How these two sequences are made consistent distinguishes the memory behavior in strong and weak models. The quality of a memory model is indicated by hardware/software efficiency, simplicity, usefulness, and bandwidth performance.

Memory Consistency Issues The behavior of a shared-memory system as observed by processors is called a *memory model.* Specification of the memory model answers three fundamental questions: (1) What behavior should a programmer/compiler expect from a shared-memory multiprocessor? (2) How can a definition of the expected behavior guarantee coverage of all contingencies? (3) How must processors and the memory system behave to ensure consistent adherence to the expected behavior of the multiprocessor?

In general, choosing a memory model involves making a compromise between a strong model minimally restricting software and a weak model offering efficient implementation. The use of *partial order* in specifying memory events gives a formal description of special memory behavior.

Primitive memory operations for multiprocessors include *load* (*read*), *store* (*write*), and one or more synchronization operations such as *swap* (atomic *load-store*) or *conditional store.* For simplicity, we consider one representative synchronization operation *swap,* besides the *load* and *store* operations.

Event Orderings On a multiprocessor, concurrent instruction streams (or threads) executing on different processors are *processes.* Each process executes a code segment. The order in which shared memory operations are performed by one process may be used by other processes. *Memory events* correspond to shared-memory accesses. Consistency models specify the order by which the events from one process should be observed by other processes in the machine.

The *event ordering* can be used to declare whether a memory event is legal or illegal, when several processes are accessing a common set of memory locations. A *program order* is the order by which memory accesses occur for the execution of a single process, provided that no program reordering has taken place. Dubois et al. (1986) have defined three primitive memory operations for the purpose of specifying memory consistency models:

(1) A *load* by processor P_i is considered *performed* with respect to processor P_k at a point of time when the issuing of a *store* to the same location by P_k cannot affect the value returned by the *load.*
(2) A *store* by P_i is considered *performed* with respect to P_k at one time when an issued *load* to the same address by P_k returns the value by this *store.*
(3) A *load* is *globally performed* if it is performed with respect to all processors and if the *store* that is the source of the returned value has been performed with respect to all processors.

As illustrated in Fig. 5.19a, a processor can execute instructions out of program order using a compiler to resequence instructions in order to boost performance. A uniprocessor system allows these out-of-sequence executions provided that hardware interlock mechanisms exist to check data and control dependences between instructions.

When a processor in a multiprocessor system executes a concurrent program as illustrated in Fig. 5.19b, local dependence checking is necessary but may not be sufficient to preserve the intended outcome of a concurrent execution.

Maintaining the correctness and predictability of the execution results is rather complex on an MIMD system for the following reasons:

(a) The order in which instructions belonging to different streams are executed is not fixed in a parallel program. If no synchronization among the instruction streams exists, then a large number of different instruction interleavings is possible.
(b) If for performance reasons the order of execution of instructions belonging to the same stream is different from the program order, then an even larger number of instruction interleavings is possible.
(c) If accesses are not atomic with multiple copies of the same data coexisting as in a cache-based system, then different processors can individually observe different interleavings during the same execution. In this case, the total number of possible execution instantiations of a program becomes even larger.

Example 5.8 Event ordering in a three-processor system (Dubois, Scheurich, and Briggs, 1988)

To illustrate the possible ways of interleaving concurrent program executions among multiple processors updating the same memory, we examine the simultaneous and asynchronous executions of three program segments on the three processors in Fig. 5.19c.

The shared variables are initially set as zeros, and we assume a *Print* statement reads both variables indivisibly during the same cycle to avoid confusion. If the outputs of all three processors are concatenated in the order P_1, P_2, and P_3, then the output forms a 6-tuple of binary vectors.

There are $2^6 = 64$ possible output combinations. If all processors execute instructions in their own program orders, then the execution interleaving *a, b, c, d, e, f* is possible, yielding the output 001011. Another interleaving, *a, c, e, b, d, f,* also preserves the program orders and yields the output 111111.

If processors are allowed to execute instructions out of program order, assuming that no data dependences exist among reordered instructions, then the interleaving *b, d, f, e, a, c* is possible, yielding the output 000000.

Out of 6! = 720 possible execution interleavings, 90 preserve the individual program order. From these 90 interleavings not all 6-tuple combinations can result. For example, the outcome 000000 is not possible if processors execute instructions in program order only. As another example, the outcome 011001 is possible if different processors can observe events in different orders, as can be the case with replicated memories.

Atomicity From the above example, multiprocessor memory behavior can be described in three categories:

(1) Program order preserved and uniform observation sequence by all processors.
(2) Out-of-program-order allowed and uniform observation sequence by all processors.
(3) Out-of-program-order allowed and nonuniform sequences observed by different processors.

This behavioral categorization leads to two classes of shared-memory systems for multiprocessors: The first allows *atomic memory accesses*, and the second allows *nonatomic memory accesses*. A shared-memory access is atomic if the memory updates are known to all processors at the same time. Thus a *store* is atomic if the value stored becomes readable to all processors at the same time. Thus a necessary and sufficient condition for an atomic memory to be sequentially consistent is that all memory accesses must be performed to preserve all individual program orders.

In a multiprocessor with nonatomic memory accesses, having individual program orders that conform is not a sufficient condition for sequential consistency. In a cache/network-based multiprocessor, the system can be nonatomic if an invalidation signal does not reach all processors at the same time. Thus a *store* is inherently nonatomic in such an architecture unless special hardware mechanisms are provided to assure atomicity. Only in atomic systems can the ordering of memory events be strongly ordered to make the program order consistent with the memory-access order.

With a nonatomic memory system, the multiprocessor cannot be strongly ordered. Thus weak ordering is very much desired in a multiprocessor with nonatomic memory accesses. The above discussions lead to the division between strong and weak consistency models to be described in the next two subsections.

5.4.2 Sequential Consistency Model

The *sequential consistency* (SC) memory model is widely understood among multiprocessor designers. In this model, the *loads, stores,* and *swaps* of all processors appear to execute serially in a single global memory order that conforms to the individual program orders of the processors, as illustrated in Fig. 5.20. Two definitions of SC model are given below.

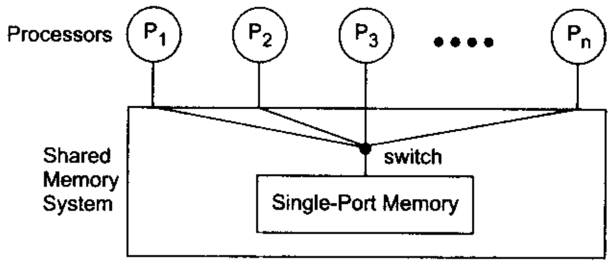

Fig. 5.20 Sequential consistency memory model (Courtesy of Sindhu, Frailong, and Cekleov; reprinted with permission from *Scalable Shared-Memory Multiprocessors,* Kluwer Academic Publishers, 1992)

Lamport's Definition Lamport (1979) defined *sequential consistency* as follows: A multiprocessor system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Dubois, Scheurich, and Briggs (1986) have provided the following two sufficient conditions to achieve sequential consistency in shared-memory access:

(a) Before a *load* is allowed to perform with respect to any other processor, all previous *load* accesses must be globally performed and all previous *store* accesses must be performed with respect to all processors.

(b) Before a *store* is allowed to perform with respect to any other processor, all previous *load* accesses must be globally performed and all previous *store* accesses must be performed with respect to all processors.

Maintaining the correctness and predictability of the execution results is rather complex on an MIMD system for the following reasons:

(a) The order in which instructions belonging to different streams are executed is not fixed in a parallel program. If no synchronization among the instruction streams exists, then a large number of different instruction interleavings is possible.

(b) If for performance reasons the order of execution of instructions belonging to the same stream is different from the program order, then an even larger number of instruction interleavings is possible.

(c) If accesses are not atomic with multiple copies of the same data coexisting as in a cache-based system, then different processors can individually observe different interleavings during the same execution. In this case, the total number of possible execution instantiations of a program becomes even larger.

Example 5.8 Event ordering in a three-processor system (Dubois, Scheurich, and Briggs, 1988)

To illustrate the possible ways of interleaving concurrent program executions among multiple processors updating the same memory, we examine the simultaneous and asynchronous executions of three program segments on the three processors in Fig. 5.19c.

The shared variables are initially set as zeros, and we assume a *Print* statement reads both variables indivisibly during the same cycle to avoid confusion. If the outputs of all three processors are concatenated in the order P_1, P_2, and P_3, then the output forms a 6-tuple of binary vectors.

There are $2^6 = 64$ possible output combinations. If all processors execute instructions in their own program orders, then the execution interleaving *a, b, c, d, e, f* is possible, yielding the output 001011. Another interleaving, *a, c, e, b, d, f,* also preserves the program orders and yields the output 111111.

If processors are allowed to execute instructions out of program order, assuming that no data dependences exist among reordered instructions, then the interleaving *b, d, f, e, a, c* is possible, yielding the output 000000.

Out of 6! = 720 possible execution interleavings, 90 preserve the individual program order. From these 90 interleavings not all 6-tuple combinations can result. For example, the outcome 000000 is not possible if processors execute instructions in program order only. As another example, the outcome 011001 is possible if different processors can observe events in different orders, as can be the case with replicated memories.

Atomicity From the above example, multiprocessor memory behavior can be described in three categories:

(1) Program order preserved and uniform observation sequence by all processors.

(2) Out-of-program-order allowed and uniform observation sequence by all processors.

(3) Out-of-program-order allowed and nonuniform sequences observed by different processors.

This behavioral categorization leads to two classes of shared-memory systems for multiprocessors: The first allows *atomic memory accesses*, and the second allows *nonatomic memory accesses*. A shared-memory access is atomic if the memory updates are known to all processors at the same time. Thus a *store* is atomic if the value stored becomes readable to all processors at the same time. Thus a necessary and sufficient

degrees of weak consistency. In this section, we describe the *weak consistency* model introduced by Dubois et al. (1986) and a TSO model introduced with the SPARC architecture.

The DSB Model Dubois, Scheurich, and Briggs (1986) have derived a weak consistency memory model by relating memory request ordering to synchronization points in the program. We call this the DSB model specified by the following three conditions:

(1) All previous *synchronization* accesses must be performed, before a *load* or a *store* access is allowed to perform with respect to any other processor.

(2) All previous *load* and *store* accesses must be performed, before a *synchronization* access is allowed to perform with respect to any other processor

(3) *Synchronization* accesses are sequentially consistent with respect to one another.

These conditions provide a weak ordering of memory-access events in a multiprocessor. The dependence conditions on shared variables are weaker in such a system because they are only limited to hardware-recognized synchronizing variables. Buffering is allowed in *write buffers* except for operations on hardware-recognized synchronizing variables. Buffering memory accesses in multiprocessors can enhance the shared memory performance.

With different restrictions on the memory-access ordering, many different weak memory models can be similarly defined. The following is another weak consistency model, called the TSO (total store order), developed by the SPARC architecture group at Sun Microsystems.

Example 5.9 The TSO weak consistency model used in SPARC architecture (Sun Microsystems, Inc., 1990 and Sindhu et al., 1992)

Figure 5.21 shows the weak consistency TSO model developed by Sun Microsystems' SPARC architecture group (1990). Sindhu et al. described that the *stores* and *swaps* issued by a processor are placed in a dedicated store buffer for the processor, which is operated as first-in-first-out. Thus the order in which memory executes these operations for a given processor is the same as the order in which the processor issued them (in program order).

The memory order corresponds to the order in which the switch is thrown from one processor to another. This was described by Sindhu et al. as follows: A *load* by a processor first checks its store buffer to see if it contains a *store* to the same location. If it does, then the *load* returns the value of the most recent such *store*. Otherwise, the *load* goes directly to memory. Since not all *loads* go to memory immediately, *loads* in general do not appear in memory order. A processor is logically blocked from issuing further operations until the *load* returns a value. A *swap* behaves like a *load* and a *store*. It is placed in the store buffer like a *store*, and it blocks the processor like a *load*. In other words, the *swap* blocks until the store buffer is empty and then proceeds to the memory.

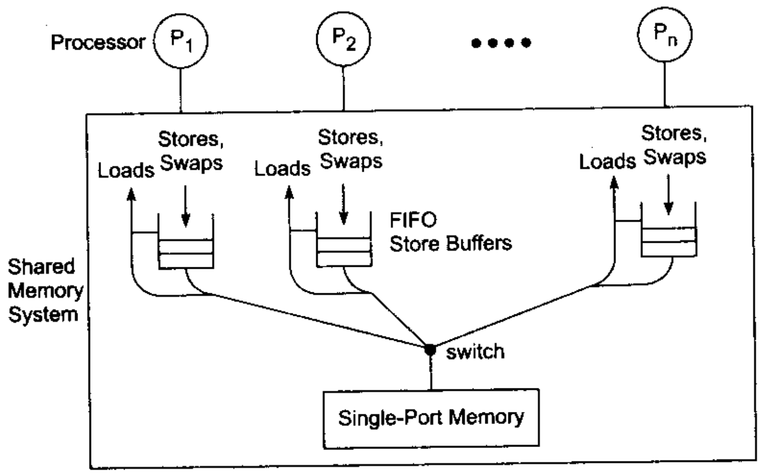

Fig. 5.21 The TSO Weak consistency memory model (Courtesy of Sindhu, Frailong, and Cekleov; reprinted with permission from *Scalable Shared-Memory Multiprocessors*, Kluwer Academic Publishers, 1992)

A TSO Formal Specification Sindhu, Frailong, and Cekleov (1992) have specified the TSO weak consistency model with six behavioral axioms. Only an intuitive description of their axioms is abstracted below:

(1) A *load* access is always returned with the latest *store* to the same memory location issued by any processor in the system.
(2) The memory order is a total binary relation over all pairs of *store* operations.
(3) If two *stores* appear in a particular program order, then they must also appear in the same memory order.
(4) If a memory operation follows a *load* in program order, then it must also follow the *load* in memory order.
(5) A *swap* operation is atomic with respect to other *stores*. No other *store* can interleave between the *load* and *store* parts of a *swap*.
(6) All *stores* and *swaps* must eventually terminate.

Note that the above axioms (5) and (6) are identical to axioms (4) and (5) for the sequential consistency model. Axiom (1) covers the effects of both local and remote processors. Both axioms (2) and (3) are weakened from the corresponding axioms (2) and (3) for the sequential consistency model. Axiom (4) states that the *load* operations do not have to be weakened, as far as ordering is concerned. For a formal axiomatic specification, the reader is referred to the original paper by Sindhu et al. (1992).

Comparison of Memory Models In summary, the weak consistency model may offer better performance than the sequential consistency model at the expense of more complex hardware/software support and more programmer awareness of the imposed restrictions. The relative merits of the strong and weak memory models have been subjects of debate in the multiprocessor research community.

The DSB and the TSO are two different weak-consistency memory models. The DSB model is weakened by enforcing sequential consistency at synchronization points. The TSO model is weakened by treating *reads*, *stores*, and *swaps* differently using FIFO store buffers. The TSO model has been implemented in some SPARC architectures, while the DSB model has not been implemented in real systems yet.

Sindhu et al. (1992) have identified four system-level issues which also affect the choice of memory model. First, they suggest that one should consider extending the memory model from the processor level to the process level. To do this, one must maintain a process switch sequence. The second issue is the incorporation of I/O locations into the memory model. I/O operations may introduce even more side effects in addition to the normal semantics of *loads* and *stores*.

The third issue is code modification. In the SPARC architecture, synchronization of code modification is accomplished through the use of a *flush* instruction defined in the TSO model. Finally, they feel that the memory model should address the issue of how to incorporate pipelined processors and processors with noncoherent internal caches. The interested reader is referred to their original paper and the SPARC Architecture Manual for details of these issues.

Strong memory ordering introduces unnecessary processor/cache waiting time and reduces the amount of concurrency. Weak consistency has the potential to eliminate these shortcomings. Besides sequential consistency, DSB and TSO weak memory models, other memory consistency models, such as *processor consistency* and *release consistency*, will be treated in Chapter 9.

Summary

In this chapter we studied the functions and technical requirements of the system bus and cache memory, and also discussed some basic issues related to shared main memory in a multiprocessor system. We saw that the single system bus has performance limitations as processors become faster and the number of processors in the system increases. With this background, we shall study other system interconnect strategies in latter chapters.

The earliest multiprocessor systems were built around a single system bus. We studied the basic system requirements of the bus, i.e. addressing, timing, arbitration, transaction modes, interrupts, and so on. As one specific example of a bus specification, we looked at Futurebus+. However, bus-based communication of the earlier multiprocessors, usually based on a single backplane, has limited scalability.

With rapidly increasing computing power, it became clear that communication between different sub-systems of a computer system—and also between computer systems—is as important as the storage and processing of data. As demands on the system interconnect grew rapidly, performance limitations in using a single bus such as Futurebus+ became obvious. Scalable Coherent Interconnect (SCI) and InfiniBand, which grew out of the unsuccessful Futurebus+ effort, employ point-to-point links and packet-switching, and can therefore support highly scalable systems.

Cache memories are provided between the processor and main memory to bridge the huge speed mismatch between these two sub-systems. Over the last two or three decades, this speed mismatch has grown larger, because processor speeds have risen much faster than main memory speeds. Addressing models, direct mapped *versus* set associative cache, block size, and other relevant cache performance

issues were discussed. Multiple levels of cache are often employed; based on their design goals, different models of the same processor family may employ different multi-level cache designs.

Interleaving of memory modules is a technique for achieving higher aggregate memory bandwidth in support of higher system performance. Different schemes for memory interleaving have been considered, along with related performance issues. Memory allocation schemes such as paging and swapping were also considered.

For multiprocessor systems with shared memory, apart from the strict sequential consistency model, weaker consistency models are also considered—the aim being to achieve a greater degree of parallelism, and thereby higher system performance. Basic concepts of atomicity of memory accesses and event ordering are used to define memory consistency models; two specific such models, DSB and TSO, were discussed.

Exercises

Problem 5.1 This is an illustrative example of a design specification of a backplane bus for a shared-memory multiprocessor with 4 processor boards and 16 memory boards under the following assumptions:

- Bus clock rate = 200 MHz.
- Memory word length = 64 bits; processors always request data in blocks of four words.
- Memory access time = 100 ns.
- Shared address space = 2^{40} words.
- Maximum number of signal lines available on the backplane is 96.
- Synchronous timing protocol.
- Neglect buffer and propagation delays.

Specify the following in your design of the bus system:

(a) Maximum bus bandwidth.
(b) Effective bus bandwidth (worst case).
(c) Arbitration scheme.
(d) Name and functionality of each of the signal lines.
(e) Number of slots required on the backplane.

Justify any additional assumptions you need to make.

Problem 5.2 Describe the daisy-chaining (Fig. 5.4) and the *distributed arbiter* (Fig. 5.5b) for bus arbitration in a multiprocessor system. State the advantages and shortcomings of each case from both the implementational and operational points of view.

Problem 5.3 Read the paper by Mudge et al. (1987) on multiple-bus systems and solve the following problems:

(a) Find the maximum bandwidth for a multiprocessor system using b buses, where $b > m$ and m is the number of memory modules and the system has n processors.
(b) Prove that $BW_b < np$, where $p > 0$ is the probability that an arbitrary processor will generate a request to access the shared memory at the start of a memory cycle.

Problem 5.4 Estimate the effective MIPS rate of a bus-connected multiprocessor system under the following assumptions. The system has 16 processors, each connected to an on-board private cache which is connected to a common bus. Globally shared memory is also connected to the bus. The private cache and the shared memory form a two-level access hierarchy.

Each processor is rated 500 MIPS if a 100% cache hit ratio is assumed. On the average, each instruction needs 0.2 memory access. The read access and write access are assumed equally probable.

For a crude approximation, consider only the penalty caused by shared-memory access and ignore all other overheads. The cache is targeted to maintain a hit ratio of 0.95. A cache access on a read-hit takes 2 ns; that on a write-hit takes 4 ns with a write-back scheme, and with a write-through scheme it needs 100 ns.

When a cache block is to be replaced, the probability that it is dirty is estimated as 0.1. An average block transfer time between the cache and shared memory via the bus is 100 ns.

(a) Derive the effective memory-access times per instruction for the write-through and write-back caches separately.

(b) Calculate the effective MIPS rate for each processor. Determine an upper bound on the effective MIPS rate of the 16-processor system. Discuss why the upper bound cannot be achieved by considering the memory penalty alone.

Problem 5.5 Explain the following terms associated with cache and memory architectures.

(a) Low-order memory interleaving.

(b) Physical address cache versus virtual address cache.

(c) Atomic versus nonatomic memory accesses.

(d) Memory bandwidth and fault tolerance.

Problem 5.6 Explain the following terms associated with cache design:

(a) Write-through versus write-back caches.

(b) Cacheable versus noncacheable data.

(c) Private caches versus shared caches.

(d) Cache flushing policies.

(e) Factors affecting cache hit ratios.

Problem 5.7 Consider the simultaneous execution of the three programs on the three processors shown in Fig. 5.19c. Answer the following questions with reasoning or supported by computer simulation results:

(a) List the 90 execution interleaving orders of the six instructions {*a, b, c, d, e, f*} which will preserve the individual program orders. The corresponding output patterns (6-tuples) should be listed accordingly.

(b) Can all 6-tuple combinations be generated out of the 720 non-program-order interleavings? Justify the answer with reasoning and examples.

(c) We have assumed atomic memory access in this example. Explain why the output 011001 is not possible in an atomic memory multiprocessor system if individual program orders are preserved.

(d) Suppose nonatomic memory access is allowed in the above multiprocessor. For example, an invalidation does not reach all private caches at the same time. Prove that 011001 is possible even if all instructions were executed in program order but other processors did not observe them in program order.

Problem 5.8 The main memory of a computer is organized as 64 blocks, with a block size of eight words. The cache has eight block frames. In parts (a) through (d), show the mappings from the numbered blocks in main memory to the block frames in the cache. Draw all lines showing the mappings as clearly as possible.

(a) Show the direct mapping and the address bits that identify the tag field, the block number, and the word number.

(b) Show the fully associative mapping and the address bits that identify the tag field and the word number.

(c) Show the two-way set-associative mapping and the address bits that identify the tag field, the set number, and the word number.

(d) Show the sector mapping with four blocks per sector and the address bits that identify the sector number, the block number, and the word number.

Problem 5.9 Consider a cache (M_1) and memory (M_2) hierarchy with the following characteristics:

M_1: 64K words, 5 ns access time

M_2: 4M words, 40 ns access time

Assume eight-word cache blocks and a set size of 256 words with set-associative mapping.

(a) Show the mapping between M_2 and M_1.

(b) Calculate the effective memory-access time with a cache hit ratio of $h = 0.95$.

Problem 5.10 Consider a main memory consisting of four memory modules with 256 words per module. Assume 16 words in each cache block. The cache has a total capacity of 256 words. Set-associative mapping is used to allocate cache blocks to block frames. The cache is divided into four sets.

(a) Show the address assignment for all 1024 words in a four-way low-order interleaved organization of the main memory.

(b) How many blocks are there in the main memory? How many block frames are there in the cache?

(c) Explain the bit fields needed for addressing each word in the two-level memory system.

(d) Show the mapping from the blocks in the main memory to the sets in the cache and explain how to use the tag field to locate a block frame within each set.

Problem 5.11

(a) A uniprocessor system uses separate instruction and data caches with hit ratios h_i and h_d, respectively. The access time from the processor to either cache is c clock cycles, and the block transfer time between the caches and main memory is b clock cycles.

Among all memory references made by the CPU, f_i is the percentage of references to instructions. Among blocks replaced in the data cache, f_{dir} is the percentage of *dirty* blocks. (*Dirty* means that the cache copy is different from the memory copy.)

Assuming a write-back policy, determine the effective memory-access time in terms of h_i, h_d, c, b, f_i, and f_{dir} for this memory system.

(b) The processor memory system described in part (a) is used to construct a bus-based shared-memory multiprocessor. Assume that the hit ratio and access times remain the same as in part (a). However, the effective memory-access time will be different because every processor must now handle cache invalidation in addition to reads and writes.

Let f_{inv} be the fraction of data references that cause invalidation signals to be sent to other caches. The processor sending the invalidation signal requires i clock cycles to complete the invalidation operation. Other processors are not involved in the invalidation process. Assuming a write-back policy again, determine the effective memory-access time for this multiprocessor.

Problem 5.12 A computer system has a 128-byte cache. It uses four-way set-ssociative mapping with 8 bytes in each block. The physical address size is 32 bits, and the smallest addressable unit is 1 byte.

(a) Draw a diagram showing the organization of the cache and indicating how physical addresses are related to cache addresses.

(b) To what block frames of the cache can the address $000010AF_{16}$ be assigned?

(c) If the addresses $000010AF_{16}$ and $FFFF7Axy_{16}$ can be simultaneously assigned to the same cache set, what values can the address digits x and y have?

Problem 5.13 Consider a shared-memory multiprocessor system with p processors. Let m be the average number of global memory references per instruction execution on a typical processor.

Let t be the average access time to the shared memory and x be the MIPS rate of a uniprocessor using local memory. Consider the execution of n instructions on each processor of the multiprocessor.

(a) Determine the effective MIPS rate of the multiprocessor in terms of the parameters m, t, x, n, and p.

(b) Suppose a multiprocessor has $p = 32$ RISC processors, $m = 0.4$, and $t = 0.01$ μs. What is the MIPS rate of each processor (i.e. x = ?) needed to achieve a multiprocessor performance of 5600 MIPS effectively?

(c) Suppose $p = 32$ CISC processors with $x = 200$ MIPS each are used in the above multiprocessor system with $m = 1.6$ and $t = 0.01$ μs. What will be the effective MIPS rate?

Problem 5.14 Consider a RISC-based shared-memory multiprocessor with p processors, each having its own instruction cache and data cache. The peak performance rating of each processor (assuming a 100% hit ratio in both caches) is x MIPS. You are required to derive a performance formula, taking into account cache misses, shared-memory accesses, and synchronization overhead.

Assume that on the average α percent of the instructions executed are for synchronization purpose, and the penalty for each synchronization operation is an additional t_s μs. The number of memory accesses per instruction is m. Among all memory references made by the CPU, f_i is the percentage of references to instructions. Assume that the instruction cache and data cache have hit ratios h_i and h_d, respectively, after a long period of program tracing on the machine. On cache misses, instructions and data are accessed from the shared memory with an average access time t_m μs.

(a) Derive an expression for approximating the effective MIPS rate of this multiprocessor in terms of p, x, m, f_i, h_i, h_d, t_m, α, and t_s. Note that f_i, h_i, h_d, and α are all fractions and t_m and t_s are measured in μs. Ignore the cache-access time and other system overheads in your derivation.

(b) Suppose $m = 0.4$, $f_i = 0.5$, $h_i = 0.95$, $h_d = 0.8$, $\alpha = 0.02$, $x = 500$, $t_m = 0.05$ μs, and $t_s = 1$ μs. Determine the minimum number of processors needed in the above multiprocessor system in order to achieve an effective MIPS rate of 2000.

(c) Suppose the total cost of all the caches and shared-memory is upper-bounded by \$25,000. The cache memory costs \$1.25/Kbyte, and the shared memory costs \$0.1/Kbyte. With $p = 16$ processors, each having an instruction cache of capacity $S_i = 32$ Kbytes and a data cache of capacity $S_d = 64$ Kbytes, what is the maximum shared-memory capacity C_m (in Mbytes) that can be acquired within the budget limit?

Problem 5.15 Consider the following three interleaved memory designs for a main memory system with 16 memory modules. Each module is assumed to have a capacity of 1 Mbyte. The machine is byte-addressable.

Design 1: 16-way interleaving with one memory bank.
Design 2: 8-way interleaving with two memory banks.
Design 3: 4-way interleaving with four memory banks.

(a) Specify the address formats for each of the above memory organizations.

(b) Determine the maximum memory bandwidth obtained if only one memory module fails in each of the above memory organizations.

(c) Comment on the relative merits of the three interleaved memory organizations.

Problem 5.16 Consider a memory system for the erstwhile Cray 1 computer. There are $m = 16$ interleaved modules. The access time of a module is $t_a = 50$ ns and the memory cycle time is $t_c = 12.5$ ns. We know that for this memory system the maximum memory bandwidth of 80M words per second is achieved for vector loads/stores except when the stride is a multiple of 16 (bandwidth: 20M words per second) or a multiple of 8 (but not 16) (bandwidth: 40M words per second).

(a) Find the bandwidth for all strides for similar systems but with the following parameters: $t_c = 12.5$ ns, $t_a = 50$ ns, $m = 17$.

(b) Repeat part (a) for the following parameters: $t_c = 12.5$ ns, $t_a = 50$ ns, $m = 8$.

Problem 5.17 Consider the concurrent execution of two programs by two processors with a shared memory. Assume that *A, B, C, D* are initialized to 0 and that a *Print* statement prints both arguments indivisibly at the same cycle. The output forms a 4-tuple as either *ADBC* or *BCAD*.

P_0:	P_1:
a. $A = 1$	d. $C = 1$
b. $B = 1$	e. $D = 1$
c. Print *A, D*	f. Print *B, C*

(a) List all execution interleaving orders of six statements which will preserve the individual program order.

(b) Assume program orders are preserved and all memory accesses are atomic; i.e., a store by one processor is immediately seen by all the remaining processors. List all the possible 4-tuple output combinations.

(c) Assume program orders are preserved but memory accesses are nonatomic; i.e., a store by one processor may be buffered so that some other processors may not immediately observe the update. List all possible 4-tuple output combinations.

Problem 5.18 Compare the relative merits of the four cache memory organizations:

(1) Direct-mapping cache
(2) Fully associative cache
(3) Set-associative cache
(4) Sector mapping cache

Answer the following questions with reasoning:

(a) In terms of hardware complexity and implementation cost, rank the four cache organizations with justification.

(b) With respect to flexibility in implementing block replacement algorithms, rank the four cache organizations and justify the ranking order.

(c) With each cache organization, explain the effects of block mapping policies on the hit ratio issues.

(d) Explain the effects of block size, set number, associativity, and cache size on the performance of a set-associative cache organization.

Problem 5.19 Explain the following terms associated with memory management:

(a) The role of a memory manager in an OS kernel.

(b) Preemptive versus nonpreemptive memory allocation policies.

(c) Swapping memory system and examples.

(d) Demand paging memory system and examples.

(e) Hybrid memory system and examples.

Problem 5.20 Compare the memory-access constraints in the following memory consistency models:

(a) Determine the similarities and subtle differences among the conditions on sequential consistency imposed by Lamport (1979), by Dubois et al. (1986), and by Sindhu et al. (1992), respectively.

(b) Repeat question (a) between the DSB model and the TSO model for weak consistency memory systems.

(c) A PSO (partial store order) model for weak consistency has been refined from the TSO model. Study the PSO specification in the paper by Sindhu et al. (1992) and compare the relative merits between the TSO and the PSO memory models.