Presentation 8

Shared Memory Architecture

מודל זכרון משותף הוא צורת העבודה בתחנת עבודה בעלת מעבד רב-ליבות, סוג המחשב הנפוץ ביותר היום. המודל הזה מתאים למקבילות ברמת הפתיל (TLP) כהמשך ישיר של שיטות ה- ILP במעבד superscalar. מספר ליבות (בדרך כלל 2 עד 8) מחלקות ביניהם פתילים שמטפלים בחלק ממבנה נתונים אחד, על פי חלוקה פונקציונלית או חולקת נתונים. כיוון שכל הפתילים פועלים על אותה קבוצת נתונים שיושבת בזכרון משותף, חייבים למנוע סיכונים (data hazards) עם סנכרון מתאים.

Slide 2 – 4 Shared Memory Multiprocessor

In **shared memory**, the global memory space A may be physically partitioned into M blocks (0 to M – 1) but the full memory appears unified through the switching fabric. Each of the N processors (0 to N – 1) can access the full address space (address 0 to A – 1) via the internal network. Threads running on the N CPUs communicate among themselves by writing and reading shared addresses in memory. For many activities, shared memory requires synchronization of memory accesses to prevent data hazards.

Memory access to shared memory systems can be subdivided into three types. In uniform memory access (UMA), the whole memory is accessible by all processors, with uniform access time from any CPU to any memory location. In non-uniform memory access (NUMA), each CPU is physically attached to one part of shared memory. Access time to a given memory address depends on the distance to memory from the processor. In cache-only memory architecture (COMA), each processor has part of shared memory as private cache, without main memory. COMA requires that data be migrated from processor cache to processor cache for sharing.

Because of the rapidly decreasing cost of multicore systems, **symmetric multiprocessor** (SMT) is the most popular example of the UMA model. An SMT consists of a number of processors of an equivalent type with a symmetric relationship to main memory, as shown on slide 4. Each core is basically equivalent, and each has the same relationship and access time to main memory. Generally, each core has a private L1 cache, while the cores in a single package share a common L2 cache. In a certain deviation from pure SMT, the sharing of data among cores within one package is performed in L2 and is simpler than sharing between packages.

Slide 5 – 7 Open MP for Shared Memory Systems

Open MP is an API (application program interface) for multiprocessing designed to simplify thread-based parallelization for shared memory programming in C/C++ and Fortran. OMP provides directives for explicit parallelization, extending familiar constructs to a Fork-Join Model. Programs initiate as a single thread called the Master thread (also called the consumer thread) and execute sequentially until a parallel construct is encountered. The Master thread creates (forks or spawns) a team of producer threads that execute in parallel on multiple CPU cores. When the producer threads complete, a Join operation synchronizes and terminates the producer team, leaving the Master thread to continue. Forks can be nested within parallel sections.

The general code structure is shown on slide 6. The statement

#pragma omp parallel private(..., ..) shared(...) { ... }

defines a parallel team of threads with private (per thread) variables and shared variables. Slide 7 shows a partial list of options, including reduction (a joining of private producer results to a single Master result.

Slide 8 "Hello Worlds" Program

The "Hello Worlds" program for OMP is shown on slide 8. The **pragma omp parallel** defines the private variable tid used by each thread to get its thread ID using the OMP system call omp_get_thread_num(). Each thread displays "Hello World from thread = ID" for its thread ID number. The master thread (with tid = 0) obtains the total number of threads using the OMP system call nthreads = omp_get_num_threads() and displays the number. The Fork then terminates in a Join and the program returns to sequential execution.

Slide 9 – 11 Parallel For Directive

Within the **pragma omp parallel** block, the **pragma omp for** performs data decomposition of a standard C for loop. Slide 10 shows a program that initializes arrays a[N] and b[N] with N = 1000, and then performs

```
for (i=0; i < N; i++)    c[i] = a[i] + b[i];
```

in chunks of 100 over (1000 values / 100 values per thread) = 10 threads.

The **pragma omp parallel** directive defines a[], b[], c[], and chunk as shared, and defines a private value of i for each thread. The **pragma omp for** defines producer threads that are dynamic — when a thread completes, it is closed without waiting to synchronize. A new thread can run on the same CPU to perform another chunk of the work. Slide 11 illustrates data decomposition in graphic form.

Slide 12 Sections Directive

Within the **pragma omp parallel** block, the **pragma omp sections** performs functional decomposition of instructions within a standard C for loop. Slide 12 shows a program that initializes arrays a[N] and b[N] with N = 1000, and then performs

```
for (i=0; i < N; i++)    c[i] = a[i] + b[i];
for (i=0; i < N; i++)    d[i] = a[i] * b[i];
```

in two parallel threads. The **pragma omp parallel** defines a[], b[], c[N], d[N] as shared and defines a private value of i for each thread. The **pragma omp sections** opens a team of producer threads, and the **pragma omp section** defines the activity of each thread.

Slide 13 – 15 Race Conditions

A race condition is a data hazard caused by parallel access to shared memory. For example, the program fragment

```
#pragma omp parallel shared(x) num_threads(2) {  x = x + 1; }
```

should create two threads to independent increment x by 1 and provide the result x ← x + 2. But as shown on slide 13, if the threads execute as interleaved, the sequence of data reads and writes to the shared variable x produces the final result x ← x + 1. To prevent race conditions, synchronization methods can protect threads from conflict with other threads.

Four synchronizations defined in OMP are:

#pragma omp master	Only the master thread (thread 0) performs the following block.
#pragma omp critical	Only one thread can execute the following block at a time. Other threads wait for this thread to leave the critical section before entering.
#pragma omp barrier	Each thread reaching the barrier waits until all threads reach the barrier.
#pragma omp atomic	A memory update in the next statement must be completed atomically. This is a mini-critical section for memory writes.

Slide 15 shows how the race condition in the previous example can be corrected by defining the instruction x = x + 1 as critical. This way, the threads are made to run sequentially. The first thread to begin will complete before other threads read x.

Slide 16 Reduction

The reduction directive joins a list of private variables into a final result for the master thread. For example, the directive **reduction(+:sum)** adds the private copies of the variable sum from each thread into the master thread's copy. The example on slide 16 shows a team of producer threads, each one evaluating part of the scalar product of vectors a[] and b[]. The reduction combines the partial results into the full scalar product.

Slide 17 – 19 Bus-Based Symmetric Multiprocessor (SMP)

Most symmetric multiprocessor (SMP) systems are based on a PC workstation architecture in which processors are interconnected with main memory through a bus. Typically, one core is selected at system initialization time as the bootstrap processor (BSP) that performs system startup and runs OS functions. The other cores are designated application processors (AP), which remain in the halt state until the OS allocates application threads.

Two possible methods for sharing of cache between cores in a processor package are shown on slide 18. In the Pentium D package (Intel's first dual core x86 CPU), each core has private L1 and L2 cache. Threads running on the cores must update main memory to share data between them. In more recent x86 CPUs, each core has a private L1 cache and a common L2 cache, allowing memory sharing between threads without updating main memory. Slide 19 shows the organization of cache in the Pentium M (the last single-core upgrade of the Pentium 4) and in the Core Duo (dual core CPU). In the Pentium M, a cache manager (control unit) mediates between the external bus (to main memory), L2 cache, prefetch, and the core. The Core Duo introduces a fairness unit to insure that neither core suffers data starvation. The fairness unit connects both cores to the cache manager (control unit), which is now split between bus access functions and L2 access functions.

Slide 20 – 23 Communication Overhead in SMP

Performing the sum c[] = a[] + b[] sequentially requires 4 load-add-store operations. On a 64-bit CPU, each array is 4 elements × 8 bytes per element = 32 bytes, so that an optimizing compiler places the three arrays in a single 128-byte cache block, for a maximum of one cache miss.

Slide 20 shows an OMP routine to add the arrays in parallel. Running in parallel, each of 4 cores must perform a cache update to load the cache block (sequentially, over the serial bus), perform 1 load-add-store operation, and then update the main memory. Since all the cores are writing to the same cache block, this operation must be performed atomically. Cache updates take much more time than arithmetic operations, so that the parallel version will suffer a much longer run time than the sequential version. This is an extreme example of a common issue in SMP.

Another issue is the time to implement the reduce operation. The left side of slide 21 shows the sequential evaluation of the scalar product

$$\sum_{\mathtt{i=0}}^{\mathtt{3}} \mathbf{a}[\mathbf{i}] * \mathbf{b}[\mathbf{i}] .$$

Ignoring cache overhead, the sequential version executes 4 iterations × 4 instructions per iteration + 1 store = 17 instructions. The parallel code spawns 4 threads to perform 1 iteration per thread. Then, the reduce operation performs 2 parallel partial sums (another iteration) and then a final iteration over the two partial sums. The 3 iterations perform 12 instructions, and the speedup is 17/12 = 1.42. Inverting Amdahl's equation

$$1.42 = \frac{1}{1 - F_P + \frac{F_P}{4}} \qquad \Rightarrow \qquad F_P = \frac{1 - \frac{1}{1.42}}{1 - \frac{1}{4}} = 0.394$$

the overhead is equivalent to 39% of the program running in parallel, or 61% of the program running sequentially.

An estimate of the capacity of an SMP system can be found by requiring that interconnection network capacity be greater than the volume of exchanged data. The capacity can be written as

$$\text{Interconnection capacity} = R_n \times W$$

R_n = transfer rate (transfer cycles per second)

W = transfer width (bytes per transfer)

and the volume of data can be written as

$$\text{Exchange demand volume} = N \times D_x \times M$$

N = CPUs

D_x = average data access rate (bytes per second) per access

M = inter-CPU access rate = cache miss rate

The total volume of data transferred will be less than the interconnection capacity if the total number of CPUs satisfies

$$N \le \frac{R_n W}{D_x M}$$

For example, on a 1 GHz CPU with PCI express at 16 GB per second,

R_n = 1 GHz

W = 16 bytes per transfer

M = 0.1 (communication dominated miss rate)

D_x = 4 bytes per load × 0.25 loads per instruction × 10^9 instructions per second

The limit is $N \leq \frac{10^9 \times 16}{10^9 \times 0.1} = 160$ CPUs

Slide 24 – 26 Performance

Slide 24 shows a comparison of the Core Duo dual core and Pentium M single core processors on the (non-threaded) SPEC benchmarks Cint and Cfp. Since the code is not multi-threaded, this graph shows how one core of a Core Duo CPU compares with the single core of the Pentium M. On integer programs, the two CPUs are about equal in performance, while the Core Duo shows some speedup on floating point programs because of improvements to the FPU and cache.

Slide 25 shows a comparison of the CPUs on multi-threaded benchmarks. On some tests, the Core Duo is twice as fast as the Pentium M, showing the effectiveness of parallel execution.

Slide 26 compares the CPUs using data decomposition and functional decomposition. Generally, data decomposition provides a higher speedup, because it divides loop length. Functional decomposition divides some of the work among cores, but the loop length is unchanged and most of the work remains sequential. The results on functional decomposition depend strongly on the operating mode of the CPU. Performance optimization is the fastest CPU mode, used when the computer is connected to external power. Power optimization is a power-saving mode, used when the computer is running on battery power. In this mode, many CPU functions are shut down when not needed to save power. Under these conditions, functional decomposition is less effective than data decomposition, which finishes quickly permitting the CPU to be shut down.

Slide 27 – 28 Data Hazards in Shared Memory SMP

Preventing data hazards in shared memory systems requires analysis of three levels of multiprocessor behavior. At the **highest level**, programmers must use synchronization directives to avoid race conditions when performing inherently sequential operations in concurrent programs. As seen in OMP, synchronization permits atomic execution of individual read/write operations or whole program blocks defined as critical sections. At the **system level**, SMP systems must enforce cache consistency protocols to insure that multiple processors writing shared copies of data maintain valid copies of data values. At the **hardware/software** boundary, algorithms must take the complexity of real hardware operations into account. Although instruction-level memory semantics appear atomic, algorithms must be designed to function correctly if the pipeline level implementation does not enforce atomic reads and writes to memory.

The general approach to hazards is to choose an operational definition of **program consistency**, typically correct program execution under interleaving. Enforcement of program consistency enables unambiguous program validation and detection of data hazards.

Strict consistency holds when all memory operations are performed in the order intended by programmer, which is possible to implement only on a single processor system. **Sequential consistency** (Lamport 1979) refers to any clear, consistent, repeatable definition of execution order. A consistent concurrent program will produce an identical result in any execution for which each processor executes one instruction in the specified sequential order, according to

the order specified by program. Any well-defined interleaving, such as round robin, satisfies the sequential ordering.

The programmer must assume that unsynchronized threads execute interleaved, that cache enforces write-order consistency (as specified by the interleaving and program order), and that hardware enforces the consistent ordering of reads and writes.

Slide 29 Implementing Critical Section with Semaphore

Slide 15 showed how a race condition can be corrected in OMP by defining an instruction block as critical, so that the blocks are forced to run sequentially. In order to enforce criticality, some method of signaling must be employed.

One obvious possibility is the use of **semaphores** provided by the OS. A semaphore is an unsigned number s with two operations, P(s) and V(s). The operation V(s) increments the value of s by 1. If s > 0, then P(s) decrements s by 1, and otherwise P(s) waits synchronously (blocks subsequent instructions from executing) until some other process increments s.

A **mutex** (mutual exclusion) is a binary semaphore initialized to 1 and saturated at 1 (that is, if s=1, then V(s) = 1). As shown on slide 20, a mutex can be used to define an instruction block as critical by placing P(s) before the block and V(s) after the block. As soon as any CPU enters the block, all other CPUs find s = 0 and are prevented from continuing into the block. When that CPU completes the block it sets s = 1 so that another CPU checking s will be able to run.

It is not practical to implement the mutex using OS semaphores, because it would require disabling all interrupts across the system between P and V.

Slide 30 – 32 Shared Variable Lock

Another method of enforcing criticality is the use of shared variables as a form of signaling. A simple appears on slide 30, using flags initialized to 0. Two CPUs receive the threaded code

```
      Flag1 = 1;
loop: if (Flag2 == 1) loop
      <critical section>
      Flag1 = 0;
```

```
      Flag2 = 1;
loop: if (Flag1 == 1) loop
      <critical section>
      Flag2 = 0;
```

in which each thread sets its own flag and checks the other thread's flag. Superficially it appears that if one thread begins first, the other will be forced into a **spin loop** (a branch instruction that jumps to itself), until the first thread completes and clears its flag.

The program can be checked for interleaved execution by assigning times t1, t2, t3, t4. With round robin interleaving (one possible ordering), the instructions run as

```
      Flag1 = 1;
      Flag2 = 1;
loop: if (Flag2 == 1) loop
loop: if (Flag1 == 1) loop
```

Since both flags have been set in the first two instructions, the second two instructions will create a condition of **deadlock**. The idea of sequential consistency is that the program must function properly if it executes in an interleaved order.

The program may be corrected through repeated rewriting of the flag variables:

```
loop: Flag1 = 1;
      if (Flag2 == 1)
         {Flag1 = 0 ; loop}
      <critical section>
      Flag1 = 0;
```

```
loop: Flag2 = 1;
      if (Flag1 == 1)
         {Flag2 = 0 ; loop}
      <critical section>
      Flag2 = 0;
```

When this version executes interleaved, the instructions runs without deadlock as

```
      loop: Flag1 = 1;
      loop: Flag2 = 1;
            if (Flag2 == 1) {Flag1 = 0 ; loop}
            if (Flag1 == 1) {Flag2 = 0 ; loop}
      loop: Flag1 = 1;
            <critical section>
            if (Flag2 == 1) {Flag1 = 0 ; loop}
            Flag2 = 0;
      loop: Flag1 = 1;
            if (Flag2 == 1) {Flag1 = 0 ; loop}
            <critical section>
            Flag1 = 0;
```

This version will run properly on most hardware, but is contains a subtle bug — it relies on the atomic semantics of the write to the flag variables.

Suppose that the assignment `Flag1 = 0` does not complete atomically in the hardware, so that the data read by the subsequent test `(Flag1 == 1)` has not been updated at read time. This could happen if both flags are bits in the same integer variable — when one thread writes its flag it will accidentally overwrite the flag of the other thread. This would lead to a **livelock** condition in which both CPUs keep updating their flags but never enter the critical section.

The Dekker algorithm repairs the bug associated with non-atomic hardware writes. A third variable **turn** is initialized to 1.

```
      Flag1 = 1;
      turn = 1;
loop: if (Flag2 == 1
            && turn == 1) loop
      <critical section>
      Flag1 = 0;
```

```
      Flag2 = 1;
      turn = 2;
loop: if (Flag1 == 1)
            && turn == 2) loop
      <critical section>
      Flag2 = 0;
```

When this version executes interleaved, the instructions runs without deadlock as

```
            Flag1 = 1;
            Flag2 = 1;
            turn = 1;
            turn = 2;
      loop: if (Flag2 == 1 && turn == 1) loop   // false
      loop: if (Flag1 == 1 && turn == 2) loop   // true
            <critical section>
      loop: if (Flag1 == 1 && turn == 2) loop   // true
            Flag1 = 0;
      loop: if (Flag1 == 1 && turn == 2) loop   // false
            <critical section>
            Flag2 = 0;
```

In this version, even if one of the flags is accidentally overwritten, exactly one of the conditional branches will execute as false. Modern cache coherency protocols prevent most such nonatomic memory errors.

Slide 33 – 35 Machine Language Support

Some ISAs define instructions that support program synchronization at the system level. Atomic machine language primitives provide a **hardware-level semaphore M** to enable implementation of critical constructs at the compiler level.

The most basic primitive is **test and set**. The semaphore **M** refers to an address in memory. The assembly instruction **Test_and_Set M, R** copies the value stored at address **M** to register R, and then sets **M** to 1 if its value had been 0. A variation is **fetch and add**, which writes the value stored in **M** to a register and then adds the contents of another register to **M**. Both are used with the instruction **swap** that exchanges the values in **M** and some register. The typical use of these instructions is

```
L1:  Test_and_Set M, R1 ; spinlock
     BNEZ R1, L1
     <critical section>
     Swap M, R1
```

If thread 1 and thread 2 run this code interleaved (with private copies of register R1 and M initialized to 0):

L1: Test_and_Set M, R1	R1 ← 0 M ← 1		
		L1: Test_and_Set M, R1	M = 1 R1 ← 1
BNEZ R1, L1	Not taken		
		BNEZ R1, L1	Taken
<critical section>	Execute		
		L1: Test_and_Set M, R1	M = 1 R1 ← 1
Swap M, R1	R1 ← 1 M ← 0		
		BNEZ R1, L1	Taken
		L1: Test_and_Set M, R1	R1 ← 0 M ← 1
–		BNEZ R1, L1	Not taken
		<critical section>	Execute
		Swap M, R1	R1 ← 1 M ← 0

Initializing R2 to 1, the identical behavior is obtained by replacing test and set with **Fetch_and_Add M, R1, R2**.

Slide 34 shows **compare and swap**, another possible machine instruction that can be more efficient than **test and set**. The instruction **CAS M, R1, R2** exchanges M and R2 if M equals R1 when tested. It also sets a flag when the exchange is made or clears the flag when the exchange is not made. When this instruction is used, all the CPUs may enter the critical section without restriction, but only one will be able to commit its result to state. The other CPUs will have to begin again.

A typical example is

```
L1:  LW R1, x          ; load x
     ADDI R2, R1, #1   ; prepare new value for x
     CAS x, R1, R2     ; if no change in stored x, update x
     BEQZ Cflag, L1    ; else start again
```

Any CPU can enter this section. But if the value of **x** was changed since being read by this CPU, then the CPU must cancel its work and begin again.

If thread 1 and thread 2 run this code interleaved (with private copies of registers):

L1: LW R1, x	R1 ← x		
		L1: LW R1, x	R1 ← x
ADDI R2, R1, #1	R2 ← x + 1		
		ADDI R2, R1, #1	R2 ← x + 1
CAS x, R1, R2	R1 = x x ← R2 R2 ← R1 Cflag ← 1		
		CAS x, R1, R2	R1 != x Cflag ← 0
BEQZ Cflag, L1	Not taken		
		BEQZ Cflag, L1	Taken
		L1: LW R1, x	R1 ← x
		ADDI R2, R1, #1	R2 ← x + 1
—		CAS x, R1, R2	R1 = x x ← R2 R2 ← R1 Cflag ← 1
		BEQZ Cflag, L1	Not taken

Slide 35 shows another possible pair of machine instructions that improve on CAS: **load reserve** and **store conditional**. This pair prevents a thread from updating a variable if it was written since being read in the thread, even if the new value was identical to the old value. The **load reserve** reads a memory location and writes to a flag. The **store conditional** instruction only writes to memory if the flag associated with the read is still set. A typical application of these instructions is

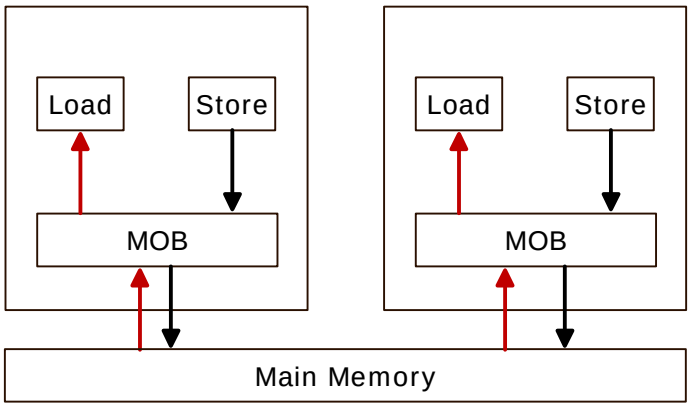
```
L1: load-reserve R1, x
    ADDI R1, R1, #1
    store-conditional x, R1
    BEQZ status, L1
```

If thread 1 and thread 2 run this code interleaved (with private copies of registers):

L1: ld-res R1, x	flag ← 1 adr ← &(x) R ← x		
		L1: ld-res R1, x	flag ← 1 adr ← &(x) R ← x
ADDI R1, R1, #1	R1 ← x + 1		
		ADDI R1, R1, #1	R1 ← x + 1
sto-cond x, R1	flag = 1 adr = &(x) flag ← 0 adr ← 0 x ← R1 status ← 1		
		sto-cond x, R1	flag != 1 adr != &(x) status ← 0
BEQZ status, L1	Not taken		
		BEQZ status, L1	Taken
–		L1: ld-res R1, x	flag ← 1 adr ← &(x) R1 ← x

Slide 36 – 40 Hardware Level Consistency without Cache

Hardware level optimizations in superscalar CPUs with dynamic execution can introduce hazards in seemingly harmless code, by changing the order of instructions. Slide 37 describes a hazard that can occur if memory writes are made through a memory ordering buffer (slide 17 in presentation 6) on the CPU. The write buffer on each CPU permits read instructions to bypass previous writes to main memory. A read hit to an address in the write buffer will update directly from the FIFO. But a read by one CPU from an address in the write buffer in a different CPU will not function properly.



For example, assembly code for two threads of the Dekker algorithm is

Thread 1		Thread 2	
	ADDI R1, R0, #1		ADDI R1, R0, #1
	SW [Flag1], R1		SW [Flag2], R1
	SW [turn], R1		SW [turn], R1
loop:	LW R2, [Flag2]	loop:	LW R2, [Flag1]
	LW R3, [turn]		LW R3, [turn]
	AND R4, R2, R3		AND R4, R2, R3
	BNEZ R4, loop		BNEZ R4, loop
	LW R5, [x]		LW R5, [x]
	ADDI R5, R5, #1		ADDI R5, R5, #1
	SW [x], R5		SW [x], R5
	SW [Flag1], R0		SW [Flag2], R0

The MOB in thread 1 could execute the LW instructions before the SW instructions that remain in the write buffer during the reads. Until the write buffer is emptied (the SW to memory performed), thread 1 will provide the value of turn from its write buffer (not knowing that thread 2 may have changed its value). Similarly, when thread 2 performs the LW R2, [Flag1] instruction, it will not know that a new value of Flag1 is sitting in the MOB of thread 1 waiting to update main memory. Both threads will execute incorrectly.

Slide 38 describes a similar problem that will occur if the physical write to a memory location takes several transmission hops over several clock cycles.

Slide 39 shows an example of another potential problem. In thread 1, the out-of-order execution may select **LW R5, [x]** to run early (it has no visible data dependencies to the single CPU). But this scheduling would read the value of x before thread 2 updated it, and the program would not execute properly. In slide 40, a **memory barrier** is inserted before the LW instruction. No instruction can be executed before the **MEMBAR** in a program listing.

Slide 41 – 44 Cache Consistency and Coherence

Slide 41 reviews the basic terminology for cache policies in a single CPU system. On a multiprocessor system, several processors access shared data in main memory, and each processor copies data blocks to its private cache. When a CPU writes to its private copy in cache, the cache and main memory become inconsistent. For example,

CPU-0 reads variable X to cache from main memory

CPU-1 reads variable X to cache from main memory

CPU-0 writes to variable X in cache

The copies of variable X in CPU-1 cache and main memory are not valid.

Strict consistency is the situation in which all copies of any data location are identical — this is almost impossible to implement. **Sequential consistency** is the situation in which processors perform operations in the specified sequential order, and the operations on each processor are performed in program order. **Cache coherence** implements sequential consistency for cache and main memory, which requires processor hardware support for cache management.

Slide 43 shows the sequence of instructions executed when a program runs in order under interleaving, but without enforcement cache coherence. In particular, the update x in the cache of CPU-0 without either updating other copies of x or alerting CPU-1 of the update, causes an execution error for the program. Slide 44 shows the same error in the case that the cache update by CPU-0 also updates memory, but does not alert CPU-1 that its cached copy is invalid.

Slide 45 – 52 Cache Coherence Protocols

The goal of cache coherence is to enforce sequential consistency for all caches and main memory, in order to permit system-wide atomic and critical constructs. The principle technique is called **Snoopy cache**. A status field is added to the tag table associated with every slot in every set in cache. When the cache manager searches for a memory block in cache (slides 45 – 46 in presentation 4), it checks both the tag, which indicates the presence of the block, and the status. The status bits are updated, depending on coherence policy, with the access history to the data.

The cache operates by **write-back policy** — changes to data in cache only update main memory when the block is swapped out of cache. This policy saves considerable work on the main data I/O bus. However, during every write to cache, the **address** being written in cache is **broadcast** on the I/O bus. Before writing to the I/O bus, a CPU must acquire access rights in competition with the other CPUs. Therefore, broadcasting the address to be written takes slightly more time than just writing to cache in an uncoordinated fashion, but takes much less time than performing a complete a data write.

The cache management system is called **snoopy** because it **monitors** all addresses written on system bus by all processors, compares written addresses with blocks in its own cache, and updates the state of cached block on an address hit. In this way, if CPU-0 and CPU-1 both copy variable **x** to cache, a write to **x** by CPU-0 only updates its private copy. But the write to **x** (according to its memory address) requires CPU-0 to acquire access rights to the I/O bus and broadcast the address of **x**. When the snoopy cache manager in CPU-1 sees the write to the address of **x**, it tags the copy of **x** in its own cache invalid.

Within snoopy cache there are two main types:

In **write update**, processor W places a write address on the I/O bus and snooping processors S_i tag their copies of the block as invalid on an address hit (if the block is in their private cache). Processor W then performs the write both to cache and to the bus. Processors S_i update their local copies of block from the I/O bus and tag the updated blocks valid. This insures a rapid update of all caches, but creates a very large bus overhead.

In **write invalidate with inquire**, processor W places a write address on the I/O bus and snooping processors S_i tag their copies of the block as invalid on an address hit. Processor performs the write to its local cache only, creating one unique valid copy of the data block. When another processor S_i needs the data clock, it performs an **inquire cycle**. S_i places the required address on the I/O bus, which forces processor W to perform a write-back and update the main memory. This method has much lower I/O bus overhead. Most shared memory systems implement write allocate with write invalidate.

Slide 47 defines the states that may be assigned to a data block in its status word.

Slide 48 shows the behavior of CPU W if a cache block is tagged as **modified**. The block is called **dirty** meaning that its value has been changed since loading from main memory (the copy in main memory is invalid). Other CPUs tag this block as invalid if it was loaded to their private caches (until it is overwritten by another block). W can continue to update this copy without updating main memory. W must responds to an inquire cycle — its cache manager snoops the memory address from some S_i that requests access to the clock. When W updates main memory for a write inquire, it also marks its own copy of the block invalid.

Slide 49 shows the behavior of CPU W if a cache block is tagged as **shared**. The block is called **clean** meaning that the value is the same as the value in main memory and other private caches. W can update this copy — it must write the update address to the I/O bus to alert processors S_i that mark their copies of the block invalid. After it updates its cache, W tags the block **modified**.

Slide 50 shows the behavior of CPU W if a cache block is tagged as **invalid**. The block must be reloaded before next read (as if it were not in cache). Any read or write access is a cache miss. Read misses are always loaded. On a write miss, the write may be performed after a load to cache (write allocate) or the write can be made directly to memory (no write allocate). Some memory locations (such as addresses used for peripheral I/O instead of data storage) are labeled "non-cacheable" and are never copied to cache.

Slide 51 shows the complete state transition diagram for the **MSI** (modified, shared, invalid) protocol. Slide 52 shows an example of a series of reads and writes of a single variable by two CPUs, and the status of the associated cache blocks. On the first read (line 1), the block is loaded to cache in processor P1 and marked shared (S). On the first write (line 2), the block is marked modifies in P1 (transition 2 from S to M). When P2 inquires to read (line 3), the block is updated to main memory and loaded to P2. P1 and P2 mark the block shared. When P2 writes (line 4), it marks the block modifies, and P1 marks the block invalid (I). And so on.

Slide 53 – 55 Processor Behavior — Exclusive

Exclusive is an additional state defined to make the protocol more efficient. When a block in the cache of processor W is marked exclusive, W cache holds the only valid copy of the data block among the caches. But the block is clean, identical to the copy in main memory. The copies in other processors S_i are marked invalid. If W updates this copy, it marks the block modified. As with modified, W responds to an inquire cycle. The exclusive status provides improved performance for private data — on eviction of an exclusive (E) block, the slot may be overwritten without any write-back to memory. Slide 55 shows an example of the extended MESI protocol.

Slide 56 – 58 Memory Thrashing with MSI

The use of a shared variable as a mutex together with the MSI protocol and snoopy cache can lead to a condition called memory thrashing (note: **th**rashing, and not **t**rashing). The value of the shared variable **mutex** is swapped with a local register in order to enforce criticality. Only the CPU that finds mutex = 0 can proceed to the critical section. But because mutex is a shared variable, each swap by a CPU causes: a read miss and a cache update, a write to one local cache (write allocate), and invalidation all other cached copies (write invalidate). These updates cause very inefficient movement of data across the I/O bus. Multiple mutex cache misses and loads create a large I/O overhead, much larger that the actual read and writes required to execute the critical section.

The main advantage of the load-reserve and store-conditional primitive (slide 35) is its compatibility with MSI. The shared variable mutex leaves most CPUs in a spin loop that involves multiple unrelated memory updates (for mutex) and memory thrashing (code on left of slide 57). The use of load-reserve and store-conditional replaces the spin loop with a sequence in which a CPU may have to repeat the critical section many times before it has the opportunity to commit its result to memory. But since this method (code on right of slide 57) performs many reads an only one write per CPU, the number of cache write updates is much smaller. When a CPU completes the critical section and writes to its cache, the other CPUs must invalidate their cache copies of shared variables, clear the reservation flag, and perform an inquire that causes the CPU to update main memory. Thus, all of the memory activity of the other CPUs takes place at the same time for the same reason — one CPU has succeeded in completing the critical section.

The memory thrashing caused by mutex is an example of a dependency called **false sharing**. This occurs when multiple threads access distinct address ranges within the same cache block, so that every write to the block causes block invalidation in other processors. An example is shown on slide 58, where a small array is used to hold unrelated variables accessed by separate threads. The small array is likely to sit in the same cache block, so that every element update invalidates the copies of every other CPU.

Slide 59 Cache Coherence and Hardware Optimizations

Slides 36 – 40 discussed several hardware optimizations in superscalar CPUs that can cause problems in shared memory systems. The MSI protocol that enforces cache coherence orders memory writes by enforcing sequential consistency. Since the CPU cannot write to its own cache without first acquiring access rights to the bus, snoopy cache insures that writes occur in a clear sequential order.

Read bypass in the MOB — require that the CPU acquire I/O access rights to before reading a queued value from its write buffer. This informs the other CPUs of the write to the cache block even if the block has not yet been written.

Write overlapping — since writes invalidate other cache copies, all physical writes must occur in the correct order.

Read/write reordering — writes invalidate other cache copies, so that the rescheduled read is invalidated when the other CPU writes to the shared variable.

Slide 60 – 62 Sun T1 Multiprocessor

A different example of a multicore processor is the Sun T1. This CPU was introduced by Sun Microsystems in 2005, based on its UltraSPARC architecture and is optimized for thread level parallelism for OLTP (slides 28 – 29 in presentation 7). The T1 contained 8 processor cores, but was a single-issue scalar CPU (one ALU per pipeline) and performed in-order execution (no dynamic scheduling). The pipeline was intended for the integer instructions characteristic of OLTP — the 8 cores shared one FPU.

The pipeline consisted of 6 stages — the 5 stages of a DLX type CPU plus a stage for thread selection. Each core handled 4 threads, performing fine-grained multithreading, switching thread on each clock cycle and skipping over stalled threads.

As seen on slide 61, each of the 8 cores had private L1 cache (split data and instruction caches) and shared L2 cache among themselves.

The T1 processors underwent considerable evolution, even when Sun was acquired by Oracle to enhance its OLTP server business. The T4 (2011) doubled the number of threads per core, superpipelined the pipeline, introduced dynamic scheduling, doubled the clock speed, added a per core FPU and cryptographic processor, and added an L3 cache. Oracle's T4 server consisting of 4 T4 CPUs (256 threads) + 1 TB RAM was found to be 25% faster than a comparable HP 4 Xeon server on the OLTP benchmark TPC-H.