (head flit, in case of wormhole routing). In Figure 2.16, the

²The term 'routing' in wormhole routing is inherently wrong. Wormhole routing is a switching technique and has nothing to do with routing. Nevertheless, the term wormhole routing has been widely embraced so we will use the term in this syllabus too.

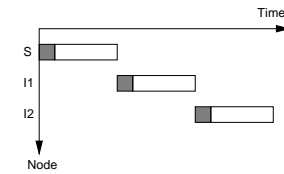


Figure 2.15.



Figure 2.16.

pipelined manner of transmission in wormhole routing can clearly be seen. Also, it is clear that wormhole routing substantially outperforms store & forward switching. In fact, wormhole routing is, in the case network congestion is absent, nearly distance independent: the time to send a packet to a neighboring node or to a node at the other end of the network only marginally differs in terms of latency.

The communication latencies of store & forward switching and wormhole routing can also be expressed by the following equations:

$$T_{s\&f} = \frac{L}{W} \times D$$

$$T_{wormhole} = \frac{L}{W} + \frac{F}{W}(D-1)$$

where L is the packet length in bits, W the channel bandwidth in bits/s, D the distance (number of hops) and F the flit length in bits. Sending a whole packet to the next hop using store & forward switching takes $\frac{L}{W}$. This needs to be done D times (there are D hops), which results in $T_{s\&f} = \frac{L}{W} \times D$. For wormhole routing, it takes $\frac{L}{W}$ to transfer the tail flit to the first hop. After that, it takes another $\frac{F}{W}(D-1)$ to transfer the tail flit to the final destination (there are $D-1$ hops left). Consequently, this results in $T_{wormhole} = \frac{L}{W} + \frac{F}{W}(D-1)$.

A potential drawback of wormhole routing is that it may cause *tree saturation*. When the head of a packet is blocked, the tail remains distributed over multiple switches in the network (the flits are locally stored in the small flitbuffers at the intermediate switches) while keeping the allocated channels blocked. This means that no other transmission can make use of the allocated channels of the blocked packet, which may again cause new packets to block. This may lead to a snowball effect, resulting in a 'tree' of blocked packets. This is illustrated in Figure 2.17. Packet A is blocked and keeps its allocated channels blocked as well. Packet B cannot continue and is also blocked. A solution to this problem is the application of so-called virtual communication channels which will be described later in this chapter.

Another solution to the tree saturation problem is by combining wormhole routing and store & forward switching. This is called *virtual cut-through switching*. In virtual cut-through switching, the buffers at the switches have a size of at least an entire packet (like in store & forward switching). The transmission of packets is, however, performed in a pipelined fashion identical to wormhole routing. Because there are packet-sized buffers at switches, this means that when a header flit is blocked, the tail of the packet continues until the entire message is buffered at the switch of the blocking header flit. So, because of the extra buffer space, the tail of a blocked packet can be removed from the network, while in wormhole routing the tail remained in the network keeping the intermediate channels allocated. It is not hard to see that this characteristic of virtual cut-through switching reduces the amount of tree saturation.

In all packet switching methods, some form of *flow control* is required. Flow control is the mechanism taking care that there are no buffer overruns and that transmitted data is not lost. Figure 2.18 depicts an

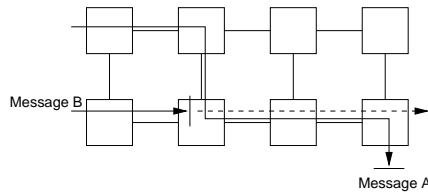


Figure 2.17.

example of such a flow control mechanism for wormhole routing. It uses a handshaking protocol between neighboring switches to control flit transmissions. The Request/Acknowledge line specifies whether or not a switch has enough buffer space to store an incoming flit. By pulling the Request/Acknowledge line down, switch D signals it can handle a new incoming flit. Then, when a flit is transmitted to switch D, the Request/Acknowledge line is raised to indicate that no new flits may be transmitted until switch D has lowered the Request/Acknowledge line again. The latter is done when switch D is able to forward its previously received flit.

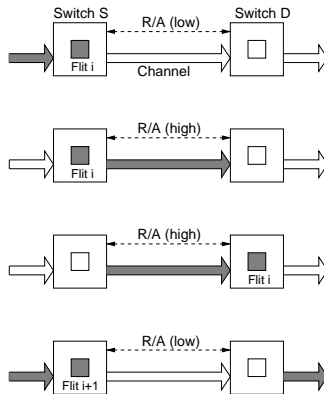


Figure 2.18.

In Figure 2.19, a generic switch architecture is shown. It contains a number of input ports and output ports. Internally, a switch may have only input or output buffers or it may feature both (like the switch in Figure 2.19). There is a crossbar to connect the input ports with the output ports (remember: this allows for non-blocking communication for all input-output permutations). Finally, there is control logic for routing incoming packets (which includes controlling the crossbar) and for scheduling of events (e.g., the order in which incoming packets are routed to the output ports).

2.5 Routing techniques

Having described techniques for packet switching (how are packets transmitted from source to destination), this section deals with packet routing. The routing algorithm determines the path the packets follow through the network to reach their destination.

The routing ‘intelligence’ can either be located at the source node (*source-based routing*) or can be present locally at each switch in the network. In source-based routing, used by for example Myrinet networks, the source node determines the path a packet should follow through the network. This path is explicitly stored

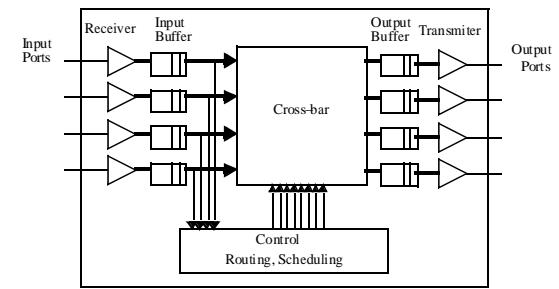


Figure 2.19.

in the header of the packet defining per hop in which direction the packet should travel. For example, such a header could look like: “go east, go east, go north, go east, ...”. A switch in the network routes an incoming packet according to direction information at the front of the packet header and subsequently removes the direction information. The intermediate switches thus ‘eat’ the packet header until no direction information is left in the packet. If everything is correct, the packet should then have reached its destination. Source-based routing is relatively easy to implement but has several drawbacks. First, packets need to include the entire path they should follow which may increase the packet size considerably. Second, packets cannot anticipate on hardware failures (e.g., broken communication channels) or occupied communication channels by choosing an alternative route.

In the case the routing intelligence is placed locally at the switches, a source node only requires to specify the destination node number in the header of the packet (yielding much smaller packets). The intermediate switches autonomously determine how an incoming packet (with the specified destination) needs to be routed. Clearly, this requires more complex routers than in source-based routing.

A potential danger in network routing is that it may cause deadlocks. This happens when there is a cyclic dependency between buffers (store & forward switching) or channels (wormhole routing). In Figure 2.20 such a situation is illustrated for wormhole routing in a mesh network. Channel C1 is allocated by a packet which wants to use channel C2. Channel C2 is allocated by another packet which wants to use C3. Yet another packet which allocated C3 wants to use C4, and finally, a fourth packet on C4 wants to use C1. Figure 2.20 also shows the channel dependence graph for the situation, which in this case contains a cycle. This implies a deadlock situation. Later on in this section, we will present different approaches for avoiding such deadlocks.

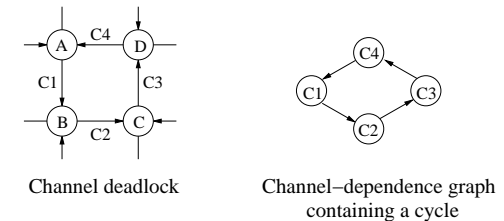


Figure 2.20.

Another characteristic of routing is that it may be *minimal* or *non-minimal*. In minimal routing, the routing algorithm always chooses a shortest path between two communicating nodes, while in non-minimal routing any path can be chosen. As a result, non-minimal routing can choose from a wider variety of paths but it may also suffer from starvation: packets are not guaranteed to reach their destination.

2.5.1 Local routing techniques

Let us now consider some routing techniques when the routing is performed locally at the switches in the network. Two main classes of such routing can be recognized: *deterministic routing* and *adaptive routing*. In deterministic routing, packets from a single source to the same destination always follow the same route. This route is often chosen such that it is minimal and deadlock-free. A well-known deterministic routing method is based on dimension ordering, such as X-Y routing which illustrated in Figure 2.21. A packet being sent from node S to node D is always first sent to the correct X coordinate and then to the correct Y coordinate. This explains the name X-Y routing. In X-Y routing, routing deadlocks cannot occur³ since the cyclic dependency shown in the right picture of Figure 2.21 cannot occur. This is because a packet that comes from the south/north (an Y direction) cannot be routed to the west/east (an X direction) anymore. Dimension ordering-based routing is a popular routing method because its simplicity.

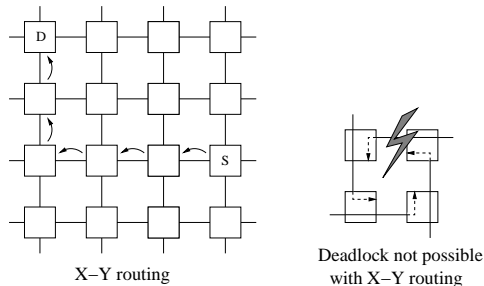


Figure 2.21.

Another example of deterministic routing is *interval labeling*, of which an example is shown in Figure 2.22. In interval labeling, output ports of switches have an interval associated with them. An incoming packet at a switch is sent to the output port with the 'matching' interval. An interval matches when the destination of the incoming packet is part of the interval. Studies have shown that the intervals can be chosen such that no routing deadlocks can occur. The Inmos C104 switch is an example of an actual implementation of interval labeling.

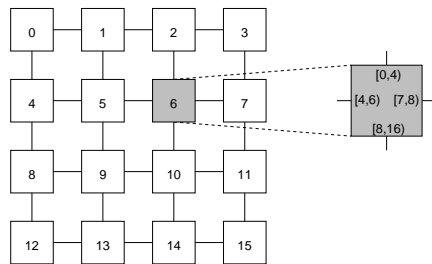


Figure 2.22.

In adaptive routing, the existence of alternative paths is exploited. Using environmental information (e.g., communication channels being busy/available, channel failures, etc.), a switch can adaptively make routing decisions. In Figure 2.23, an example of adaptive routing is shown. In this example, a packet is sent from node S to node D, while there are a number of channel blockages in the network (e.g., the channels

³Here, we assume that the network *does not* have wrap-arounds, like a torus does. In the discussion on the Cray T3D machine in the next chapter, we will describe what is needed when dimension-ordering routing is combined with a torus network.

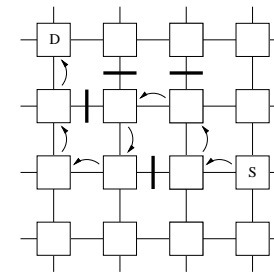


Figure 2.23.

are used for other transmissions). As can be seen, adaptive routing allows for circumventing the channel blockages and finding an unblocked route to the destination. This means that adaptive routing is less prone to network contention and is more fault-tolerant than deterministic routing. To further demonstrate that adaptive routing is less prone to contention than deterministic routing, consider Figure 2.24. The figure shows for both deterministic X-Y routing and adaptive routing what may happen when the nodes at the bottom row of a mesh send a message to the top-right node. In X-Y routing, such a communication will cause heavy contention along the right-most column of nodes. In adaptive routing, however, the message transfers can follow different routes in order to obtain a much more uniform network load.

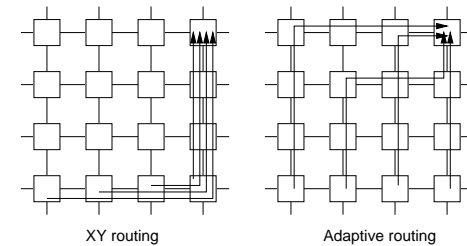


Figure 2.24.

Basically, there are three drawbacks of adaptive routing. First, switches need to be more intelligent and thus more complicated (and expensive) than in deterministic routing. Second, guaranteeing deadlock freedom in adaptive routing significantly complicates the routing algorithm. Third, packets from the same message may arrive out of order at the destination because they followed different routes. So, packets may have overtaken each other. This means that the message needs to be reassembled at the destination after all packets have been received by giving the packets sequence numbers. Needless to say, such reassembling adds communication latency.

Adaptive routing can either be minimal or non-minimal. Moreover, another distinction can be made between fully adaptive and partially adaptive routing schemes. In fully adaptive routing, there are no constraints on the directions a packet may be routed to, whereas partially adaptive routing is more restrictive. We will come back to partially adaptive routing in the next subsection.

Deadlock avoidance

To avoid routing deadlocks, one can follow several approaches. First, and the simplest method, is to choose for deterministic routing. When there are no wrap-arounds in the network, this immediately solves the deadlock problem. Another approach is to use a partially adaptive routing algorithm. On the left side of Figure 2.25, the turns that packets are allowed to make in X-Y routing are shown. The turns indicated by the

dashed arrows are not allowed in X-Y routing to guarantee that no cycles can occur. But you can immediately see that it is not needed to disallow two turns in a clockwise (or anti-clockwise) cycle. Disallowing only one turn will suffice. This is illustrated on the right part of Figure 2.25 in which only north-west and south-west turns are disallowed to avoid cycles. This yields so-called *west-first routing*: first route a packet to the west (if needed), then route the packet adaptively to the north, south or east. Because of the latter stage in which a packet can be adaptively routed in three directions, this routing method is called partially adaptive.

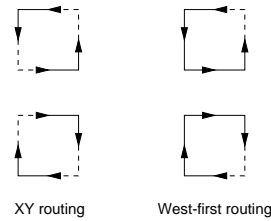


Figure 2.25.

In Figure 2.26, three examples of west-first routing are shown in a mesh network. As can be seen, west-first routing is non-minimal as it is able to circumvent channel blockages. In this example, only one destination node lies west of the source node. So for this case, the message is first transmitted to the west (due to channel blockages it continues traveling to the west) after which it is adaptively routed to the north and east. For the two other transmissions, the destinations are to the east of the sources which implies that the messages can immediately be routed adaptively to the north/south/east.

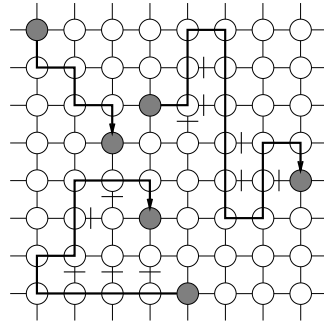


Figure 2.26.

Another method to avoid deadlocks is by introducing *virtual communication channels*. A virtual channel is a logical link between two nodes/switches using its own physical buffers. Multiple of these virtual channels are (time-)multiplexed over a single physical channel. The concept of a virtual link is illustrated in Figure 2.27. Using some scheduling policy (e.g., round robin), a virtual channel buffer is selected from which data (e.g., a flit) is transmitted along the physical link. After the transmission, a next virtual channel buffer is selected for transmission.

An example of how virtual channels can help avoiding deadlocks is shown in Figure 2.28, which refers to the deadlock situation in Figure 2.20. In the new network, there are two virtual channels between nodes A and D as well as between nodes C and D. By carefully selecting the output channel (dependent on where a packet came from), one can now avoid deadlock. This is demonstrated by the channel-dependence graph on the right in Figure 2.28: the cycle has been broken.

Another example of the application of virtual channels is shown in Figure 2.29. This figure depicts a

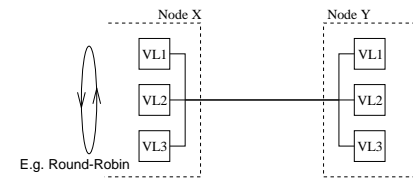


Figure 2.27.

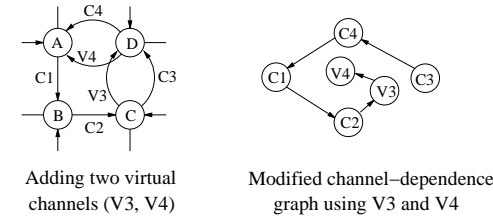


Figure 2.28.

mesh network with two virtual bi-directional links in the north-south direction and a single bi-directional link in the east-west direction. In the case a packet needs to be transmitted to a destination which lies to the east of the source (so it must be routed in the +X direction), the +X subnetwork is used. This +X subnetwork consist of one the two virtual channels in the north-south direction and the links to the east. Clearly, by only using the links in the +X subnetwork, deadlock cannot occur since it is impossible to form cycles in this network. Naturally, the same holds for the -X subnetwork in the case the destination lies to the west of the source.

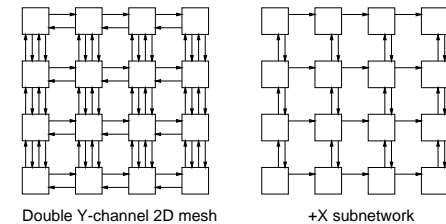


Figure 2.29.

Besides the fact that virtual channels can avoid routing deadlock, they have a few additional advantages. By applying virtual channels, the network throughput is increased. If a packet on a virtual channel is blocked (waiting for another channel to become free), other packets may still be transmitted along the physical link via different virtual channels. Another advantage of virtual channels is that they facilitate the mapping of a logical topology of communicating processes onto the physical topology of the network. Finally, a subset of the virtual channels can be dedicated to guarantee communication bandwidth to certain system-related functions such as debugging and monitoring. In that case, parallel applications are not affected by debug-ging/monitoring network traffic and vice versa.

Naturally, there also are disadvantages of virtual channels. They require more complex hardware (e.g., more buffer space, scheduling logic, etc.) thereby increasing expenses. Moreover, while virtual channels increase the network throughput, they may also increase the communication latency. This is because the use of virtual channels implies that packets need to share (the bandwidth of) communication channels with other

packets. Finally, there is a remote chance that packets (from the same message) following the same route but along different virtual channels overtake each other. In that case, it is required to reassemble the packets at the destination (which again adds latency).

2.6 Complex communication support

In the last section of this chapter on networking, we will shortly mention two architectural techniques for providing complex communication support to parallel applications. These techniques exploit the fact that parallel applications often contain common communication patterns. Examples are:

- Partner communication (unicast)
- Multicast (one-to-many)
- Broadcast (one-to-all)
- Exchange (many-to-many or all-to-all)

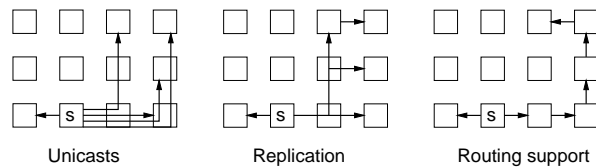


Figure 2.30.

The network switches in the network may feature hardware support for these communication patterns. To illustrate this, consider Figure 2.30 in which a multicast operation is implemented in three different ways. The multicast can be implemented without hardware support, using unicast communications like a software environment such as MPI does. In that case, the unicasts are often sent according to a spanning tree structure to minimize the number of unicasts and to reduce the multicast latency. Another possibility is offered when the switches allow for data replication. In that case a multicast message contains all destinations in its header and the switches in the network may copy incoming messages. After copying, a switch can send the (copied) messages in multiple directions along different output channels (based on the destinations mentioned in the header). The third way to implement the multicast is to provide routing support so that a single message can reach multiple destinations. For this method, again multiple destinations need to be specified in the message header.

there are two implementation schemes: *Virtual Shared Memory (VSM)* and *Shared Virtual Memory (SVM)*.

3.1 VSM and SVM

In both the VSM and SVM schemes, the application programmer sees a machine which features one big shared memory that is globally addressable. The memory references made by applications are, if required, translated into the message-passing paradigm. So, if a processor accesses a memory location which is not located in its own local memory, the VSM/SVM schemes translate the memory access into a message transfer to access a remote memory. Now, what is the difference between VSM and SVM?

Virtual Shared Memory (VSM) is a hardware implementation. This means that the virtual memory system of the OS is transparently implemented on top of VSM. Or, in other words, the OS also thinks it is running on a machine with a shared memory. Because VSM is a hardware implementation, it usually applies a small 'unit of sharing': typically a cache block. Assuming a cache block as the unit of sharing, then the cache controller determines whether or not a memory reference is located in the local memory. If not, then the controller will send a message to the remote node owning the memory location in order to retrieve the cache block in question.

Shared Virtual Memory (SVM) is a software implementation at the level of the OS with hardware support from the Memory Management Unit (MMU) of the processor. So, this implementation is not transparent to the OS. The unit of sharing are OS memory pages. If a processor addresses a certain memory location, the MMU (like in a normal processor) determines whether the memory page associated with the memory access is in (local) memory or not. In a normal computer system, when a page is not in memory, it is swapped in from disk by the OS. However, in SVM, the OS fetches the page from the remote node which owns that particular page.

Chapter 3 Multicomputers

Having discussed the processor and network building blocks, we can now start talking about actual parallel computer architectures which apply these building blocks. In this chapter, we will focus on so-called *multicomputers*, which are distributed memory MIMD architectures. The conceptual model of a multicomputer is shown in Figure 3.1.

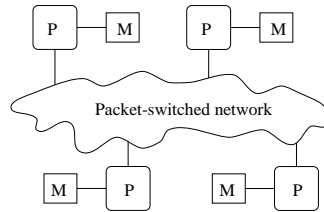


Figure 3.1.

Typically, multicomputers are message-passing machines which use packet switching to exchange data. They often use a scalable point-to-point (direct) network to connect the processors with each other. Each processor in a multicomputer has a private memory. There is no global address space since a processor can only access its own local memory. This means that communication and synchronization between processors is performed using message passing. As a consequence, communication is not transparent: programmers need to explicitly put communication primitives in their code.

Sometimes, multicomputers are also called *shared nothing architectures*. This is because all system resources (like memories, disks, and so on) are distributed and only privately accessible by the local processor. So, there are no shared resources.

Multicomputer architectures may scale to large numbers of processors. Good examples are the ASCI Red (9632 processors) and the ASCI White (8192 processors) machines. Early 2002, the latter one was ranked as the world's fastest computer.

As was explained in the previous chapter, intermediate nodes in a direct network must route messages when two communicating nodes are not neighbors. In early multicomputers, this routing was performed in software by a processor itself. It received an interrupt at an incoming packet, after which the interrupt handler took care of the routing. Clearly, interrupting the processor each time a packet comes in is not very efficient. In modern multicomputers, there usually is a communication chip integrated with each processing node which takes care of the switching, routing and DMA transfers to the local memory (when a packet reached its final destination).

An inherent drawback of multicomputers is that there is no global accessible memory. This complicates the (transparent) sharing of data and code. For example, to execute an SPMD program on a multicomputer, the program first needs to be transferred (broadcasted) over the network to all the nodes. A possible solution to this problem is by 'mimicking' a shared memory on top of a distributed memory machine. To this end,

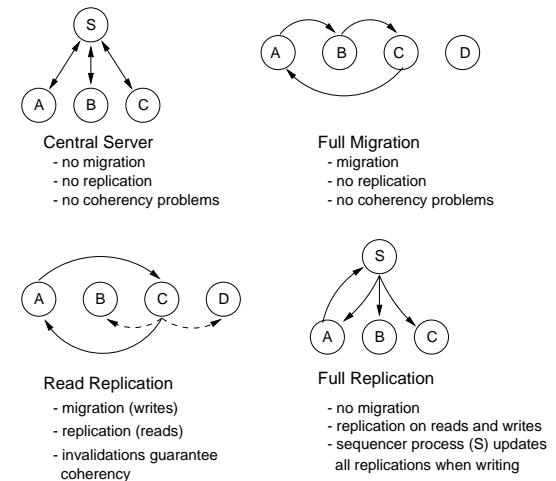


Figure 3.2.

There are four approaches to VSM and SVM, as illustrated in Figure 3.2. In the *central server* approach, a single reader or a single writer can access a memory block (= unit of sharing) at a time by having all requests go through a central server. The server is the only entity in the network that knows the actual location in distributed memory of the requested block and sequences the requests. Here is a scenario how this algorithm works:

1. Node A sends a request to S for memory on node B.

2. Node S receives the request from node A.
3. Node S forwards the request to node B.
4. Node B replies to node S.
5. Node S forwards the reply to node A.

The advantage of this method is that a strong coherency and consistency is maintained. Although the central-server algorithm is simple to implement and ensures a high level of memory coherence, it limits parallelism (single reader/writer of a data block) and the central server can become a bottleneck. To overcome the latter problem, shared data can be distributed among several servers. In such a case, clients must be able to locate the appropriate server for every data access.

Another single reader, single writer method is the *full migration* approach. Here, there is no central server, but instead, all nodes know (based on the static distribution of addresses) where in the network the original owner processor of a memory block is located. One processor at a time may own a block of data, and when a request is made for that data, it must transmit it to the requester and invalidate its copy. Thus, only one copy of a block exists at a time in the entire network, and that copy may be read or written with impunity. Like already mentioned, in this algorithm, each piece of data has an associated default owner — a node with the primary copy of the data. The owners change as the data migrates through the system. When another node needs a copy of the data, it sends a request to the original owner. If the owner still has the data, then it returns the data. But if the owner has given the data to some other node, then it forwards the request to the new owner. A drawback of this scheme is that a request may be forwarded many times before reaching the current owner, which is in some case more wasteful than broadcasting. And, again, this approach limits parallelism (single reader/writer of a data element).

Read replication is a multiple reader, single writer approach that attempts to minimize latency by allowing more than one processor to have read-only copies of memory blocks. Again, all the nodes know, based on static address distribution, where a particular memory block resides in distributed memory. When a processor performs a non-local read, it simply gets a copy of the memory block from the owner processor. In the case of a write, the following is done:

1. Node A determines that the address it requires is non-local, and sends a write request for memory on node C.
2. Node C receives the request from node A, but determines that it has lent copies of the block to nodes B and D.
3. Node C sends invalidate messages to B and D, which will force further reads to miss.
4. Node C migrates the requested block to node A.
5. Node A receives the reply from C.
6. Node A can now read and write the block.

Finally, *full replication* is a multiple reader, multiple writer protocol that attempts to minimize latency by allowing more than one processor to have read/write copies of memory blocks. Again, all the nodes know, based on static address distribution, where a particular memory block resides in distributed memory. Read misses are handled by sending a copy of the requested block from any node that has it. When a processor wants to write to a copy of a block, a write request is sent to a special sequencer process, which assigns a sequence number to the request, and broadcasts the write to all processors with a shared copy of the block. Each processor then independently updates its copy.

We will return to VSM and, in particular, to the VSM read replication scheme in the next chapter.

3.2 Real multicomputers

In the last section of this chapter, we will briefly discuss two actual multicomputer architectures.

3.2.1 The IBM SP2

The first machine is the IBM SP2, in which the processing elements are POWER2 processors. They are connected using switch boards as shown on the left in Figure 3.3. Such a switch board contains a two-level multistage network consisting of eight 8×8 switches (note, in Figure 3.3 the channels are bidirectional). A 32-processor system can be constructed from a single switch board. To construct an 80-processor system, five switch boards are connected with each other like is shown on the right in Figure 3.3. A 128-processor system can be built by connecting eight switch boards, each accommodating 16 processors, in a multistage network using an extra row of 'bended' switch boards (i.e., switch boards of which all 32 ports are used for switching and not for connecting a processor). This is shown on the lefthand side of Figure 3.4.

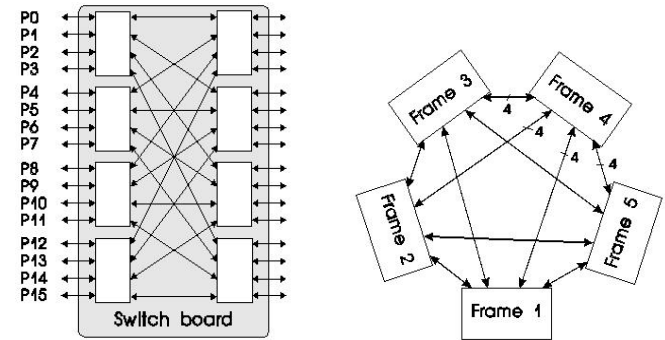


Figure 3.3.

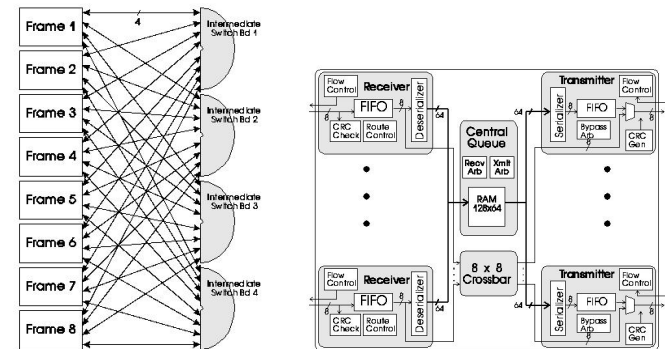


Figure 3.4.

On the right of Figure 3.4, the architecture of a switch within a switch board is shown. It contains eight receiver and transmitter elements which implement a variant of wormhole routing. A receiver element directly routes an incoming flit, via an 8×8 crossbar, to the appropriate transmitter when this transmitter is available (i.e., its output channel is not occupied by another packet). If a transmitter is not available, then the receiver

packs the incoming flits together in bundles of 64 bits (= 8 flits) and stores these bundles in a central 128-entry buffer. The bundles stay in this central buffer until the required transmitter becomes available again. When the central buffer is full, then the flow control will stop the stream of incoming flits.

3.2.2 The Parsytec CC

Increasing the efficiency of communication networks by changing their topology often leads to a decrease of realizability due to the high cost. Particularly, networks with multistage topologies can offer a small diameter, large bisection width and a large number of redundant paths but may be hard and expensive to construct because of the complex wiring structure. By combining a multistage network with an easy to realize mesh network, the Parsytec CC multicomputer addressed (it is not produced anymore) this tradeoff between efficiency and realizability. This machine consisted of PowerPC 604e processors connected in a Mesh of Clos network which is, as its name already suggests, based on the Clos multistage network. A Clos network of height h , constructed of switches with $2k$ bidirectional communication channels, is defined by the following recursive scheme:

- A single switch with $2k$ bidirectional communication channels of which k are connected to nodes is a Clos network of height 1.
- A Clos network of height h is built by connecting k Clos networks of height $h - 1$ by k^{h-1} switches. Since each of the k subnetworks has k^{h-1} external channels, k^{h-1} switches are used at level h such that the i -th external channel of each subnetwork connects to the i -th switch at level h .

In the remainder of this section, we assume that the routers are configured with eight communication channels of which at most four can be connected to nodes, i.e. $k = 4$. Figure 3.5 shows a Clos network of height 2 in which four channels of each switch at the top level are left unused. Subsequently, the Mesh of Clos(h,r) topology is defined by replacing the r top stages of a Clos network of height h by a $2^r \times 2^r$ mesh structure. Two examples of a Mesh of Clos network are depicted in Figure 3.6: a Mesh of Clos(2,1) and a Mesh of Clos(3,1). Both networks contain a 2×2 mesh, the first having clusters of four nodes and the other having clusters of sixteen nodes. The latter is actually called a *Fat Mesh of Clos* since the mesh structure that interconnects the clusters can be regarded as four independent layers of 2D meshes (as illustrated by the different shaded surfaces in Figure 3.6). Note that there are several optimizations possible for the networks shown in Figure 3.6 by connecting unused channels.

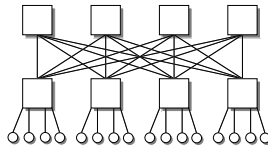


Figure 3.5.

Table 3.1 gives a comparison of some network characteristics of the Mesh of Clos and mesh topologies. From this table can be seen that, for small values of r , the Mesh of Clos has better network characteristics than a mesh.

	Mesh of Clos(h,r)	Mesh
Router degree	8	4
Diameter	$2 \cdot (2^r - r - 1) + \log_2 N$	$2 \cdot (\sqrt{N} - 1)$
Bisection width	$N \cdot 2^{-r-2}$ (with $r > 0$)	\sqrt{N}

Table 3.1.

Another important issue is, of course, the number of switches (and channels) the network requires to be built. In Figure 3.7, the number of switches needed to build a Mesh of Clos (dark surface) or a mesh (light

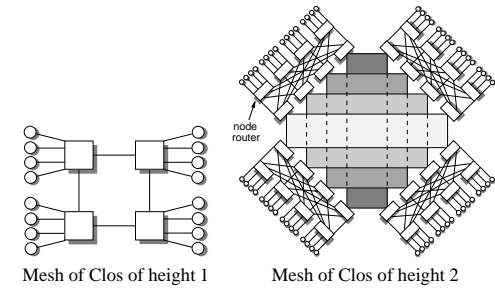


Figure 3.6.

surface) is shown in a graph. The axis labeled with r defines the number of top stages that are removed from the Clos network to form the Mesh of Clos. Note that the surface within the triangle between $r = 2$ and 1024 nodes is not defined because r would then be equal to or larger than the height of the Clos network. Or, in other words, all Clos levels or more levels than there are available would be removed. Figure 3.7 illustrates that the number of switches, which is rapidly increasing for a pure Clos ($r = 0$), can be reduced considerably by replacing the top stages of the Clos for a mesh structure, i.e. by increasing r . However, as demonstrated by Table 3.1, r should not become too large in order to preserve a small network diameter and a large bisection width. More specifically, if $r \rightarrow h$ then the MoC's network diameter becomes similar to that of the mesh network while the bisection width may eventually become worse than that of the mesh.

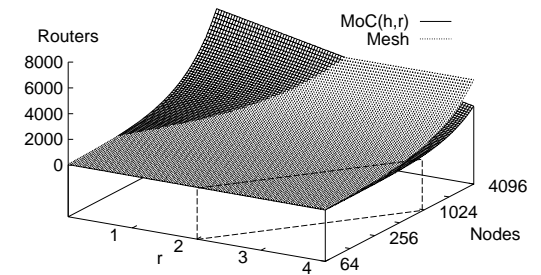


Figure 3.7.

Chapter 4

Multiprocessors

Parallel computers that have a single globally addressable memory are called *multiprocessors*. The processors in these machines are typically connected using indirect networks or, as we will see later on, a combination of an indirect network and a point-to-point network. The communication and synchronization between processors is established by writing to and reading from shared locations in the global memory. For synchronization, this means that locks should be used to protect accesses to critical regions (to maintain mutual exclusion). As we will describe later on, we need hardware support to construct such locks. Another characteristic of the shared-memory paradigm is that message passing can be emulated. So, environments such as MPI are relatively easy to port to shared-memory machines.

Traditional multiprocessors do often not scale well to a large number of processors since they typically use indirect networks, such as a bus. For example, a bus-based multicomputer may already run into saturation problems beyond 8 processors. Later in this chapter, we will describe systems such as so-called NUMA (Non-Uniform Memory Access) machines that try to alleviate this problem.

Another, and probably the biggest, problem that computer architects need to address in the design of multiprocessors (with caches) is the cache coherency problem. In the next section, we will elaborate on this problem.

4.1 Cache coherency in shared memory machines

Maintaining cache coherency is a problem in multiprocessor systems where the processors contain local caches. In these system data inconsistencies between different caches in the system can easily occur. Three main sources of the problem can be identified:

- Sharing of writable data
- Process migration
- I/O activity

We will illustrate each of these three problem areas with an example. All examples consist of three pictures of a simple two-processor multicomputer system of which the leftmost picture always refers to the initial state of the system, the one in the middle to a situation in which we assume a system with write-through caches and the one on the right to a situation in which we assume a system with write-back caches.

The first example, dealing with the sharing of writable data, is shown in Figure 4.1. On the left hand side of this figure, the initial state of the system is shown: the processors P1 and P2 both have data element X in their local caches. In the scenario depicted in the middle, processor P1 writes to data element X. Because the caches are write-through, both P1's local cache is updated as well as the main memory. However, the copy of data element X in the cache of processor P2 is outdated now. So, when processor P2 tries to read X, it receives bogus data. In the scenario depicted on the right, P1 also writes X but now for write-back caches. Again, P2 receives bogus data when reading X. This may even be true when X would not have been cached in P2's cache since data element X in the main memory may also be outdated (write-back caches).

In Figure 4.2, cache coherency problems related to process migration are illustrated. Again, the leftmost picture shows the initial status: P1 caches data element X, whereas P2 does not. In the scenario depicted

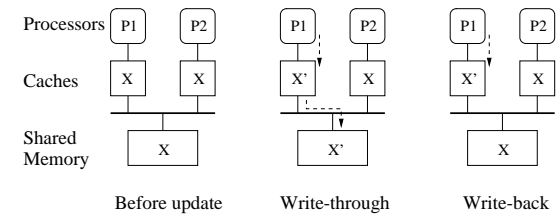


Figure 4.1.

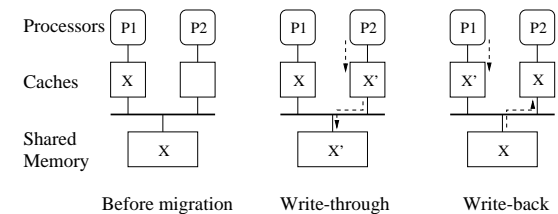


Figure 4.2.

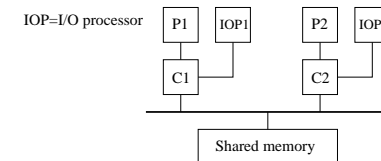
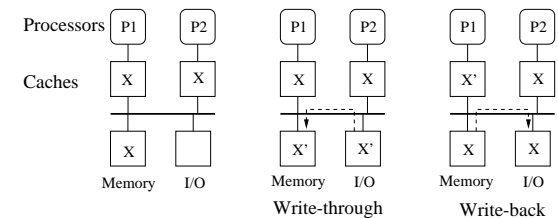


Figure 4.3.

in the middle, a process on P2 first writes to X and then migrates to P1. After migration, the process starts reading X. But there is still an outdated copy of X located in the cache of P1, which again results in the reading of bogus data. In the scenario shown on the right, a process on P1 first writes to element X and then migrates to P2. After migration, the process on P2 starts reading X. Subsequently, it will fetch an outdated version of X from the main memory.

At the top of Figure 4.3, the cache coherency problems related to I/O are illustrated. In the two-processor multiprocessor architecture, an I/O device has been added to the bus. Initially, both caches contain data element X. In the middle scenario, the I/O device receives a new element X and stores this element directly in the main memory (we call the I/O device DMA-enabled, where DMA stands for Direct Memory Access). When either P1 or P2 reads element X, it gets an outdated copy. In the scenario on the right, P1 writes to element X, after which the I/O device wants to transmit X. To this end, the I/O device reads X from the main memory. Again, an outdated copy is read because of the write-back caches.

A possible solution to the I/O problem is shown at the bottom of Figure 4.3. For this solution it is assumed that the caches can be kept consistent for the two other sources of coherency problems (writable shared data and migration). By moving the I/O device next to a processor, implying that it always needs to access the bus through the processor's cache, the cache always contains the most recent version of a data element which has been accessed by the I/O device. Another and widely-used solution to the I/O coherency problem is by not caching the address ranges used for I/O at all.

The mechanism to maintain the coherency of the caches in a multiprocessor system is called a *cache coherency protocol*. Cache coherency protocols are, as we will see later on in this chapter, based on a set of cache block *states* and *state transitions*. There are two main types of cache coherency protocols: *write-invalidate* and *write-update*. These protocols are illustrated in Figure 4.4.

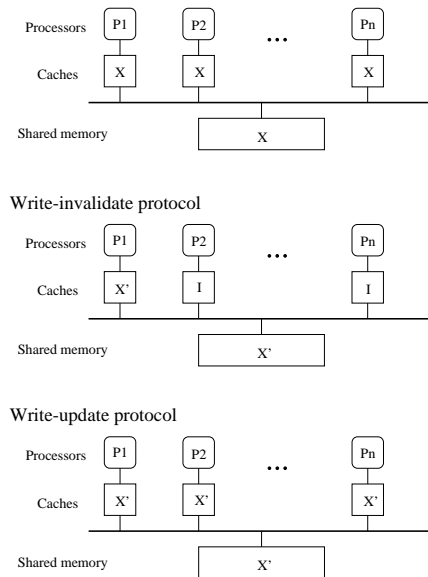


Figure 4.4.

At the top of Figure 4.4, the initial situation is sketched. At a certain moment in time, processor P1 writes to data element X. In a write-invalidate protocol (shown in the middle of Figure 4.4), the copies of X residing in other caches are invalidated (they receive the state 'invalid'). So, this clearly is a read-replication protocol as discussed in Section 3.1. When writing to X in a write-update protocol (shown at the bottom of Figure 4.4),

all of its copies in the system are updated with the new value. Hence, this is a full-replication protocol (see Section 3.1). Clearly, a write-update protocol requires more bandwidth than a write-invalidate protocol as it needs to broadcast new data values.

The write-invalidate protocol suffers from a phenomena called *false sharing*. In false sharing, invalidations of data elements are performed which are not necessary for correct program execution. Let's illustrate this using the following code executed by a 2-processor multiprocessor system.

```
Processor 1:      Processor 2:
while (true) do   while (true) do
A = A + 1         B = B + 1
```

If A and B are located in the same cache block, then writing to A will invalidate B in the cache of processor 2 and vice versa. Subsequently, a cache miss for A and B will occur in each loop-iteration due to a ping-pong of invalidations.

Later in this chapter, we will return to the topic of cache coherency and look at cache coherency protocols in more detail.

4.2 Uniform Memory Access (UMA)

Traditional multiprocessor systems use an indirect network and have an *Uniform Memory Access (UMA)* architecture. The latter means that (the latency of) accessing the shared memory is the same for all processors in the system. Two bus-based UMA multiprocessor systems are depicted in Figure 4.5, one with caches and one without caches.

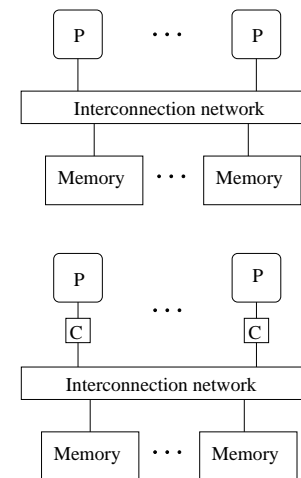


Figure 4.5.

A popular class of UMA machines, which are commonly used for (file-)servers, are the so-called *Symmetric MultiProcessors (SMPs)*. In an SMP, all system resources (memory, disks, other I/O devices, etc.) are accessible by the processors in a uniform manner, i.e., the processors have a uniform view of the entire system.

The traditional UMA multiprocessors as described above suffer from several problems. The first problem is that they are hardly scalable. Bus-based UMA architectures suffer from bus saturation when scaling to a

large number of processors. Connecting the processors with a crossbar also limits the scaling since it is too expensive (wiring constraints). Scaling a multiprocessor with a multistage network leads to wiring constraints and a possible higher memory latency since the number of stages needs to be increased.

Possible solutions to the poor scalability of these multiprocessor systems are:

- Reduce bus traffic by means of caching. This will only work for scaling up to a certain number of processors, after which the cache misses of the processors in the system again lead to bus saturation.
- Cluster a small number of processors around a memory via an indirect network (e.g., an SMP-cluster) and connect multiple clusters via a scalable point-to-point network. This leads to so-called Non-Uniform Memory Access (NUMA) machines. NUMA architectures, and related systems, will be discussed later on in this chapter.

A second problem of UMA architectures is that memory contention occurs when multiple processors want to access the memory at the same moment. This problem might be solved by splitting the memory into multiple memory banks which are accessed based on address ranges. For example, in a dual-banked memory, the even addresses could go to the one bank while the odd addresses go to the other one. In UMA architectures with a multistage network, tree saturation may occur when multiple processors want to access the same memory module/bank. A potential solution to this is the use of *message combining*.

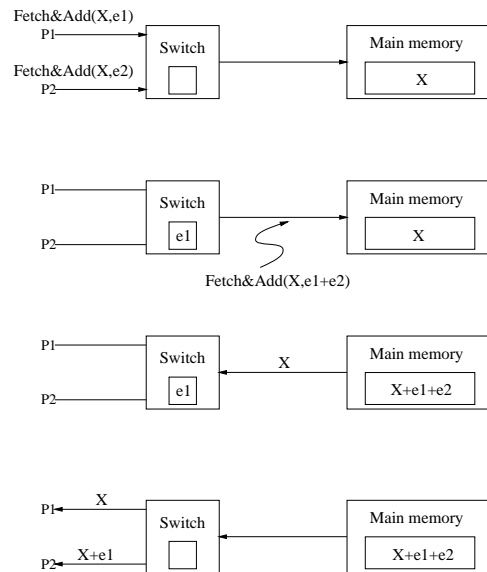


Figure 4.6.

In Figure 4.6, the use of message combining with the atomic Fetch&Add operation is illustrated. The figure shows four states of a switch in the last stage of a multistage network. Only one output of the switch is shown, which is connected to a memory module. At a certain moment in time, the switch receives two Fetch&Add operations (i.e., messages) for the memory module, as shown at the top of Figure 4.6. A Fetch&Add(X,e) operation is defined as follows: it retrieves data element X from memory and substitutes X with $X+e$ in the memory. This is all done as an atomic operation, i.e., it cannot be interrupted by another memory access.

When a switch receives multiple (two in the case of Figure 4.6) of these Fetch&Add operations at the same time, it may combine the operations. Assuming two Fetch&Add operations, this works as follows. First, the

switch stores the data value of one of the two Fetch&Add operations. Let's pick $e1$. It then sends a single combined Fetch&Add($X,e1+e2$) operation to the memory module. The result of this combined operation is that $X+e1+e2$ is stored in the memory while X is returned. The return value X is immediately forwarded to the switch where the original Fetch&Add($X,e1$) came from, while the switch where the Fetch&Add($X,e2$) came from receives $X+e1$. The latter is possible because the value $e1$ was stored at the switch.

4.3 Non Uniform Memory Access (NUMA)

As described earlier, UMA multiprocessor systems do not scale well. Therefore, Non Uniform Memory Access (NUMA) multiprocessor architectures are becoming increasingly popular. In a NUMA machine, there are multiple SMP clusters (having an internal indirect/shared network) which are connected in a scalable message-passing (often direct) network. This is illustrated in Figure 4.7.

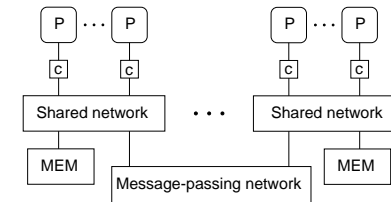


Figure 4.7.

The programmer of a NUMA machine still sees one global address space. So, a NUMA architecture is a logically shared, physically distributed memory architecture. This means that we are back at VSM (see Section 3.1)!

In a NUMA machine, the cache-controller or the MMU of a processor determines whether a memory reference is local to the SMP's memory or it is remote. In the latter case, a message is sent over the message-passing network to access the remote data. This implies that accesses to the local memory are fast, while remote accesses can be much slower (therefore the name NUMA). The ratio of local:remote memory access latencies may range from 1:[2-15].

To reduce the number of remote memory accesses, NUMA architectures usually apply caching—processors can cache the remote data. But when caches are involved, cache coherency needs to be maintained. That's why these systems are typically called CC-NUMA, or Cache Coherent NUMA. Most CC-NUMA machines use a write-invalidate (read replication) cache coherency protocol.

4.3.1 Latency hiding

While caches in a CC-NUMA architecture *reduce* the average latency of memory accesses, it is also important to provide methods for latency hiding. In the case a remote memory access needs to be made (at a cache miss), you would like to overlap this communication with valuable computation. Three important techniques for latency hiding are:

- *Prefetching of data*
Before remote data is actually required, try to predict that it is going to be needed and fetch it beforehand from the remote node.
- *Hardware-threading*
When the processor threatens to be stalled for a remote data access, schedule a new thread of control (lightweight process). So, while one thread performs a remote memory access, another thread takes over the processor. We will come back to this latency hiding technique in the discussion of the Cray MTA machine.
- *Relaxed memory consistency models*: how consistent should the view of the memory be?

Let us briefly explain the last issue: relaxed memory consistency models. In a non-relaxed *sequential consistency* (SC) memory model all memory accesses are atomic and strongly ordered. The first means that all processors notice changes to the memory in a consistent manner (e.g., all invalidations are received at a write action before another processor performs a memory access), while the second means that all memory accesses are made strictly in program order. The latter implies that, for example, all invalidations from a previous write action of a processor must have been acknowledged before the processor may perform a next memory reference.

The concept of SC is illustrated in Figure 4.8. You could think of it as connecting the processors with a switch to the memory: the switch determines which processor may access the memory. After an access is fully completed, the switch may switch to another processor to access the memory. A processor which performs a memory operation but which does not have access to the switch (the switch is set to another processor), must wait until the switch is set into the right position.

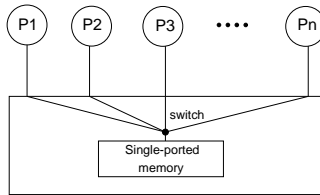


Figure 4.8.

With SC, the outcome of code fragment in Figure 4.9 is more or less predictable. If "Hi 1" is printed, then "Hi 2" will certainly not be printed, and vice versa (it is left to the reader to check this!).

```

Processor 1:           Processor 2:
A = 0;                 B = 0;
.....                 .....
A = 1;                 B = 1;
if (B == 0) printf("Hi 1\n");  if (A == 0) printf("Hi 2\n");

```

Figure 4.9.

Implementing SC yields a programming model which is transparent to the programmer, but it also results in sub-optimal performance because of the strictness of SC. Modern superscalar processors allow for aggressive reordering of instructions to hide memory latencies (e.g., by filling the pipelines with independent instructions after a load instruction) and feature write buffers to perform writes at the 'back-ground' while the processor continues with subsequent instructions. These latency hiding optimizations cannot be fully exploited with a SC memory model. Therefore, many multiprocessor systems also provide support for a relaxed consistency (RC) memory model.

There are a number of well-known RC models, such as

- Processor consistency (Sparc): loads may bypass writes within a processor
- Partial store order (Sparc): loads/writes may bypass writes within a processor
- Weak consistency (PowerPC) and Released consistency (Alpha,MIPS): no ordering between references

In all RC memory models, there are so-called *memory fences*. A memory fence is a synchronization operation (e.g., an atomic read-modify-write operation which will be discussed later on) which causes all previous memory operations to finish before the operation associated with the memory fence itself is performed. By placing the memory fences at the right places in the code, the RC memory model yields the same execution semantics as the SC model (but with higher performance because memory behavior is more relaxed in

between memory fences). To illustrate this, consider Figure 4.9 again. Assuming an RC memory model without placing memory fences, the code becomes unpredictable. If "Hi 1" is printed, then "Hi 2" could also be printed because the 'A = 1' may still not be finished: the write action may still reside in a write buffer of processor 1 waiting to be handled. But, by placing memory fences before the `if`-statements, the same execution semantics as the SC memory model is established again.

4.3.2 Disadvantages of CC-NUMA

The main disadvantage of CC-NUMA is that remote data can only be held in small local caches. When large amounts of remote data are needed (e.g., due to an incorrect data partitioning over the distributed memories) or remote data is needed infrequently, this means that subsequent accesses to remote data may often not be found in the local cache anymore, resulting in slow remote memory accesses. So, the relatively small caches in CC-NUMA machines may severely limit the performance.

One could of course increase the size of the caches. However, this is expensive and may also increase the latency of local memory references (larger caches tend to be slower than smaller ones). Another potential solution is to implement page migration/replication at the OS level. So, when certain remote data is frequently accessed, the OS may decide to replicate (read accesses) or migrate (write accesses) the memory page(s) in which the data is located. Such an OS-level implementation of page replication/migration is however quite complex and not very efficient (entire pages need to be transferred). Another drawback is that the migration/replication only works at page granularity. So, a problem might occur when there are parallel accesses that have a finer granularity: they may cause false sharing.

4.4 Cache Only Memory Architecture (COMA)

The disadvantages of CC-NUMA machines, as discussed in the previous section, lead to the design of another type of multiprocessor architecture: the Cache Only Memory Architecture (COMA). COMA machines are similar to NUMA machines, but their main memories act as direct-mapped or set-associative caches (see Figure 4.10). So, data blocks are hashed to a location in the DRAM cache according to their addresses. Data that is fetched remotely is actually stored (= 'cached') in the local main memory (\rightarrow replication). In addition, data blocks do not have a fixed home location: they can freely migrate through the system.

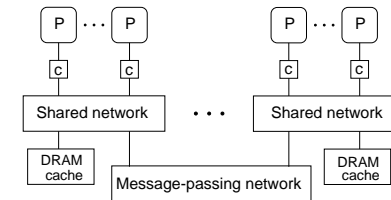


Figure 4.10.

COMA architectures often have a hierarchical (i.e., tree-like) message-passing network. A switch in such a tree contains a directory with data elements residing in its sub-tree. Since data has no home location (it is being cached in one of the main memory caches in the system), it must be explicitly searched for. This means that a remote access (i.e., there is a cache miss in the local main memory cache) requires a traversal along the switches in the tree to search their directories for the required data. This scheme allows for the message combining optimization: if a switch in the network receives multiple requests from its subtree for the same data, it can combine them into a single request which is sent to the parent of the switch. When the requested data returns, the switch sends multiple copies of it down its subtree.

For cache coherence, COMA architectures typically apply the write-invalidate coherency protocol. In that case, coherence is maintained at the granularity of main-memory cache blocks. Some characteristics of COMA make the coherency protocol more complicated to implement than the protocols used for CC-NUMA machines. For example, when a last copy of a data block resides in a main memory cache and an associated

processor decides to remove this copy (e.g., the replacement algorithm selected the cache entry in question to make room for new data), great care must be taken. Simply removing the copy from the main memory cache entirely removes a piece of memory from the system! In such a situation, the copy is usually moved to another main memory cache which is close by.

The big advantage of COMA systems are the main memory caches which are able to store large amounts of remote data (if needed). During the execution of an application, a local main memory cache gets automatically filled with the ‘working set data’ of an associated processor. The main memory caches require, however, extra memory-subsystem hardware. Besides the DRAM for the main memory cache itself, there should be an extra tag memory with which it can be checked whether or not a data block in the main memory cache is the required data block. In addition, when the main memory cache is set-associative, there should be extra comparators to perform the tag-check for multiple set-entries in parallel. Finally, there should be an extra state memory in which the state bits are stored for the main-memory cache blocks.

4.5 COMA versus CC-NUMA

Let us summarize the discussion on CC-NUMA and COMA architectures with a direct comparison between the two. COMA tends to be more flexible than CC-NUMA. The reason for this is that COMA transparently supports the migration and replication of data without the need of the OS and it does so at a fine granularity. The latter implies that the amount of false sharing is reduced. On the other hand, COMA machines are more expensive and complex to build. This is because they need non-standard memory management hardware (e.g., tag and state-bit memories) and the coherency protocol is harder to implement (e.g., take care that the last copy of a data element is not removed). Additionally, remote accesses in COMA are often slower than those in CC-NUMA since the tree network needs to be traversed to find the data.

Comparing the two architectures performance-wise is hard. It is highly dependent on the application which of the two architectures yields better performance. If the miss rates of the caches in the systems are small, then the performance of CC-NUMA and COMA will be similar. If capacity cache misses¹ dominate the performance, then COMA will outperform CC-NUMA because the main memory caches in COMA usually are large enough to store all required data (unlike CC-NUMA’s small processor caches). If coherence cache misses² dominate the performance, CC-NUMA will probably outperform COMA due to the higher latency of remote accesses in COMA.

4.6 Simple COMA (S-COMA)

To reduce the complexity and expenses of COMA systems due to the extra memory management hardware, a variant of COMA has been proposed: Simple COMA (S-COMA). In S-COMA, the allocation of a cache block in the main memory cache is done at the granularity of OS pages. So, like in SVM, the MMU determines whether or not the page of an address reference is in the local DRAM. In the case it is present, there is a cache hit, otherwise it’s a cache miss. Since the MMU takes care of determining whether or not a memory reference hits the main memory cache, there is no need for a tag memory and comparators.

Unlike SVM, however, S-COMA architectures manage the coherence of data blocks in hardware at a fine granularity, typically at the size of processor cache blocks (so not main memory cache blocks). Clearly, the fine grained coherency reduces false sharing. Another consequence of the fine grained coherency is that remote memory accesses transfer cache blocks rather than whole pages. This is shown in Figure 4.11. Let’s illustrate this scheme by explaining what happens on a cache miss. If a memory reference misses the local main memory cache (the associated page is not in cache), then first a new page is allocated in the main memory cache. Subsequently, the required data is fetched from a remote main memory cache and this is done at the granularity of a processor cache block.

Because remote data is fetched at the granularity of processor cache blocks, this means that a page can be partially filled with valid data. So, after a main memory cache hit (i.e., the page in question resides in the cache), it must also be checked whether or not the requested data is valid in the page. This is done using a state-bit memory. If the data is not valid (i.e., it has never been fetched or it has been invalidated), then it should still be fetched remotely.

¹A capacity miss occurs because the data does not fit in the cache.

²A coherence cache miss happens because of the invalidation of data.

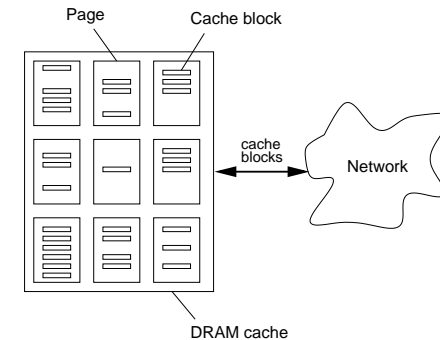


Figure 4.11.

In contrast to COMA, S-COMA is able to implement fully associative main memory caches. The OS in S-COMA may simply place the pages at any location in the main memory cache and specify this location in the page table, while in COMA it would mean that a lot of hardware comparators must be used (which is infeasible) to implement full associativity.

However, main memory cache misses in S-COMA are often slower than in COMA due to the OS support: the page fault indicating a cache miss is handled in software. In addition, there is a probability of *false replacement* in S-COMA: a page fault (main memory cache miss) allocating a new local page and fetching a single remote cache block may replace an entirely filled page from the main memory cache.

4.6.1 Reactive NUMA (R-NUMA)

Other types of multiprocessor architectures have been proposed that try to combine the strengths of CC-NUMA and (S-)COMA systems. An example is Reactive NUMA (R-NUMA) which attempts to combine CC-NUMA with S-COMA. We explained that in CC-NUMA remote items are stored in the (small) local processor cache. So, the capacity cache miss rate will be high when there is a lot of re-use of many remote data blocks. In S-COMA, the remote data blocks can be stored in large main memory caches (→ cheap re-use). However, if remote data blocks are not re-used very often, fetching and storing them may be expensive performance-wise. In R-NUMA, the architecture dynamically switches between CC-NUMA (with OS-level migration/replication) and S-COMA based on the application behavior.

4.7 Cache coherency revisited

Let us now have a closer look at the cache coherency protocols used in multiprocessor architectures. All cache coherency protocols are based on *cache block states* and *state transitions*. Because the state transitions of a cache block are driven by the actions of the local processor as well as those of other processors with a copy of the cache block, an important question in coherency protocols is: how to find copies of a cache block? With regard to this, cache coherency protocols can be divided into two classes: *snoopy bus protocols* and *directory based protocols*.

Snoopy bus protocols are used in bus-based multiprocessor architectures. They are mainly applied in UMA machines, but they can also be used in the SMP clusters of, for example, CC-NUMA machines. In a snoopy bus protocol, the caches of the processors monitor bus transactions by ‘snooping’ the bus to detect if another processor is accessing a cache block of which a copy is also present in the snooping cache. If it present, then appropriate actions should be taken by the snooping cache (e.g., its copy is invalidated). We will come back to this later on. Snoopy bus protocols can be write-invalidate or write-update.

In directory based protocols, the locations of copies of cache blocks are stored in a directory. Directory based protocols are typically used to maintain global (e.g., between the SMP clusters of a CC-NUMA ma-

chine) cache coherency of CC-NUMA and COMA architectures. As these architectures rely on non-broadcast message-passing networks, snoopy bus protocols (which require that all network traffic is snooped) are not suitable for this purpose. Directory based protocols usually are write-invalidate. A write-update protocol would be too expensive performance-wise for non-broadcast types of networks.

There are several methods for implementing a directory in directory based protocols. One could, for example, use a *full-map* implementation in which for each cache block in the system it is specified whether or not a copy is present in a cache of a certain processor. This implies that for each cache block a bitarray of P bits must be stored, where P equals to the number of processors in the system. This is illustrated in Figure 4.12. The picture at the top-left depicts the initial state of the directory entry of cache block X . Then, three processors read the cache block X which causes three bits to be set in the bitarray associated with X . Thereafter, processor P_n writes cache block X . This results in the invalidation of the copies in processors P_1 and P_2 , followed by the resetting of the associated bits in the bitarray.

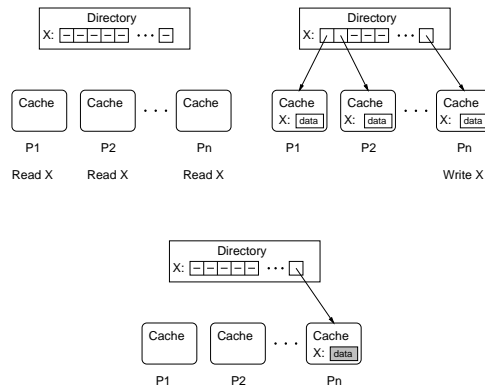


Figure 4.12.

A drawback of the full-map approach is the size of the directories: the number of bits in a directory scales with $O(P^2)$, where P is the number of processors in the system. This is because both the number of directory entries and the number of bits per entry scale with the number of processors.

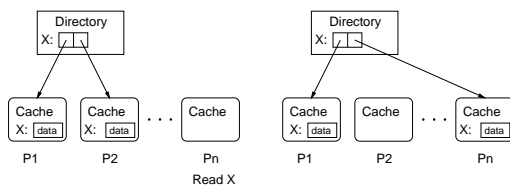


Figure 4.13.

A more restricted implementation of a directory is the *limited map*. In such a directory implementation, there can only exist a limited number of copies of a cache block in the system. This is illustrated in Figure 4.13, in which processors P_1 and P_2 hold cache block X (shown on the left). In this example, only two copies of a cache block are allowed. When processor P_n reads cache block X , one of the two copies at P_1 or P_2 has to be invalidated (in Figure 4.13, this is P_2 's copy). Clearly, the limited-map implementation requires less storage than the full-map implementation but it also restricts parallelism.

One could also distribute the information on copies, like is done in a *chained directory*. In a chained directory, an entry for a cache block contains a pointer to the head of the 'copy list'. When a processor

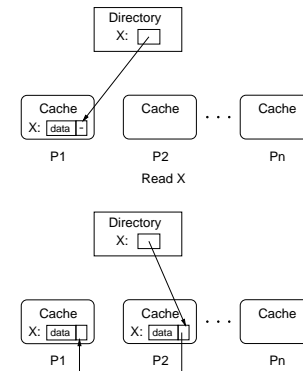


Figure 4.14.

receives a copy of a cache block to read on, the head pointer for that cache block is updated with a pointer to the new owner and the old head pointer is stored in the new owner's local cache. This is shown in Figure 4.14. So, this creates a linked list of copies of a cache block. For example, to invalidate all copies of a cache block, one simply traverses the linked list for that particular cache block.

A drawback of a chained implementation is that replacement of cache blocks may suffer from a higher overhead than in the other directory implementations. When a cache decides to remove a cache block (i.e., it needs to make room for other data), the cache block should be removed from the linked list. In a single linked list implementation, this requires a list traversal to find the previous cache block in the list (see your data structures course). To avoid such list traversals, a double linked list implementation can be applied.

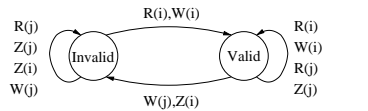
4.7.1 States and state transitions

As was described earlier, cache coherency protocols are based on states and state transitions. Several times already, we mentioned the 'invalidation' of cache blocks. This implies that a cache block in the valid state is put into the invalid state, specifying that the processor cannot use this data anymore. In this section, we will treat the various types of states and state transitions in a bit more detail.

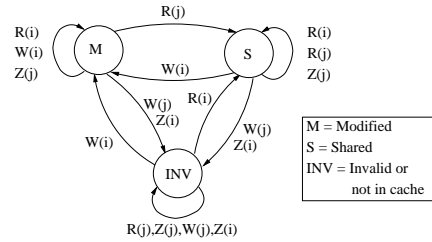
In Figure 4.15, the generic state transition diagrams for a write-through (top) and a write-back cache (bottom) are shown. First, consider the diagram for the write-through cache i . It shows the possible state transitions for a single cache block in this cache i . The cache block can be in one of two states: valid or invalid. Based on actions of either cache i itself or another cache j , the block can move from one state to the other. We have defined three actions: R(ead) from the cache block, W(rite) to the cache block, and replace (= Z) the cache block from the cache. Each of these actions is parameterized with the cache that performs the operation (i = local cache, j = other cache). We assume that when a cache block does not reside in a cache at all, it is in the invalid state.

When an R(i) or W(i) operation is performed in the invalid state, the state of the block moves to valid. It stays there while R(i), W(i), R(j) or Z(j) operations are performed. However, if a W(j) is performed (implying that another processor writes to the cache block), then the state of the block in cache i moves to invalid: the cache block is invalidated. Of course, performing a Z(i) in valid state also brings the cache block in invalid state. Other caches may freely operate on the cache block without affecting cache i when the block in i is in invalid state.

In the state transition diagram for write-back cache i , there are three states: invalid, shared and modified. When reading the cache block (from invalid state), the block is moved to the shared state. Multiple copies of a cache block may be in this shared state (that's why nothing happens at an R(j)). However, when cache i writes to the block, the state is moved to modified. This indicates that the block is dirty and should be written back to main memory when it is replaced. Only a single copy of a cache block can be in the modified state.



State-transition graph of write-through cache i



State-transition graph of write-back cache i

W(i) = write to block by cache i
 R(i) = read block by cache i and j ≠ i
 Z(i) = Replace block in cache i

Figure 4.15.

This can be seen by the W(j) transitions: whenever another cache writes the cache block, the block in cache j is invalidated. While a cache block is in modified state, it can freely be written to (and, of course, read from) by cache i without the need to invalidate other copies of the block since it is unique in the system. A modified cache block moves back to the shared state whenever another cache j reads the block (there is more to this situation, which is discussed below).

Having seen the generic state transition diagrams for write-through and write-back caches, we can now compare the two. The advantage of cache coherency in write-through caches is that the main memory is always consistent. So, if there is a cache miss, then the cache can immediately fetch a consistent copy from main memory. In write-back caches this is not true as there may be an updated version of the data (in modified state) somewhere in a cache which has not been written back to main memory yet. So, if a cache miss occurs and somewhere in the system a modified copy of the cache block exists, then this modified data should be fetched. This can be done in two ways (for now, we assume a read miss):

- The memory transaction of the cache miss is halted and the cache with the modified copy first writes its cache block back to main memory (and changes its state to 'shared') after which the memory transaction is continued to fetch the required data from memory.
- The memory transaction of the cache miss is 'redirected' to fetch the data immediately from the cache with the modified copy. After this, the cache with the modified copy writes the cache block back to main memory and changes its state to 'shared'.

The first solution increases the average latency of cache misses (since dirty copies first need to be written back) while the second solution is more complex to implement. So far, we assumed a read miss. When the cache miss is a write miss, the cache with the modified copy changes the block into the invalid state after the modified data has been sent to the cache with the write miss. In that case, the second solution does not require the write back to main memory (it is left to the reader to find out why).

The disadvantage of write-through caches is of course the additional network traffic they cause. In addition, a cache coherency protocol for write-through caches with only two states suffers from the fact that for

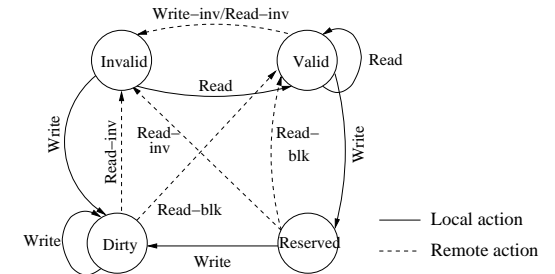
every write action it must be determined whether or not copies of the cache block need to be invalidated. With two states you do not know if a block is unique or shared in the system. This could result in a directory lookup for every write action. In write-back caches, the modified state implies a unique cache block and therefore a modified cache block can be freely written to without the need for invalidating. Naturally, protocols for write-through caches can also have more than two states for the same purpose.

Let us illustrate the above with some example state transitions for a snoopy-bus cache coherency protocol with write-back caches and an MSI state transition diagram such as shown in Figure 4.15. Table 4.1 shows the actions and state transitions for a three processor system. It also specifies the bus actions and indicates where data comes from. In this example, we assume that when a cache block does not reside in a cache at all it has no (—) state.

Proc. action	P1 state	P2 state	P3 state	Bus act.	Data from
P1 read X	S	—	—	Rd	Memory
P3 read X	S	—	S	Rd	Memory
P3 write X	I	—	M	I	—
P2 write X	I	M	I	RdI	P3's cache/memory
P1 read X	S	S	I	Rd	P2's cache/memory
P3 read X	S	S	S	Rd	Memory

Table 4.1.

In the first action, processor P1 reads block X (of which no copies exist) by fetching it from memory and puts the block in the shared state. When processor P3 also reads X, the data is again fetched from main memory and the block is also put in the shared state. Then, processor P3 writes to X. This causes an invalidate transaction for this block on the bus. Processor P1's cache notices this transaction (remember, it is snooping the bus) and invalidates its copy of X. The cache block X in P3 is put in the modified state. When processor P2 writes to X, the modified cache block X is fetched from P3 (directly or via the main memory, see above) while also invalidating all other copies. Block X is put in modified state in P2. Hereafter, processor P1 reads X, which is fetched from P2's cache (again, either directly or via memory). Subsequently, P2 writes X back to memory and changes block X to the shared state. Finally, processor P3 reads X which can be fetched directly from the main memory since there are only shared copies.



Write-once protocol using write-invalidate on write-back caches

Figure 4.16.

Cache coherency protocols in real systems are much more complex than the protocols just described as they attempt to reduce network traffic and the number of invalidations. A classic example of a slightly more complicated (but still not up to the level of today's protocols) cache coherency protocol is the *write-once* protocol. In the state transition diagram of the write-once protocol, shown in Figure 4.16³, there are four

³Note, the transitions for read operations in the reserved/dirty states are omitted as they do not change state.

states: invalid, valid, reserved and dirty. Figure 4.16 assumes a snoopy bus protocol. The normal arrows indicate actions of the local processor on a cache block, while the dashed arrows refer to actions of remote caches (as snooped from the bus) on a copy of the cache block. There are three types of remote actions that can be snooped: *read-blk*: read the cache block from main memory, *read-inv*: read the cache block from main memory (due to cache miss) in order to write to it, and *write-inv*: a remote cache has a cache write hit (i.e., it writes to a valid copy of the cache block). In the case of the read-inv and write-inv transactions, copies of the cache block in question residing in other caches need to be invalidated (the transactions therefore have the extension *-inv*).

The write-once protocol tries to combine the strengths of write-through (\rightarrow consistent memory) and write-back (\rightarrow less network traffic) caches. The protocol exploits two classes of behavior in parallel applications:

1. the processors alternate an execution sequence in which they read a shared data element, perform some computations on it, and write it back to memory again. After one processor has finished this sequence, another processor applies the sequence to the shared data element.
2. a processor performs multiple write (and read) actions on a shared data element before another processor accesses the data element.

For behavior 1 (alternating ‘read-compute-write’ actions,) the write-once protocol acts like a write-through cache: if a cache block is in the valid state and is being written to, then the write action is forwarded to the main memory (write-through) and the block moves to the reserved state. This action also invalidates all copies of the block in the system. So, after a read-compute-write action, the main memory is still up to date. This means that a next read-compute-write action on the shared element by another processor can immediately fetch the required data from memory (rather than getting a modified copy from a cache).

When writing to a block in the reserved state (a second write on the block) or invalid state (both refer to behavior class 2), the block moves to the dirty state. This write action is not forwarded to the main memory (write-back). Hence, multiple writes to a block are not written through to the memory, which reduces network traffic. Also, no invalidations need to be made when writing to a block in the reserved and dirty states because a block in one of these two states is unique in the system.

Another snoopy bus protocol, which is offered by many commodity microprocessors, is the *MESI protocol*. As the name suggests, there are four states in this protocol:

- M(odified) : a dirty, exclusive (= unique) cache block
- E(xclusive) : a clean, exclusive cache block
- S(hared) : a clean, shared cache block
- I(nvalid) : a block which is not resident in cache

The exclusive state has been added to reduce invalidation overhead: the cache can freely write a cache block in the exclusive state without the need to invalidate copies in the system. To support the exclusive state, an extra status line needs to be added to the bus, which signals whether or not a copy of a cache block exists in the system. If at a read miss this status line is set, then the fetched cache block will be put in the shared state, otherwise (there is no copy in the system) it will be put in the exclusive state.

4.7.2 Cache coherency in a cache hierarchy

So far, we talked about cache coherency in multiprocessor architectures with only one level of caches. In modern systems, processors have two or three levels of caches. So, what happens to the cache coherency protocol when there are multiple levels of caches? For example, you would not like to have snooping logic at each of these cache levels. A solution which is commonly used is by preserving the *inclusion property*. Assuming two cache levels, the inclusion property can be defined by the following two requirements:

1. If a cache block is in the L_1 cache, then it also must be in the L_2 cache.
2. If a block is in the modified state in the L_1 cache, then it must also be marked as modified in the L_2 cache. This can, for example, be accomplished by a write-through L_1 cache.

When a multiprocessor system with two levels of caches with, for example, a snoopy bus protocol adheres to the inclusion property, only the L_2 caches need to have snooping logic.

The first requirement for inclusion is not trivial to achieve. Differences in block size, associativity, etc. between the caches at the different levels may result in data being cached at level 1 and not being cached at level > 1 . For example, when the L_1 cache is set-associative and the L_2 cache is direct mapped, two cache blocks may reside in the L_1 cache at the same time while in the L_2 cache they map onto the same entry.

When a 2-level cache hierarchy is restricted to the following limitations, automatic inclusion is achieved: the L_1 must be direct-mapped and the L_2 either direct-mapped or set-associative. The block sizes must be identical and the sets $L_1 \leq \text{sets}_{L_2}$. It is left to the reader to check that such a setup achieves inclusion.

4.7.3 Scalable Coherent Interface (SCI)

To finalize the discussion on cache coherency, the Scalable Coherent Interface (SCI) is worth mentioning. SCI is a network standard which attempts to be a commodity network standard for multiprocessor machines (so far, this has not worked). The SCI standard defines a scalable point-to-point interconnection network which works in a bus-like manner. This means that communication is not stream based (i.e., pack data into a message \rightarrow transmit the message over the network as a stream \rightarrow unpack and interpret the data from a message), but address based. So, you can directly transmit read and write operations with an accompanying address along the communication lines. Clearly, this is very efficient for shared memory machines.

The SCI standard also includes a cache coherency protocol. This is a write-invalidate protocol using chained directories. The directories apply double linked lists to store the sharers of cache blocks, where the head of a list is the only sharer which may modify its copy of the data. So, in order to get write access to a cache block, a cache first needs to get ahead of the list.

The standard defines communication links of up to 8 GBytes/s. Current implementations, however, typically reach 1.25 Gbit/s. Example machines that use SCI for their networks, are the Convex Exemplar and Data General CC-NUMA server. Both machines apply SCI links in a ring topology.

4.8 Synchronization

Besides cache coherency, hardware-supported synchronization also plays an important role in multiprocessors. Such synchronization is similar to software based (OS-level) synchronization for critical sections (e.g., semaphores, monitors) and prevents that multiple processors are accessing shared data at the same time which may result in race conditions. Hardware-supported synchronization is established using *lock* and *unlock* (software-)functions which are built around atomic *read-modify-write* instructions. An example of such an atomic read-modify-write instruction is the *test-and-set* instruction, of which the execution semantics can be defined using the following C code:

```
test-and-set( int *address ) {
    temp = *address;
    *address = 1;
    return (temp);
}
```

The old contents of a memory location (referred to by the address operand) are read and returned, while also writing ‘1’ to the memory location. This is all done as an atomic operation which cannot be interrupted by other memory transactions. To see how the test-and-set instruction is used to construct locks, consider the C code for the *lock* and *unlock* functions below.

```
lock( int *lock ) {
    while ( test-and-set( lock ) == 1 );
}

unlock( int *lock ) {
    *lock = 0;
}
```

A lock is successfully grabbed whenever the test-and-set instruction returned a '0'. This requires the execution of the *unlock* function to write a '0' into the memory location associated with the lock. So, while the test-and-set instruction returns a '1', the lock is held by another processor. Because the loop 'spins' until the lock is successfully grabbed, this lock implementation is called a *spin lock*. An alternative implementation type is the so-called *suspend lock*. With a suspend lock, a processor is stalled (i.e., put to sleep) until it receives an interrupt notifying that it has successfully grabbed the lock. Clearly, such suspend locks are more complex to implement than spin locks.

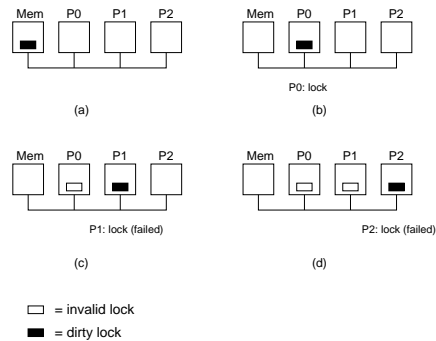


Figure 4.17.

Spin locks may cause *thrashing*. This is illustrated in Figure 4.17. In the situation depicted in Figure 4.17(a), none of the processors acquired the lock. In (b), P0 successfully grabs the lock and writes a '1' to the associated memory location. Assuming write-back caches, P0's cache now contains a dirty lock. Subsequently, in (c), P1 tries to grab the lock by executing a test-and-set instruction. To this end, P1 gets a copy of P0's cache block containing the dirty lock after which P1 writes a '1' to it. P1 fails to grab the lock since its test-and-set will return a '1' (P0 still holds the lock). But because it wrote to the lock's memory location, the copy of the lock in P0 is invalidated. Then, P2 tries to acquire the lock and, like P1, fails to do so. Because P2's test-and-set instruction again writes to the lock, it invalidates the copy at P1. So, each subsequent execution of a test-and-set instruction causes copies of the lock in other caches to be invalidated. When multiple processors are contending for a lock, this means that there can be a cache miss on the lock's memory location for every executed test-and-set instruction.

To avoid such thrashing, the code of the *lock* function could be changed as follows:

```
lock( int *lock ) {
    while ( test-and-set( lock ) == 1 )
        while ( *lock == 1 );
}
```

This is called a *snooping lock*. The snooping lock mainly spins on the inner loop, which only *reads* the memory location associated with the lock. Clearly, this causes no invalidations. When the read action of the inner loop returns a '0' (implying that another processor released the lock), the outer loop quickly tries to grab the lock using the atomic test-and-set instruction. If it fails to do so (i.e., another processor snapped the lock), then it returns to the inner loop again. Alternatively, one could also use a so-called *test-and-test-and-set lock*:

```
lock( int *lock ) {
    for (;;) {
        while ( *lock == 1 );
        if ( test-and-set( lock ) == 0 )
            break;
    }
}
```

4.8.1 Barrier synchronization

Another type of synchronization which can be important to parallel applications is *barrier synchronization*. In barrier synchronization, which applies to both multicomputer and multiprocessor architectures, the processors wait for each other until they all have reached a certain point of execution. Naturally, a barrier synchronization can be implemented in software, like MPI does, but some architectures also provide hardware support for it. For example, a shared counter could be decreased every time a processor reaches the barrier. When the counter reaches zero, the barrier synchronization has completed implying that the participating processors can continue execution.

Another method for hardware barrier synchronization is by providing hardwired barrier lines such as illustrated in Figure 4.18. In this example, a processor pulls its barrier line down when it has reached the barrier. By performing a NOR operation on all barrier lines, a processor can determine whether or not all processors have reached the barrier. Of course, this scheme can also be implemented using the AND operation.

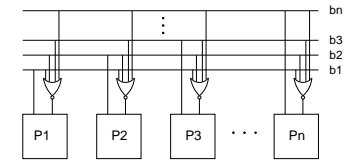


Figure 4.18.

4.9 Disk storage considerations

Parallel applications typically operate on vast amounts of data, which does not always fit into the main memory (e.g., parallel database applications). We therefore also have a brief look at the RAID (Redundant Array of Inexpensive Disks) concept since RAID systems are often applied in parallel computers as secondary storage.

In general, RAIDs improve on the fault tolerance and performance as compared to traditional disks. In a RAID system, the data is *striped* over multiple disks. In some cases (we will explain this below), the disks can be accessed in parallel by different processors which is of course important to parallel computers (e.g., SMPs). There are 7 RAID levels (counting from 0), each with a different scheme to provide redundancy and, as a consequence, each obtaining different performance characteristics. RAID levels 1 up to 5 can survive one disk crash, while level 6 can survive two of them.

Let us have a closer look at some of RAID levels. In level 0, there is no redundancy at all. Data is simply striped over the disks. This means that access to the disks can be fully parallel. Consequently, RAID level 0 yields the highest performance of all levels. In level 1, data is mirrored on an extra set of disks. So, this requires twice the number of disks. Using this scheme, recovering from a disk crash is fast since you simply switch over to the mirror disk. In level 3, called Bit-interleaved Parity, data is striped over the disks at the granularity of bits. So, the bits of a byte are striped over the disks. In addition, there is one extra disk containing parity bits, you can reconstruct a byte after a single disk has crashed. However, because striping is done at bit granularity, read and write operations should be sent to *all* disks. Consequently, there is no parallel disk access by multiple processors. In RAID levels 4 and 5, data is striped at the granularity of blocks, as illustrated in Figure 4.19. In level 4, called Block-interleaved Parity, there is one redundant disk containing parity information with which blocks can be reconstructed after a single disk crash. Because data is striped at block granularity, reads can be performed in parallel but writes still need to access all disks: the parity information needs to be rebuilt after a write. To rebuild the parity information all data blocks associated with the parity block need to be read. For example, when writing to block 1 in Figure 4.19, block P0 must be rebuilt by reading blocks 0 up to 3. A drawback of RAID level 4 is that the parity disk may become a bottleneck. Therefore, RAID level 5 (called Block-interleaved Distributed Parity) solves this problem by distributing the parity information over the disks in a round-robin fashion. This is also illustrated in Figure 4.19.

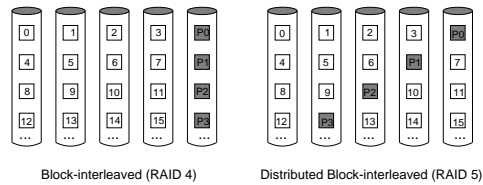


Figure 4.19.

4.10 Real multiprocessors

We conclude the chapter on multiprocessor architecture by showing some highlights of three commercial machines, of which the first is the SGI Origin 2000.

4.10.1 The SGI Origin 2000

The Origin 2000 is a CC-NUMA architecture. Figure 4.20 depicts the architecture of a single node of the Origin 2000. In the node, two MIPS R10000 microprocessors are connected via a crossbar (the 'Hub') to a local shared memory. Also attached to the crossbar are a Crossbow I/O interface (which is also built around a crossbar) and a router. The routers in the system nodes are connected in a scalable point-to-point network with a hypercube topology. This is shown in Figure 4.21.

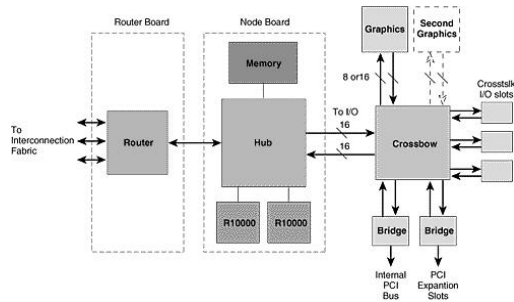


Figure 4.20.

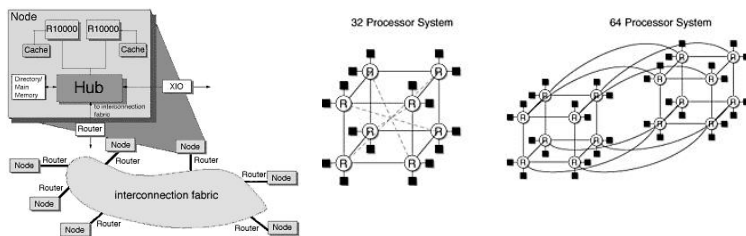


Figure 4.21.

On each node, a full-map directory is integrated to implement a write-invalidate cache coherency protocol. Discussing this protocol is beyond the scope of the syllabus. The machine also provides hardware support for page replication and migration.

4.10.2 The Cray T3D

The Cray T3D is a NUMA machine. It does not provide hardware support for cache coherency. This means that caches only hold data that is locally used by a processor, while writable shared data is not being cached. On the left in Figure 4.22, a node of the T3D is depicted. It contains two Dec Alpha 21064's which are connected to a network interface and a block transfer engine. The latter is a DMA device which can transfer large blocks of memory to or from remote memories.

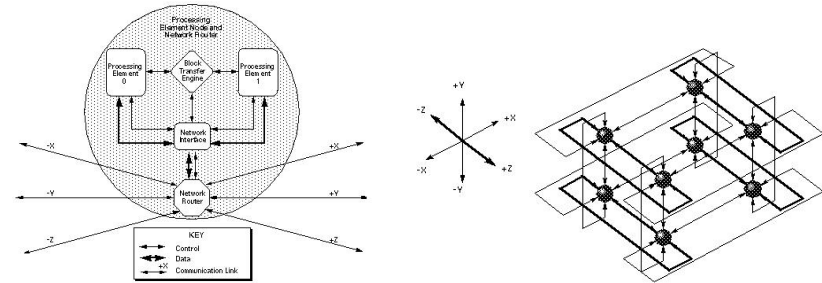


Figure 4.22.

The network interface on a node is connected to a network router which implements the switching and routing of packets in the message-passing network. In the T3D, this message-passing network is a three-dimensional torus as shown on the right in Figure 4.22.

The router architecture is shown on the left in Figure 4.23. It routes packets using a dimension ordering method. At the top of the router, packets come in from the network interface. They are first routed into the X direction by the X dimension switch. When the correct X coordinate has been reached, packets are moved to a Y dimension switch to route them in the Y direction. Having reached the correct Y coordinate, packets are routed in the Z direction to their final destination. At the destination, packets leave the router again through the network interface.

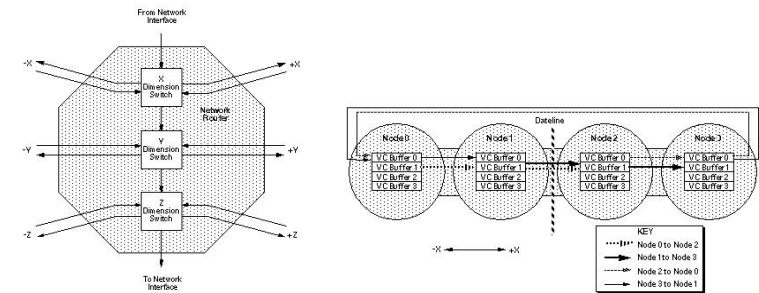


Figure 4.23.

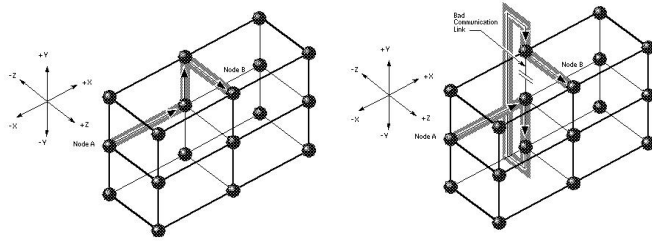


Figure 4.24.

In Figure 4.24, two routing examples are shown. On the left, a packet is sent from node A to node B using the dimension ordering method discussed above and via a shortest path. The routers in the T3D have some fault tolerance built in since they are able to circumvent broken links. This is illustrated on the right in Figure 4.24. Again, a packet is routed from node A to node B, but now with a broken channel. Because it is a torus network, the packet can also follow the -Y direction instead of the +Y direction to reach node B.

Since the torus network has wrap-arounds, and thus inherent cycles in its topology, care must be taken that no routing deadlocks occur. To this end, the T3D applies virtual channels, as shown on the right in Figure 4.23. In each direction, there are two virtual channels (in Figure 4.23 only the +X virtual channels are shown) which are connected to the buffers such that no deadlocks can occur.

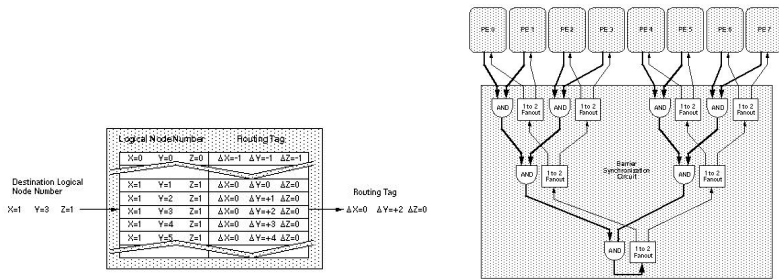


Figure 4.25.

The T3D uses logical coordinates for its nodes and the routers use a translation table to find the actual, physical node associated with a logical coordinate. Such a translation table is shown on the left in Figure 4.25. In the left column of the table, the logical coordinates of processors are listed, whereas the right column indicates how messages should be routed to reach the associated physical node. This logical→physical coordinate translation has a big advantage: broken nodes can be ‘replaced’ by activating one of the redundant nodes (a T3D is shipped with such redundant nodes) in the system. This can be done by simply updating the routing information in the translation tables.

The T3D also offers hardware support for barrier synchronization, as shown on the right in Figure 4.25. All processors have hardwired barrier lines which are connected in a binary tree topology using AND gates. A processor activates its barrier line when it has reached the barrier. If the AND gate at the root of the tree yields a ‘1’, then all processors have reached the barrier. Of course, the result of the root AND gate is routed back to all the processors so that they know when to continue with processing.

4.10.3 The Cray MTA

The last multiprocessor to be discussed is the Cray MTA. This is a UMA machine, but a very unusual one. MTA stands for MultiThreaded Architecture: a processor in the Cray MTA can have 128 concurrently executing *hardware threads*⁴, where each thread has its own flow of control. The global machine organization of the MTA is shown in Figure 4.26. There are four types of resources in the machine: processors, I/O processors, I/O caches and memories. These resources are distributed over the network, which has a $16 \times 16 \times 16$ 3D toroidal mesh topology (shown at the bottom of Figure 4.26). Note that the horizontal and vertical rings in this topology are interleaved in the third dimension. This results in a node degree of four, whereas a normal 3D torus (see the Cray T3D) requires a node degree of six. Consequently, the unused communication ports at the nodes may be used for connecting resources.

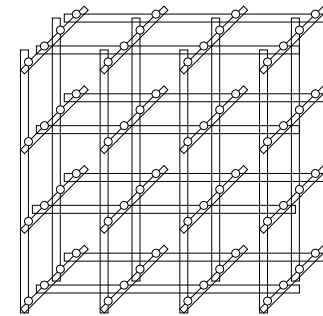
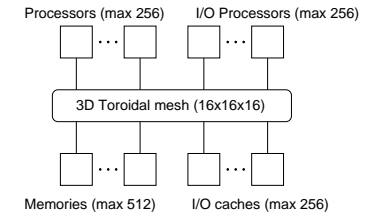


Figure 4.26.

As explained, a processor in the MTA can concurrently execute 128 hardware threads. To this end, each thread requires its own context. Therefore, there are 32 general purpose registers (GPRs) for each thread, yielding a total $128 * 32 = 4096$ GPRs per processor! Besides these GPRs, there also are several special purpose registers per thread. This is shown in Figure 4.27. Examples of special purpose registers that need to be stored on a per-thread basis are the program counter (PC) and the status register (keeping, for example, condition bits).

The processor schedules the threads for execution. Because the threading mechanism is implemented in hardware, the thread switching can be done in a single cycle. So, at every instruction a new thread may be scheduled for execution. The threading in the MTA machine allows for an unusual ‘optimization’: the machine does not have any data caches! As a consequence, there is no cache coherency problem, implying that complex coherency protocols are not needed at all. Instead, the MTA machine *hides* memory latencies by simply scheduling another thread for execution. So, when a certain thread performs a memory access, another thread starts executing. For this purpose, the entire machine is pipelined, as shown in Figure 4.28. At each cycle, a (different) thread can inject an instruction into this pipeline.

⁴Not to be confused with software threads such as offered by operating systems or packages like Pthreads.

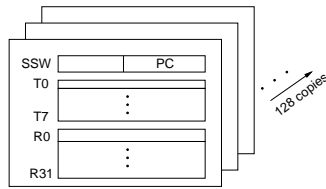


Figure 4.27.

The processor can always inject instructions from different threads into the pipeline, since threads are independent of each other. So, as long as all active instructions within the pipeline are from different threads, there are no dependency problems. It may however be hard to keep the pipeline filled this way: it means that there should be enough runnable threads (this is a task of the compiler and programmer) to avoid bubbles in the pipeline. Therefore, to keep the pipeline filled, the machine also allows for injecting multiple instructions from the same thread into the pipeline. To do so, the instructions from the same thread which are active in the pipeline must be independent of each other. To check this condition, each instruction contains a *lookahead* (set by the compiler) which specifies the number of succeeding, independent instructions following the instruction in question.

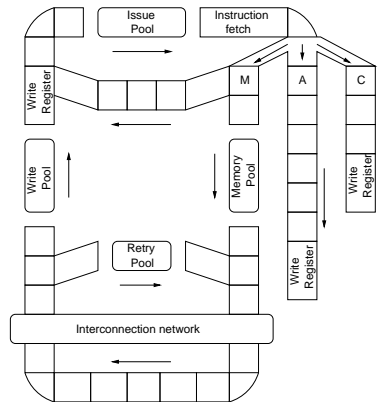


Figure 4.28.

The Cray MTA also exploits instruction-level parallelism since threads consist of so-called LIW (Large Instruction Word) instructions containing three operations. One of these operations is arithmetic (A), one is a memory access (M) and the third one can be a branch or a simple arithmetic operation (C). Like in the VLIW paradigm, the operations in an LIW instruction can be executed in parallel.

Another nice feature of the Cray MTA is the *tagged* memory. Each memory location has a number of tag bits associated with it. These tag bits can be used in several ways. First, one could set traps with the bits which can be very useful for debugging purposes. So, whenever a memory location with a certain tag value is accessed, a trap could be generated. Second, a memory location with a certain tag-bit setting, can be handled by the processor as an indirection (i.e., a pointer to another location). So, instead of accessing the initial memory location, the memory access is forwarded to another location. Third, using full/empty tag bits, synchronization can be established. For example, a read to the memory does not complete until the full bit has been set.

Chapter 5

Other parallel machines

Having discussed the two most popular classes of parallel machines, namely multicomputers and multiprocessors, there also exist other types of parallel machines. In this chapter, we will have a (very) brief look at two of these alternative parallel computer architectures.

5.1 Supercomputers

Traditional supercomputers remain an important player in the field of parallel processing as certain application domains (such as weather forecasting applications) still heavily rely on them. Supercomputers contain a relatively small number of processors, often between 2 and 64. A processor consists of a high-performance RISC-like scalar unit and a number of vector units for vector processing. Supercomputers typically provide a shared address space, which in some machines is physically shared (UMA) and in others physically distributed (NUMA). Most of the development costs of these machines is spent on the interconnect between the processors and the memory modules/banks. The reason for this is that the memory bandwidth should be extremely high in order to efficiently serve the vector units with the vector data.

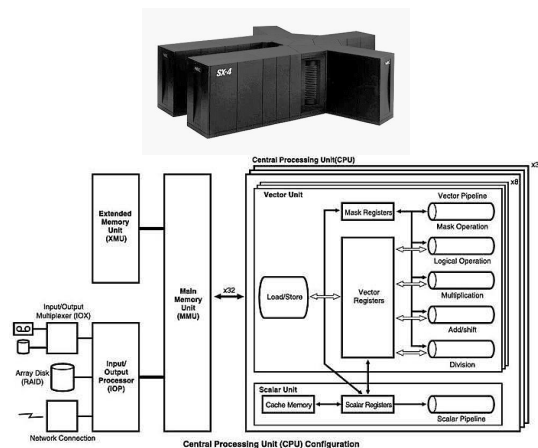


Figure 5.1.

Well-known examples of supercomputers are the NEC SX4, Cray Y-MP and Cray T-90. In Figure 5.1, a picture of a NEC SX4 together with an overview of its architecture are shown. As can be seen, a processor

in the SX4 consists of a scalar unit and eight vector units with each six vector pipelines. In the entire system, there can be 32 of such processors.

5.2 SIMD array architectures

Vector processors are called data-parallel since they exploit data parallelism. Another large class of data-parallel machines, which was very popular in the seventies and early eighties, is that of SIMD array architectures. In SIMD array machines there are a large number (in the range of 100-16,384) of typically straightforward (sometimes even 1-bit) processors. These processors are synchronized as they all execute the same instruction in lock step. Sometimes, a processor can choose to 'skip' the execution of a certain instruction by means of instruction masks. Each processor executes the global instruction on its own local data. This data can be located in a local memory of the processor (distributed memory, which is the most common implementation) or in a shared memory.

The network between the processors can be point-to-point or it can be an indirect multistage network. In the latter case, the processors can use one of the predefined routing functions supported by a multistage network, such as broadcast, permutation, shift, etc., in order to perform interprocessor communication.

SIMD architectures are scarcely used anymore and almost entirely replaced by MIMD architectures. For some application domains in which very fine grained parallelism is abundant (like in image processing), SIMD architectures could (and possibly are) still be applied.

However, SIMD-like techniques *are* widely-used in a different context, namely inside microprocessors. Multimedia instructions (such as Pentium's MMX instructions) perform a single operation on a small number of data elements in parallel. These data elements are stored in the instruction's operand registers.

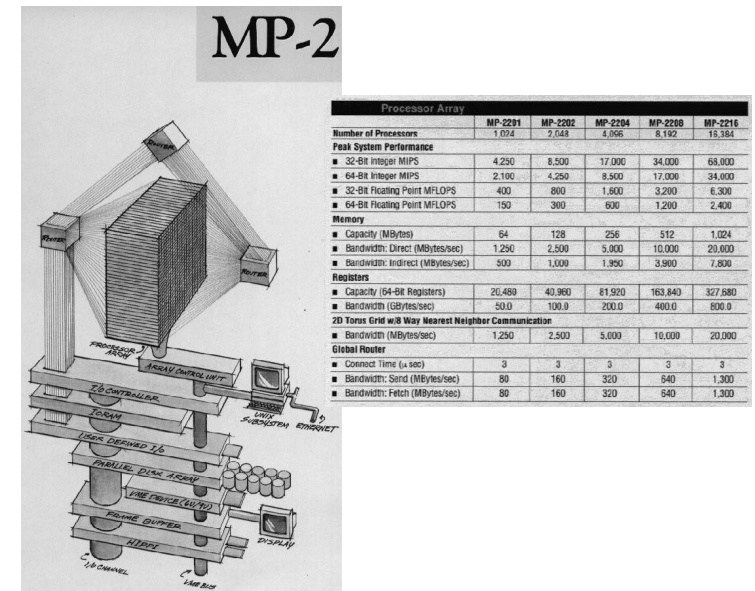


Figure 5.2.

An example of a SIMD array architecture (which is not produced anymore) is depicted in Figure 5.2, which shows (some characteristics of) the MasPar 2 machine. The MasPar 2 can have up to 16K processors,

which are connected using two networks. First, there is a torus network with a node degree of eight. These eight communication channels per processor are used to connect all neighboring processors, including those diagonal to a processor. The second network consist global routers, with which any processor can be reached in a few hops.

Appendix: example examinations

This appendix contains two example examinations with which you can practice for a real examination.

Example examination 1

1. High performance processors & memory hierarchy

1. Explain why most of today's parallel systems contain commodity processors instead of special-purpose processors.
2. Explain why caches are of great importance to current microprocessors and describe three types of caches.
3. Assume a 2-level cache hierarchy with a cache hit ratio H_1 for the level 1 cache and H_2 for the level 2 cache. The cache access latencies are C_1 and C_2 respectively. The access latency of the main memory is M . Give a formula for the average memory access time of this system.

2. Networks

1. Make a comparison between a bus, multi-stage network and crossbar in terms of latency, bandwidth and costs.
2. Explain how a pipelined bus (split transaction bus) yields higher throughput than a normal bus.
3. Explain the difference between a blocking and non-blocking multi-stage network.
4. Assume a 16×16 Omega network using 2×2 switches. Show the switch settings for routing a message from node 6 to node 2 and from node 10 to node 7 simultaneously. Does blocking exist in this case?

3. Routing and switching

1. Explain the advantage of virtual cut-through switching over wormhole routing.
2. The communication latencies for store&forward switching and wormhole routing are expressed by:

$$T_{s\&f} = \frac{L}{W} \times D$$

$$T_{wormhole} = \frac{L}{W} + \frac{F}{W}(D - 1)$$

where L is the packet length in bits, W the channel bandwidth in bits/s, D the distance (number of hops) and F the flit length in bits. Explain both formulas.

3. Explain why source-based routing can never be adaptive.
4. Describe the advantages and disadvantages of adaptive routing.

4. Shared memory computers

1. Explain why page migration/replication in (CC-)NUMA systems is difficult to realize.
2. Data prefetching and multi-threading are techniques that can help to hide latencies in (CC-)NUMA systems. Give a rough indication how these techniques could be supported by extending (the architecture of) traditional commodity processors.
3. Explain why S-COMA is more cost-efficient than normal COMA.

5. Cache coherency

1. Describe how snoopy-bus cache coherency protocols work.
2. Why are snoopy-bus protocols uncommon in CC-NUMA machines?
3. Explain why coherency protocols for write-back caches have separated Read-Write and Read-Only states.

6. Performance evaluation of computer architectures

1. Explain why abstraction level is an important issue in computer architecture simulation.
2. Dynamic instruction predecoding is an optimization technique for instruction level simulation. Instruction predecoding can also be performed statically (as a preprocessing step before actual simulation). Explain how this would work and try to give some advantages and disadvantages of static instruction predecoding.
3. Give two potential problems of trace-driven simulation.

Answers

1

1. The development of special-purpose processors is a costly endeavor and it's hard to compete, performance wise, with the large players in the world of commodity processors. The latter is important because Amhdal's law dictates that sequential performance should not be forgotten.
2. The performance gap between processors and main memory is growing. Nowadays, it takes hundreds to thousands processor cycles to load a piece of data from memory (and during these cycles, at least one processor pipeline is stalled). This gap can be closed (at least, the most of it) with caches. Three types (note: not implementations!) are: the TLB for caching virtual address translations, the instruction and data caches.
3. The level-1 cache always needs to be accessed (to check data is available or not). The level-2 cache will only be accessed in the case of a level-1 cache miss (chance = $1 - H_1$). The main memory will only be accessed in the case of a level-1 and a level-2 miss (chance = $(1 - H_1) * (1 - H_2)$). So the average memory access time is: $T_m = C_1 + (1 - H_1) * C_2 + (1 - H_1) * (1 - H_2) * M = C_1 + (1 - H_1) * (C_2 + (1 - H_2) * M)$

2

1. **Bus:** the bandwidth is linear to the width of the bus and is also dependent on the access time (e.g. a traditional bus v.s. a split-transaction bus). The latency is mostly constant and dependent on the arbitration delay and the bus protocol. The costs scale with the bus width and the number of resources attached to the bus. **Multi-stage network (MIN):** when using an $n \times n$ MIN with $k \times k$ switches, the overall bandwidth is between the bandwidth of 1 communication channel (when there is blocking) and that of n channels (when there is no blocking). The latency depends on the number of hops to be made (= number of stages = $O(\log_k n)$). The costs depends on the number of switches and the bandwidth of

the channels. **Crossbar:** when using a $n \times n$ crossbar, then the bandwidth ranges from the bandwidth of 1 channel to that of n channels (again, depending on blocking or non-blocking). The latency is constant and dependent on the time to allocate a channel. The costs are dependent on the number of switches ($= O(n^2)$).

2. In a traditional bus, only a single transaction can be active at one moment in time. In a split transaction bus, there are separate address and data busses and transactions are split into address and data transactions. Reading a piece of data from the memory, e.g., is done by putting the address on the address bus. After the address reached the memory, the address bus can again be used for the next transaction (while the memory may still be fetching the data for the previous request). When the memory fetched a chunk of data, it is sent back to the processor over the data bus. Attached to the data is a tag describing to which request this data belonged.
3. In a blocking MIN, not all permutations (connections from input ports to output ports) are possible without causing a switching conflict. In the case of such a conflict, one or more communication streams needs to be blocked.
4. There is no blocking (there is only one crossover between the communications).

3

1. Wormhole routing may cause tree saturation as flitbuffers are small and when the head of a message is blocked, the tail remains distributed over multiple switches in the network and it keeps the allocated channels blocked (causing tree saturation). In virtual cut-through switching, the buffers at the switches are at least of the size of an entire message. This means that when a header flit is blocked, the tail of the message continues until the entire message is buffered at the switch of the blocking header flit. Clearly, this reduces the amount of tree saturation as channels will be freed earlier.
2. Store&forward: sending a whole packet to the next hop takes $\frac{L}{W}$. This needs to be done D times (there are D hops), which results in $T_{s\&f} = \frac{L}{W} \times D$. For wormhole routing: it takes $\frac{L}{W}$ to transfer the tail flit to the first hop. After that, it takes another $\frac{L}{W}(D-1)$ to transfer the tailflit to the final destination (there are $D-1$ hops left).
3. In source-based routing, the sending side computes the path the message should take and stores this path in the message header. This path is fixed and cannot be adapted during the transmission of the message.
4. Advantages: less prone to network contention (so, potentially higher throughput), more fault tolerant. Disadvantages: out-of-order receipt of packets, need deadlock/starvation avoidance support, more complex/expensive switches

4

1. It needs to be implemented at the OS-level (which is complex and slow). Furthermore, as the granularity of sharing/coherency are whole pages, the chance of false sharing between processors is quite large.
2. For data prefetching, the processors need to be extended with at least a prefetch engine which determines *when* and *where* (i.e., which data) to prefetch data. In addition, rather than simply storing the prefetched data in the data cache, extra buffer space can be added for this purpose. The latter overcomes the trashing of the data cache in the case bogus data (which will not be used) is prefetched. For multi-threading, the processors need a lot of extra registers, both for storing per-thread data and for storing per-thread status (PC, status bits, etc.). In addition, a scheduling mechanism must be added for scheduling the threads.
3. In S-COMA you use the MMU to determine whether a page is located in local memory or not. This means that you do not need the tag memory and comparators used in COMA. As a result, the OS-managed main memory can be fully associative which is not feasible for COMA (would take too many

comparators). However, DRAM cache misses in S-COMA may be slower than in COMA because of the OS-level handling of page faults.

5

1. In snoopy-bus protocols, every cache (as part of a processor) attached to the bus monitors all bus transactions. When a snooping cache detects a bus transaction which involves an address which is also present in the snooping cache, appropriate actions are taken. In a write-invalidate protocol, this means that write actions of one cache in the system will invalidate all other copies of the data in the other caches. For write-update systems, the writing cache needs to update all copies in other caches. For snoopy-bus protocols it is important that caches "announce" enough information about their actions (e.g. announce all write actions, including cache hits) so that other (snooping) caches can react on these actions.
2. CC-NUMA machines typically are not entirely bus based. They often consist of SMP clusters connected via a direct or multi-stage network. Broadcasting of information to all processors in the system, as needed by snoopy-bus protocols, is therefore harder and more expensive (performance-wise) to implement.
3. Two reasons: in a write-back cache you should know if a block is dirty in order to write it back to memory when the block is invalidated by another cache (which wants to write the block) or when it simply is evicted. Moreover, once a cache block is in Read-Write state, the cache owning the block can write to it without needing to send invalidation messages over the bus (there are no copies).

6

1. Abstract architecture models allow for fast simulation and rapid construction of the model but with limited accuracy, whereas detailed models are much slower and harder to build but also more accurate. Typically, a designer starts to use more abstract models (with which he/she can perform many experiments, e.g., explore a large design space) and gradually refines the models to perform more focused and accurate experiments.
2. Before execution, the executable is first translated into an intermediate predecoded format which can be handled efficiently by the simulation engine (e.g. references to modeled registers are explicitly put in the intermediate format). Advantage: more efficient simulation than dynamic instruction predecoding since there is no on-the-fly predecoding. Disadvantages: starting up the simulation takes longer (the executable first needs to be predecoded) and the size of the executable will be larger.
3. 1. It's hard to get complete traces (e.g. tracing the kernel, tracing (dynamic) libraries, etc.) 2. When tracing applications, distortion is a problem. By monitoring the application you may affect its behaviour and thereby generating bogus trace entries.

Example examination 2

1. Netwerken

1. (3 punten)

Latency en throughput zijn belangrijke metrieken voor netwerken van parallele systemen. Over het algemeen is het moeilijk om latency te reduceren. Het verhogen van de netwerk throughput is makkelijker. Beschrijf twee methoden om de netwerk throughput te verhogen, één in de context van bus-systemen en één in de context van (wormhole-routed) message-passing netwerken.

Antwoord: 1ste methode: pipelined busses of split-transaction busses. In deze busses. Even in het Engels (tekst had ik nog liggen): In a traditional bus, only a single transaction can be active at one moment in time. In a split transaction bus, there are separate address and data busses and transactions are split into address and data transactions. Reading a piece of data from the memory, e.g., is done by putting the address on the address bus. After the address reached the memory, the address bus can again be used for the next transaction (while the memory may still be fetching the data for the previous request).

When the memory fetched a chunk of data, it is sent back to the processor over the data bus. Attached to the data is a tag describing to which request this data belonged. 2e methode: virtuele communicatie kanalen. Deze kanalen worden (time) ge-multiplex'ed over een fysieke comm. kanaal. Hiervoor is bufferruimte per virt. kanaal nodig. Met b.v. een round-robin strategie kan elk virt. kanaal een fit sturen "per ronde" sturen.

2. (3 punten)

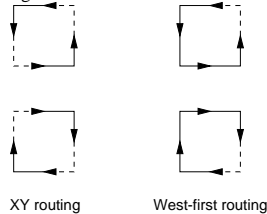
In plaats van het reduceren van latency (wat men hoofdzakelijk met caches doet), kan men ook latency verbergen, het zgn. latency hiding. Beschrijf 2 technieken in de context van NUMA shared memory systemen om de latency van "remote" memory referenties te verbergen.

Antwoord: 1ste methode: data prefetching. Haal data op van een remote processor voordat het werkelijk nodig is (de lokale processor is dus nog lekker aan het rekenen en tegen de tijd dat het de data nodig heeft, is het er al). Hiervoor is dus predictie nodig (voorspel welke data in de toekomst nodig zal zijn). Veel loop structuren in applicaties zijn goed voorspelbaar. 2e methode: hardware multithreading. wanneer een (hardware) thread een cache miss heeft en/of data van een remote processor moet halen, schedule dan een andere thread. Dit kan binnen 1 cycle.

3. (4 punten)

Beschrijf de XY en West-first routing technieken en leg het voordeel van West-first routing ten opzichte van XY routing uit. Leg ook uit waarom er geen deadlock kan ontstaan in beide routing technieken in het geval van een mesh netwerk.

Antwoord: XY: route eerst naar de juiste X coördinaat, dan naar de juiste Y coördinaat. West-first: route eerst naar het westen (indien nodig), daarna mag er vrij naar het noorden, zuiden en oosten worden ge-route. Voordeel van West-first geïllustreerd met "turn-model":



De West-first methode is dus meer adaptief (laat meer turn toe dan XY). Omdat beide methoden geen cycle in hun turn model hebben, kunnen er geen deadlocks ontstaan in een netwerk wat ook geen cycles bevat (mesh).

2. Multiprocessor systemen

1. (4 punten)

Beschrijf de verschillen tussen CC-NUMA en COMA multiprocessor architecturen. Neem in deze beschrijving zowel de verschillen in fysieke architectuur en de verschillen op performance gebied mee.

Antwoord: CC-NUMA: normaal gesproken heeft data een vast thuis-locatie. Wanneer data niet aanwezig is bij lokale processor, krijgt het een kopie van de desbetreffende processor (op cache-blok granulariteit) en wordt deze in de lokale cache neergezet. Dus remote data wordt in de lokale cache neergezet (deze is/zijn over het algemeen klein, dus wanneer er veel remote data opgehaald moet worden, is dit een probleem (slechte performance: veel cache misses). Mogelijke oplossing: pagina migratie/replicatie door het OS (langzaam, complex en grote kans op false-sharing). In COMA systemen heeft data geen vaste thuis-locatie en werken de DRAM geheugens van de processoren als DRAM caches. Hiervoor is extra HW nodig (tag memory, state memory en comparators) dus duurder. Minder problemen met het gebruik van veel remote data (DRAM caches zijn groot) en transparante implementatie van data migratie/replicatie. Omdat migratie/replicatie op de granulariteit van DRAM cache blokken gaat, kun je die redelijk klein maken om false sharing te beperken. Verder zijn remote data accessen wel duurder in COMA (de data moet expliciet in de machine gezocht worden) en is het coherency protocol lastiger te implementeren (b.v. ervoor zorgen dat niet de laatste kopie van een stukje data weggegooid wordt).

2. (2 punten)

Leg uit waarom er in COMA systemen vaak gebruik wordt gemaakt van de message-combining techniek.

Antwoord: De communicatie structuur van COMA systemen is vaak een boom, waarin de switch van een knoop een directory heeft met data elementen die aanwezig zijn in de onderliggende subbomen. Het zoeken naar data is dan dus een boom traversal. Wanneer 2 processoren zoeken naar de zelfde data en de aanvragen komen langs dezelfde knoop in de boom, dan kunnen de aanvragen gecombineerd worden in 1 aanvraag. Als de data terugkomt, worden er netjes 2 kopieën naar de desbetreffende processoren in de subbomen gestuurd.

3. (4 punten)

Verklaar waarom de DRAM caches in gewone COMA systemen niet volledig associatief (fully-associative) kunnen zijn en waarom dit wel kan in Simple-COMA systemen. Leg ook uit wat de verschillen en overeenkomsten zijn tussen Simple-COMA systemen en traditionele (software) SVM (Shared Virtual Memory) systemen.

Antwoord: De DRAM caches zijn groot met veel DRAM-cache entries. Wanneer dit volledig associatief geïmplementeerd zou worden (elk DRAM cache blok kan dan overall in het DRAM staan), dan zou dit enorm veel comparators vergen (eentje voor elke DRAM cache entry. Dit is niet te betalen. In Simple-COMA, wordt een hele pagina ge-allocceerd wanneer een stuk data nodig is. Deze pagina wordt mbv de MMU in het DRAM neergezet (normale OSes doen dat ook)...dit kan makkelijk FA gebeuren omdat je eenvoudig een pagetable gebruikt om pagina's terug te vinden. Tot zover komt Simple-COMA overeen met (software) SVM. Echter, in SVM wordt coherency ook op pagina nivo gedaan (veel false sharing dus), waar Simple-COMA dit op cache-blok nivo doet. In Simple-COMA wordt remote data dus met de granulariteit van cache-blokken overgehaald. Er is in Simple-COMA wel een extra status memory nodig die de status van de cache-blokken in een ge-allocceerde pagina aangeeft.

3. Cache Coherency

1. (4 punten)

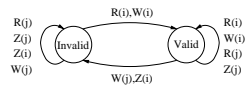
Uitgaande van een bus-gebaseerd multiprocessor systeem met een snoopy bus protocol en een write-invalidate cache coherency protocol, beschrijf de voor- en nadelen van write-through en write-back caches in een dergelijk systeem.

Antwoord: Voordeel van write-through: main memory is altijd up to date. Als er een dirty cache blok is en een andere cache wil deze data hebben, hoeft de dirty data niet eerst naar main memory geschreven te worden. Nadeel van write-through: meer netwerk traffic aangezien elke cache write naar het main memory wordt doorgeschreven. Voordeel van write-back: minder netwerk traffic. Nadeel: zie voordeel van write-through.

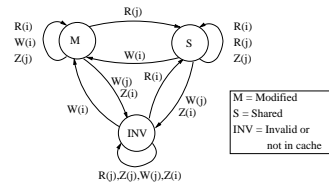
2. (4 punten)

Teken de generieke cache coherency state-transition diagrammen voor een cache-blok in zowel een write-through en een write-back cache van processor i (uitgaande van een write-invalidate coherency protocol). Teken hierin de transities van lees (R) en schrijf (W) opdrachten uitgevoerd door de lokale processor (i) en door een andere processor (j).

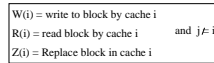
Antwoord:



State-transition graph of write-through cache i



State-transition graph of write-back cache i



3. (2 punten)

Leg uit wat false sharing is en verklaar waarom dit fenomeen een groter probleem is in SVM systemen (waarin data elementen ter grootte van pages worden "ge-shared") dan in VSM systemen (zoals CC-NUMA/COMA systemen).

Antwoord: False sharing: Sommige invalidaties zijn niet nodig voor de correcte programma executie. Voorbeeld:

Processor 1:

Processor 2:

```
while (true) do
  A = A + 1
```

```
while (true) do
  B = B + 1
```

Als A en B in hetzelfde cache block liggen, is er een cache miss per loop iteratie door het "pin-pongen" van invalidaties. Wanneer grote de granulariteit van de data elementen die geshared worden groot is (paginas (SVM) > cache blokken (VSM)) is de kans op false sharing natuurlijk groter.