6.895 Theory of Parallel Systems	Lecture 11
Implementation of Memory Consistency	
Lecturer: Bradley C. Kuszmaul	*Scribe: Seth Gilbert, Xie Yong*

Lecture Summary

1. *Sequential Consistency*
   We review the definition of sequential consistency.

2. *Uncached Consistency Algorithm*
   We present a simple algorithm and prove that it guarantees sequential consistency.

3. *Overlapping Operations and Caching*
   We discuss the difficulties introduced by allowing overlapping operations at a single processor, and by using caches.

4. *The MESI Protocol*
   We present the MESI protocol, and algorithm that takes advantage of caching to provide an efficient, sequentially consistent memory.

1 Memory Consistency

We first review the definition of sequential consistency in Section 1.1. We then define a system model in Section 1.2, and present an algorithm for implementing the usual memory operations, `load` and `store`, in Section 1.3. We then prove that this algorithm guarantees sequential consistency in Section 1.4.

1.1 Sequential Consistency

We review here the definition of sequential consistency, otherwise known as strong consistency.

Definition 1 *Assume that G is a DAG representing a multithreaded program, and that α is an execution (i.e., a single run) of G. The execution α is* sequentially consistent *if and only if there exists a topological sort (i.e., a total sort), $\mathcal{S}$, of G such that for every load, the most recent (in the total order $\mathcal{S}$) store to that memory location wrote the value that the load received.*

Note that there can be more than one topological sort of the DAG demonstrating sequential consistency.

1.2 System Model

We consider a model which contains of a set of processors, a set of memory modules and a communication network, as shown in Figure 1. We assume the following:

- The memory is organized into words.
- Processors execute `load` and `store` instructions on words.
- Processors send messages to the memory using the network.
- The memory modules perform `read` and `write` operations on the memory.

Example 1 Let the memory module M_i contain the set of all memory locations in which the low-order two bits of the page number are i. On a system with 2^{12} words per page, that means that M_i contains any memory address that when written in binary looks like:

`XXXX XXXX XXXX XXXX XXab XXXX XXXX XXXX`

where $i = 2 * a + b$, and $0 \leq a, b < 2$. M_i, then, consists of every address X for which

$$(X \gg 12)\ \ \&\ \ 3 == i$$

1.3 Uncached Consistency Algorithm

It turns out that it is easy to implement sequential consistency, if at each processor load and store operations do not overlap and processors do not cache memory values.

For this section and Section 1.4, then, we assume that each processor can only perform a single load or store operation at any given time. That is, once a processor begins an operation, it must wait until the operation completes before it can begin a new processor. We now present a simple algorithm in which there is no caching:

- `load`$(x)_i$: When processor i receives a `load` request for location x, it sends a message to the memory module containing the address x, asking for the current contents. Processor i then waits for the reply containing the value.
- `read`$(x)_j$: When memory module j receives a `read` request for location x, it reads the context of location x and sends a reply.
- `store`$(x, v)_i$: When processor i receives a `store` request to place value v in location x, it sends a message containing x and v to the memory module containing x and waits for a reply indicating that the operation is complete.
- `write`$(x, v)_j$: When the memory module j receives a write request, the memory writes v into location x and sends a reply indicating that the operation is complete.

1.4 Proving Sequential Consistency

In this section, we show that the Uncached Consistency Algorithm implements sequential consistency. To prove this, we need to first construct a topological sort of the program DAG, and then show that every `load` operations returns the most recently stored value to the same location in the total order.

Definition 2 *We sort the load and store operations in the same order in which the memory modules acutally process the corresponding messages. If the times at which two operations are processed by the memory modules overlap, then either order is allowed.*

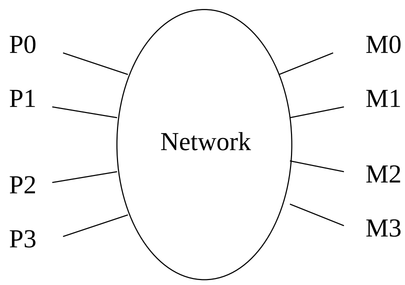

Figure 1: Model for processors to access memory through communication network.

More formally, we define sort S *as follows: given two operations in the DAG, i and j, we say that $i < j$ if the memory module accessed by i completes the operation associated with i before the memory module accessed by j completes the operation associated with j. If the operations overlap, we arbitrarily order i and j, for example, we let $i < j$ if the address associated with i is smaller than the address associated with j.*

Claim 3 *Sort* S *is a topological sort of the program DAG G where G=(V,E).*

Proof To prove that Sort S is a topological sort of the DAG, G, we need to show:

$$\forall\, i \in V, j \in V, \quad i \prec j \text{ (in } G) \quad \Rightarrow \quad i < j \text{ (in } \mathtt{S}) \ .$$

If $i \prec j$, then i sends a message to the memory, and waits for the reply before allowing any of its successors to run. The memory acutally processes the operation before sending the reply, and therefore j cannot start execution until the memory operation for i has finished. The memory operation for j cannot start until j starts. Therefore, the memory operations for i finish before the memory operation j starts, and $i < j$ in the sort S. Hence the sort S is a topological sort of the program DAG. □

Claim 4 *Every load observes the most recent store to the same location in the total order.*

Proof Consider a particular location X. The total order of all memory operations induces a total ordering of the operations on memory location X. Hence if i and j are two operations on the same memory location, where i occurs before j, then $i < j$ in the total order. For each load, the memory returns the value most recently stored value, and hence every load observes the most recent store to that location. □

Theorem 5 *The* UNCACHED CONSISTENCY ALGORITHM *implments sequential consistency*

Proof Combining Claims 3 and 4, we have shown that Sort S is a topological sort of the program DAG, G and every load observes the most recent store to the same location in the total order. Therefore, by defintion of sequential consistency, the UNCACHED CONSISTENCY ALGORITHM implements sequential consistency. □

2 More Complicated Memory

Although the UNCACHED CONSISTENCY ALGORITHM implements sequential consistency, it has two problems:

- Operations at a single processor may not overlap.
- Processors are not allowed to cache memory values.

In Sections 2.1 and 2.2 we discuss why these two problems are difficult to solve. We then present the MESI algorithm in Section 2.3, which takes advantage of caches to implement sequential consistency more efficiently.

2.1 Overlapping Operations

An overlapping operation occurs when a processor begins executing a new operations prior to receiving a reply from an earlier operations. Allowing overlapping operations lets the processor take advantage of the parallelism in operations, which is illustrated in the following example.

Example 2 Assume there is no overlapping of operations.

```
R1 = *X;
R2 = *Y;
R3 = R1 + R2;
*z = R3;
```

The critical path of this code fragment is:

- 3 round-trip delays through the network +
- 2 read operations +
- 1 write operation +
- 1 add operation

If we can fetch X and Y in parallel, we can save one read and one round trip delay, leading to a critical path of:

- 2 round-trip delays through the network +
- 1 read operations +
- 1 write operation +
- 1 add operation

Notice that round-trip delays are significantly larger than the other operations.

It is very tough, however, for hardware to detect whether the algorithm allows such parallelism. As a result, we cannot perform overlapping operations, since the algorithm might rely on sequential consistency. The following example illustrates the problem with overlapping.

Example 3 Initially all memory locations are 0.

P_0	P_1
$*X = 1$	
$*Y = 1$	
	$R_1 = *Y$ // gets 1
	$R_2 = *X$ // might get 0

Assume that the operations are allowed to overlap at processor P_0. It is possible, then, that the contents of R_2 could be 0 at the end of the execution of this code fragment. This can happen when the message for the second operation, which sets $*Y = 1$ arrives at its memory module before the message for the first operation, which sets $*X = 1$ arrives at its memory module. In this case, R_1 can return the value 1, while R_2 returns the value 0. (At some later point, the first operation will complete and $*X = 1$. This may, however, be too late.) Notice that sequential consisteny has been violated: the execution DAG orders the writing of $*X$ earlier than the writing of $*Y$, while the memory modules order the operations in the opposite order.

To solve this problem, we can try to fix the network. We can keep the packets in-order between a processor and a memory module (pair-wise First-In-First-Out). This does not fix the problem, however. If all the packets are delivered in the order in which they are sent, then we can avoid this problem. This solution, though, is not scalable.

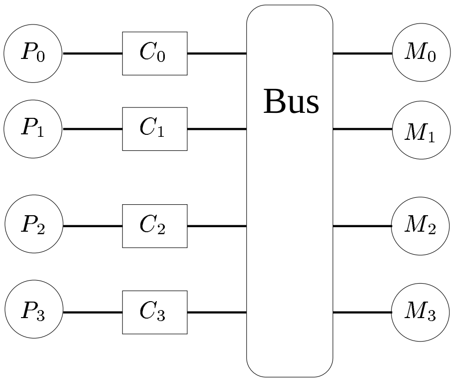

Figure 2: Diagram of a system with caches. The circles on the left represent processors, the circles on the right represent memory modules. The squares represent caches, and the oval represents a bus.

2.2 Caching

We now consider the possibility of caching memory locations. Consider again Example 2. If we allow caching, the critical path is reduced to:

- 1 read +
- 1 write +
- 1 add

in the case where all the values are cached. Caching, however, introduces a new problem. Memory may be modified by another processor, while the local cache is not updated, as shown in the following example.

Example 4

P_0	P_1
	$R_1 = *X$ // gets 0, and stores it in the cache on P_1
$*X = 1$	
$*Y = 1$	
	$R_2 = *Y$ // gets 1, strong consistency implies *X=1
	$R_1 = *X$ // still sees 0 because it is in the cache

The local cache of $*X$ on P_1 is not updated when $*X$ is updated to 1, so processor P_1 cannot see the changes of $*X$ in the memory.

2.3 MESI

The MESI protocol takes advantage of caches to implement a sequentially consistent memory more efficiently. Our system, as before, consists of processors, memory modules, and a network. Each processor, however, has a cache associated with it. (See Figure 2.) We also assume that the network is implemented by a bus. This means that every processor sees every message that is sent on the network. This allows processor to silently update their state in response to ongoing operations. Note that this also implies that only one processor can broadcast at a time, so a separate protocol is needed to handle contention. As a result, this does not scale all that well. However, this is the basis for more complicated algorithms that do scale.

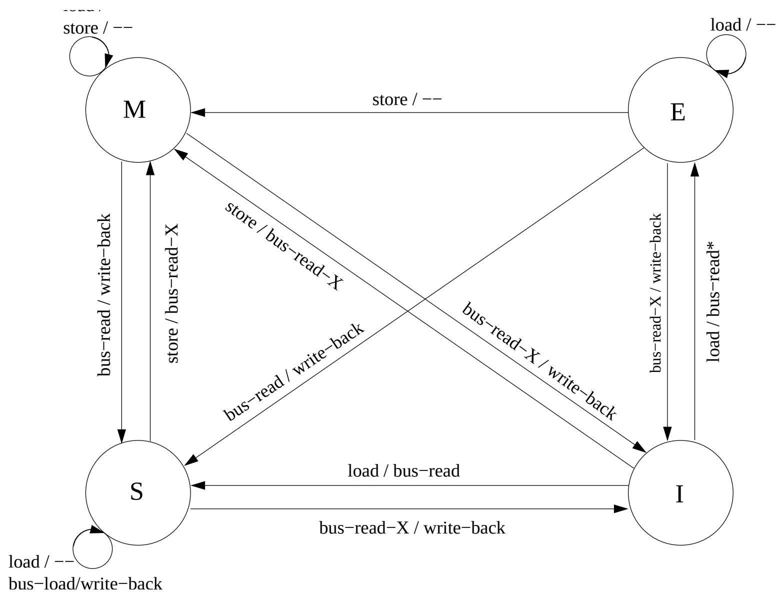

Figure 3: State diagram for the MESI protocol. Circles represent states, arrow represent transitions. Arrow are labeled `input / response`. The `input` is either a processor action (i.e., `load` or `store`) or a bus action (i.e., `bus-read` or `bus-read-X`). The responses are bus actions (i.e., `bus-read`, `bus-read-X`, or `write-back`).

The caches are (typically) organized into lines, each containing several words of data. As a result, if the algorithm chooses to cache location 100, it also fetches (and caches) locations 101, 102, and 103. This is useful, since many programs exhibit *spatial locality*: often, when a program performs an operation on a given location in the memory, it is quite likely to perform another operation on a nearby location soon afterwards. There are two types of caches.

- *Write-through caches* immediately notify the main memory of every `store` operation. This ensures that the value in the cache is always the same as the value in the main memory. (This leads to simple memory algorithms.)
- *Write-back caches* do not always immediately notify the main memory of a `store` operation. Sometimes, the cache maintains a new value, unknown to the main memory. Eventually, the value will get "written-back" to the main memory, but this may take a long time. This is more efficient, since it saves network bandwidth when the value does not really need to be written back.

The MESI protocol assumes that the system contains write-back caches. It maintains two extra bits of data for every line in the cache to determine the state of that line of the cache. Figure 3 presents the state transition protocol enforced by the algorithm. There are four possible states:

- *[M]odified*: No other cache in the system contains this line in its cache in the M, E, or S state. The cache contains a new value that needs to be written-back to main memory.
- *[E]xclusive*: No other cache in the system contains this line in its cache in the M, E, or S state. The cache contains the same value as main memory.
- *[S]hared*: No other cache in the system contains this line in its cache in the M or E state.
- *[I]nvalid*: The cache line contains no data.

A `load` operation can take place if the appropriate cache line is in the M, E, or S state. That is, a processor can read the data whenever it is available in the cache. A `store` operation can take only only if the appropriate cache line is in the E or S state. That is, a processor can only modify the memory when it it is the only cache maintaining a copy of the data. The operations are implemented as follows:

- `load`$(x)_i$: If processor i does not have the location x cached in the M, E, or S state, it sends a message on the bus requesting the location x in the S state. (This is notated in Figure 3 as a *bus-read*.) Since the message is broadcast on the bus, all other processors see this message. If some processor, j, sees this message and discovers that it has cached x in the M state, then it writes x back to memory and sets the state of x to S. (This is notated in Figure 3 as *write-back*.) If processor i sees the value written back to memory, it takes the value written and stores it in its own cache in the S state. (This is referred to as "snooping" the bus.) If no processor writes back a value to memory, then the memory module sends a message back to processor i indicating the current value of x, Processor i then caches the value of x and sets the state to S.

- `store`$(x, v)_i$: If processor i has location x in the E state, it sets the state to M and modifies the cached copy, setting it to v. If the state of x is already M, then again i can simply modify the cached copy. If x is in the S or I state, then i broadcasts a message on the bus requesting exclusive access to x. (This is notated in Figure 3 as a *bus-read-X*.) Every processor that has x in the S or E state sets the state of the cache line to I. If some processor j has x in the M state, then j writes x back to memory. If no processor writes the value back to memory, then the main memory broadcasts the value of x on the bus. Either way, processor i sees the value of x on the bus and updates its cache, setting the state to M. Processor i can then modify the value to v directly in the cache.

Further details of the MESI protocol can be found in Figure 3. Consider the following example, where $*X$ and $*Y$ are initially zero:

Example 5

	P_0	P_1
(1)		$R_1 = *Y$
(2)	$*X = 1$	
(3)	$*Y = 1$	
(4)		$R_1 = *Y$

In line 1, P_1 sets R_1 to zero, and sets the cache-line containing Y to S or E. In line 2, processor P_0 updates the value of X, and as a result the cache-line containing X is set to the M state. In line 3, processor P_0 similarly wants to update the value of Y. However, P_0 first has to obtain exclusive access to that cache-line. Processor P_0 therefore broadcasts on the bus a message requesting exclusive access to Y. When processor P_1 sees this request, it broadcasts its value of Y, and sets the state to I. Processor P_0 records this value of Y and sets the state to M. It then modifies the value directly in the cache. In line 4, processor P_1 wants to read Y. As a result, P_1 broadcasts a request for shared access to Y. Processor P_0 sees this request, and writes the value of Y back to main memory, setting its state locally to S. Processor P_1 snoops the value, and records the value of Y in its cache, setting it to state S.

The response `bus-read` indicates that the bus action resulted in an exclusive read.