Animations of Important Concepts in Parallel Computer Architecture

Mohit Gambhir
Edward F. Gehringer
Yan Solihin
North Carolina State University
{mgambhi, efg, solihin}@ncsu.edu

Abstract

Resources for teaching parallel computer architecture—specifically, cache coherence and memory consistency—are increasing in importance. Through instructor-created templates, followed by peer-reviewed student work, we have produced a set of animations that can be used for classroom presentation or self-study. These animations cover bus-based coherence protocols, such as MSI, MOESI, and Dragon; and network-based protocols, such as the full bit-vector scheme and a simplified version of SCI. Some animations illustrate the operation of a particular protocol, while others compare protocols against each other. Other animations cover memory-consistency models, such as sequential consistency, processor consistency, weak ordering, and release consistency.

Categories and subject descriptors

K.3.2 [**Computer and information science education]** Computer science education. C.1.4 **[Parallel architectures]**

1. Introduction

As uniprocessor architectures approach their physical limits, a good understanding of parallel architectures is becoming essential for anyone working in high-performance computing. The importance of this knowledge is growing rapidly as commodity architectures are adopting multicore designs. Moreover, unlike instruction-level parallelism, which is mostly "under the hood" for the application programmer, multiprocessor architectures affect the programming model—which means that programmers as well as architects need to understand how they work.

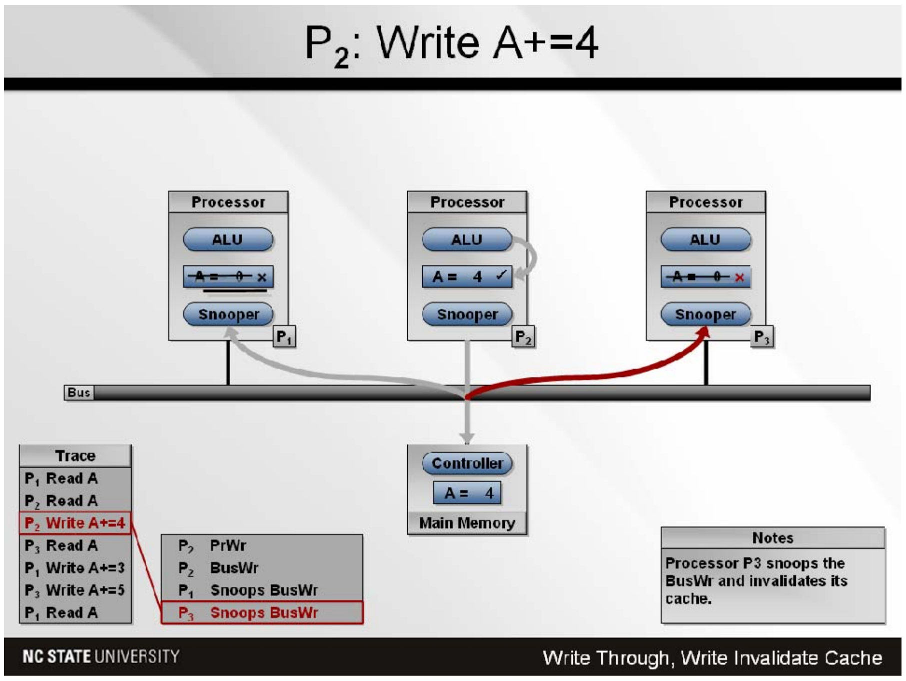
Two of the most essential concepts in parallel architecture are cache coherence and memory consistency. Students must understand the need for coherence, and tradeoffs among different coherence protocols. An even more difficult concept, in our experience, is memory consistency. It takes considerable study to grasp the implications of what it means for a memory operation to complete, and how the programming model needs to change when sequential consistency is not assured.

Resources abound for teaching ILP concepts like Tomasulo's algorithm; a Google search will bring up many animations suitable for presentation or self-study. The situation is far different for parallel architectures. For cache-coherence protocols, it is easy to locate state-transition diagrams, but hard to find animations, or step-by-step comparisons of different algorithms.

2. Creation of the animations

We have created PowerPoint animations of cache coherence and memory consistency. The original version of these animations was created by Yan Solihin in 2006, for NCSU's CSC/ECE 506, Architecture of Parallel Computers. When Ed Gehringer taught the course, he and Mohit Gambhir devised homework assignments that consisted of enhancements to the animations, and animations of protocols not covered by the original animations. Each student was assigned one such animation project. Some students did their project early in the semester,

WCAE '07, June 9, 2007 San Diego, CA
Copyright 2007 ACM 978-1-59593-797-1/07/0006…$5.00



Processor P3 snoops the BusWr and invalidates its cache.

This snapshot is taken from the basic 2-state snoopy cache-coherence protocol. Processor 2 writes to a line that is shared by the three processors. The line is written through to the memory; processor 1's and processor 3's cache are invalidated. The write by processor 2 involves a sequence of 4 operations, which are shown on four consecutive slides. The four operations are listed in a box at the lower left, with a pointer back to the program trace showing where in the program these operations occur. The above slide illustrates the last of those operations – a snoop and invalidation of processor 3's cache line (shown by the red arrow pointing to processor 3's snooper).

Figure 1. Write-through, write-invalidate cache

when the class was covering coherence; others chose later deadlines, when consistency was being studied. In this way, students were assigned to work on the animations while the material was still fresh in their mind.

This assignment was part of our Expertiza project [1, 2, 3]. The thesis of the project is that students themselves are capable of producing high-quality learning materials when given good feedback from their peers. Students chose different animations to work on. Usually, a particular animation was chosen by three to six students. Then the work was peer-reviewed by other students, who gave feedback and offered suggestions for improving it. The student reviewers were students who had chosen a *different* animation to work on.

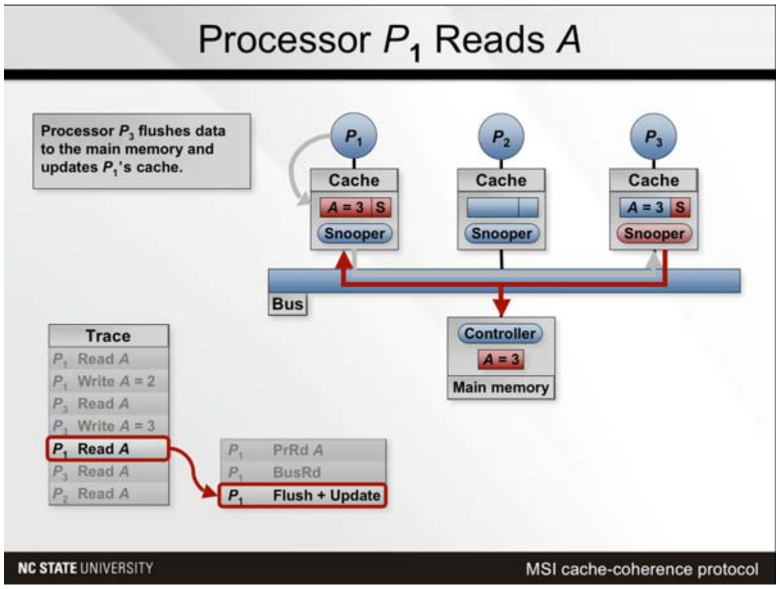
Insofar as possible, each student reviewed only one kind of animation. The reviewers thereby gained expertise in what was possible for the given problem. Since this was not the same assignment that they were doing, they could not "steal" ideas from their reviewees. Finally, we chose submissions that were rated most highly by students to be included in the final suite. We checked them for technical correctness. Then they were professionally formatted by NCSU's Distance Education and Learning Technologies Applications group. We believe they will be useful resource for instructors at other institutions who teach parallel computer architecture.

3. The animations

The animations proceed from simple to complex concepts. Arguably, the simplest coherence algorithm is write-through, so it is the subject of the first animation (Figure 1). One of the simplest write-back protocols [4] is MSI (Figure 2). MSI is a bus-based protocol, usable only where there is a shared bus that connects all processors. For multiprocessors larger than 30 or so nodes, such protocols do not suffice, and directory protocols are needed. The archetypal directory protocol is the full bit-vector protocol (Figure 3). The SCI protocol is a practical protocol, but its 29 states make it daunting to present in the classroom. Thus, we designed a simplified SCI (SSCI), shown in Figure 4.

Another class of animations are those that compare one protocol with another. These allow the student to grasp additional protocols more quickly than by studying them "from scratch." Figure 5 compares the MESI and MOESI protocols, while Figure 6 compares a 3-state and 4-state update protocol (Firefly and Dragon, respectively).

The motivation for memory consistency can be illustrated by a simple parallel algorithm, such as Peterson's algorithm. The animation shown in Figure 7 demonstrates that Peterson's algorithm will not function correctly without sequentially consistent memory. Figures 8 and 9 are snapshots from



This snapshot illustrates the MSI cache-coherence protocol. Processor 1 reads a line that is invalid in its own cache and is present in processor 3's cache in the *modified* state. The read by processor 1 involves a sequence of three operations, which are shown on three consecutive slides. The above slide illustrates the third of those operations – a flush by processor 3 to update the memory and supply data to processor 1 (shown by highlighted arrows).

Figure 2. The MSI cache-coherence protocol

animations of weak ordering and release consistency, respectively.

4. Conclusions

The interested reader is invited to peruse the figures and read the captions to see how these animations will help students understand coherence and consistency. The majority of the animations in the suite are reformatted versions of student-created slides from our Spring 2007 parallel-architecture class. In Fall 2007, we plan to add animations of additional protocols and models, with the goal of making this a comprehensive resource for visualizing important concepts in parallel computer architecture. For others who would like to create their own animations, an extensive library of widgets is available, as shown in Figure 10.

All materials referenced in this paper are available at http://people.engr.ncsu.edu/efg/animations.

5. Acknowledgments

We gratefully acknowledge the support of NCSU Distance Education and Learning Technologies Applications for assistance with design and formatting of the slides, and for supporting the first author. Colin Bryan helped write earlier versions of the assignments that these students did. Thomas Wilson was responsible for the formatting of the slides. The Expertiza project is supported by the National Science Foundation under grant 0536558.

6. References

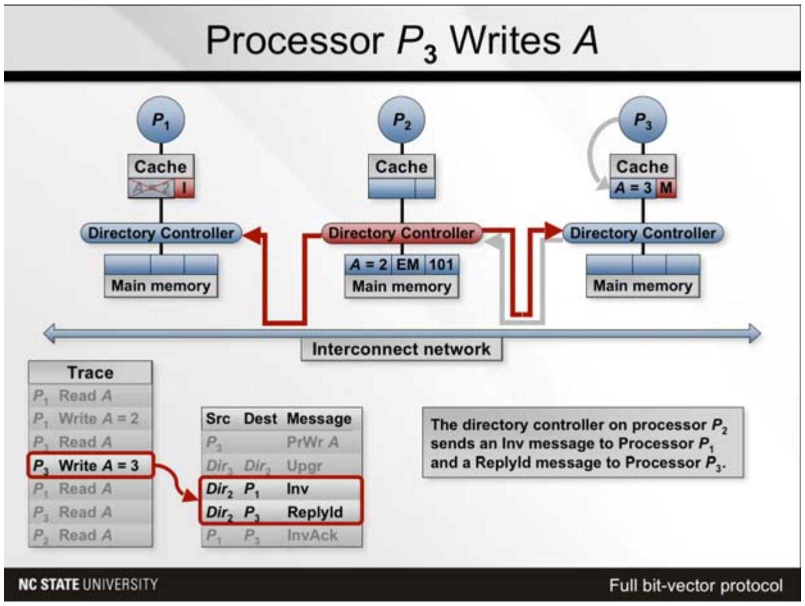
[1] Edward F. Gehringer, "Building resources for teaching computer architecture through electronic peer review," Workshop on Computer Architecture Education, 29th International Symposium on Computer Architecture, San Diego, CA, June 8, 2003.

[2 Edward F. Gehringer, Luke M. Ehresman, and Dale J. Skrien, "Expertiza: Students Helping to Write an OOD Text", OOPSLA 2006 Educators' Symposium, Portland, OR, October 23, 2006.

[3] Edward F. Gehringer, Luke M. Ehresman, Susan G. Conger, and Prasad A. Wagle, "Reusable learning objects through peer review: The Expertiza approach," *Innovate—Journal of Online Education* 3:6 (August/September 2007), to appear.

[4] David E. Culler, Jaswinder Pal Singh, with Anoop Gupta, Parallel Computer Architecture: A Hardware/Software Approach, © 1999 Morgan-Kauffman, ISBN 1-55860-343-3.

[5] Tanenbaum, Andrew S., *Distributed Operating Systems*, Prentice-Hall, 1995.



Processor P_3 Writes A

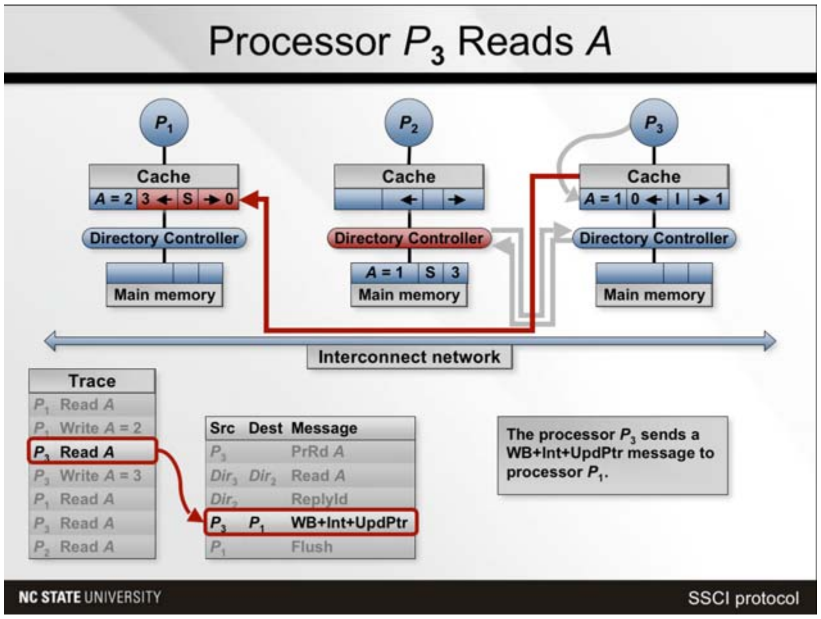
Trace
P_1 Read A
P_1 Write $A = 2$
P_3 Read A
P_3 Write $A = 3$
P_1 Read A
P_3 Read A
P_2 Read A

Src	Dest	Message
P_3		PrWr A
Dir_3	Dir_2	Upgr
Dir_2	P_1	Inv
Dir_2	P_3	ReplyId
P_1	P_3	InvAck

The directory controller on processor P_2 sends an Inv message to Processor P_1 and a ReplyId message to Processor P_3.

This snapshot is taken from a directory-based cache-coherence protocol that uses a full bit-vector to record where each block is cached. Processor 3 writes to a line that's in the *shared* state. The write by processor 3 involves a sequence of four operations, which are shown on four consecutive slides. The above slide illustrates the third of those operations – the directory sends an *Inv* (invalidation) message to processor 1 and a *ReplyId* message to processor 3, which processor 3 will later match with the *InvAck* that processor 3 will receive from processor 1.

Figure 3. Full bit-vector directory-based cache-coherence protocol



Processor P_3 Reads A

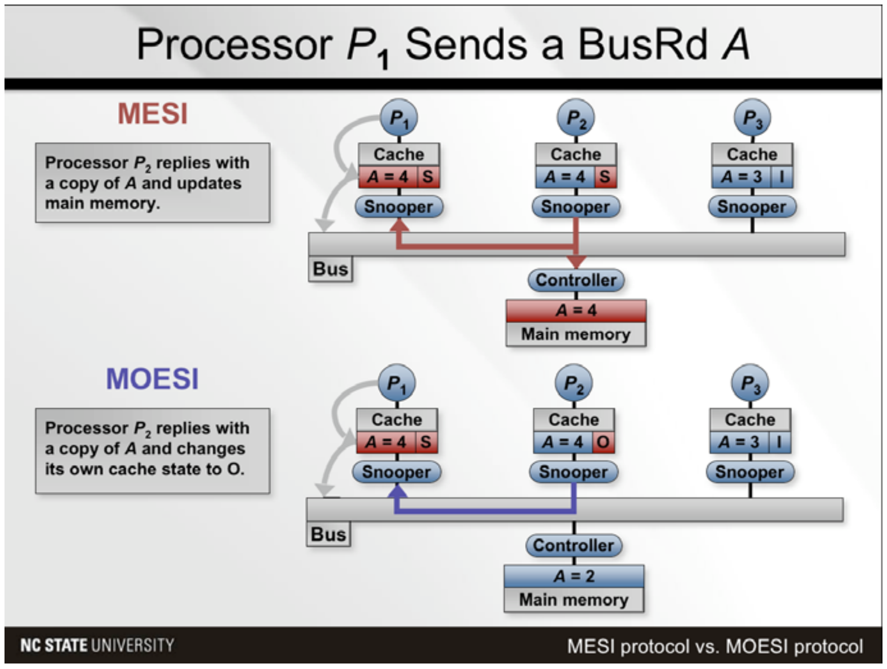
Trace
P_1 Read A
P_1 Write $A = 2$
P_3 Read A
P_3 Write $A = 3$
P_1 Read A
P_3 Read A
P_2 Read A

Src	Dest	Message
P_3		PrRd A
Dir_3	Dir_2	Read A
Dir_2		ReplyId
P_3	P_1	WB+Int+UpdPtr
P_1		Flush

The processor P_3 sends a WB+Int+UpdPtr message to processor P_1.

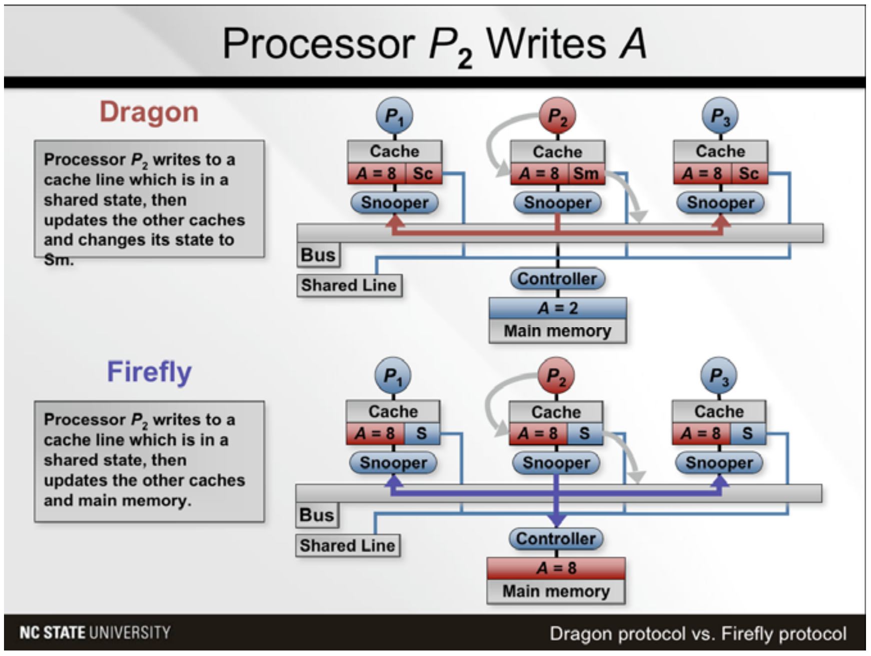
This snapshot is taken from the simplified SCI directory based cache-coherence protocol. Processor 3 reads a line that is invalid in its own cache and is present in processor 1's cache in the *modified* state. The read by processor 3 involves a sequence of five operations, which are shown on five consecutive slides. The above slide illustrates the fourth of those operations – the directory sends a *WB* + *Int* + *UpdPtr* (writeback, intervention and update pointer) message to processor 1.

Figure 4. Simplified SCI directory-based cache-coherence protocol


Processor P_1 Sends a BusRd A
MESI
Processor P_2 replies with a copy of A and updates main memory.
P_1
P_2
P_3
Cache
A = 4 S
A = 4 S
A = 3 I
Snooper
Bus
Controller
A = 4
Main memory
MOESI
Processor P_2 replies with a copy of A and changes its own cache state to O.
A = 4 S
A = 4 O
A = 3 I
A = 2
NC STATE UNIVERSITY
MESI protocol vs. MOESI protocol

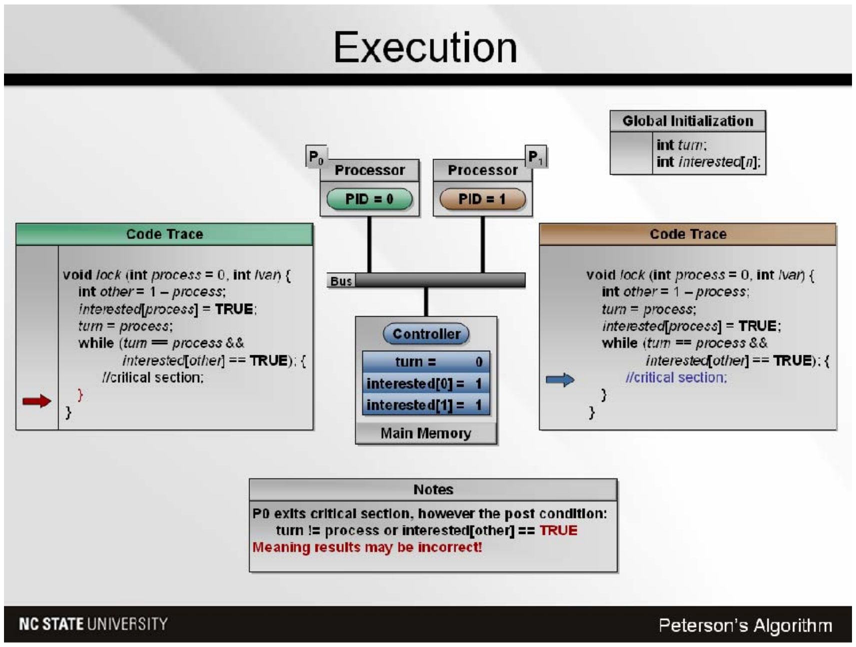
This snapshot is taken from an animation illustrating the difference between the MESI and MOESI cache-coherence protocols. Processor 1 reads a cache line that is invalid in its own cache but is present in processor 2's cache in the *modified* state. In both protocols, the value is supplied by processor 2. However, MOESI does not write the value to memory; instead, processor 2's cache changes its state to O (*owner*), and it will be responsible for updating the memory when this line is replaced. The MESI protocol, however, updates memory along with processor 1's cache.

Figure 5. Comparison of MESI and MOESI protocols


Processor P_2 Writes A
Dragon
Processor P_2 writes to a cache line which is in a shared state, then updates the other caches and changes its state to Sm.
P_1
P_2
P_3
Cache
A = 8 Sc
A = 8 Sm
A = 8 Sc
Snooper
Bus
Shared Line
Controller
A = 2
Main memory
Firefly
Processor P_2 writes to a cache line which is in a shared state, then updates the other caches and main memory.
A = 8 S
A = 8 S
A = 8 S
A = 8
NC STATE UNIVERSITY
Dragon protocol vs. Firefly protocol

This snapshot is taken from an animation illustrating the difference between the Dragon and Firefly protocols. Processor 2 writes to a cache line that is in the *shared* state. Both Dragon and Firefly are update-based protocols, so the value is propagated to other caches. However, the Dragon protocol does not write the value into memory; processor 2's cache changes its state to *shared-modified* (Sm), and will be responsible for updating the memory when this line is replaced. The Firefly protocol updates memory along with the other caches.

Figure 6. Comparison of Dragon and Firefly update-based protocols



Execution

Global Initialization

```
int turn;
int interested[n];
```

Code Trace (P0, PID = 0)

```
void lock (int process = 0, int lvar) {
   int other = 1 – process;
   interested[process] = TRUE;
   turn = process;
   while (turn == process &&
           interested[other] == TRUE); {
      //critical section;
   }
}
```

Code Trace (P1, PID = 1)

```
void lock (int process = 0, int lvar) {
   int other = 1 – process;
   turn = process;
   interested[process] = TRUE;
   while (turn == process &&
           interested[other] == TRUE); {
      //critical section;
   }
}
```

Main Memory

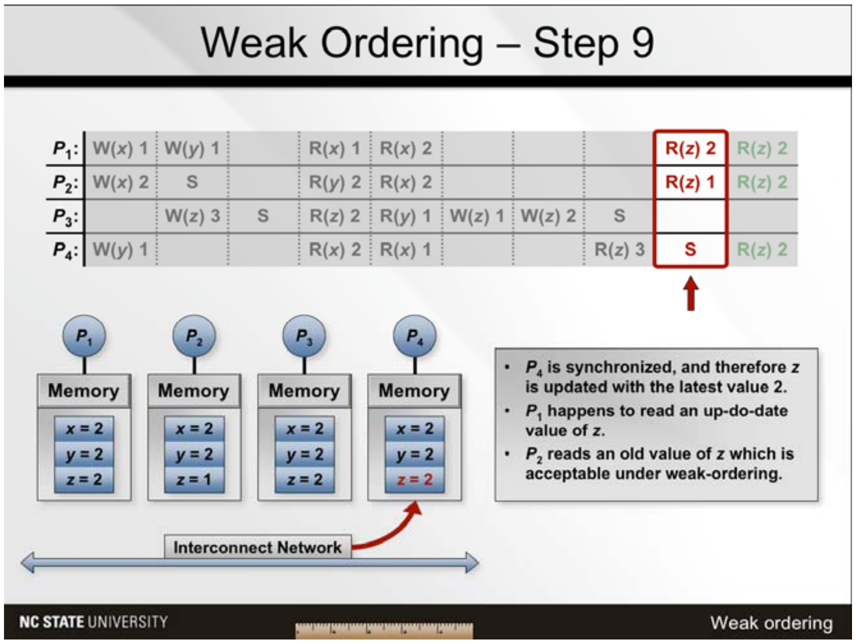
turn =	0
interested[0] =	1
interested[1] =	1

Notes

P0 exits critical section, however the post condition:
turn != process or interested[other] == TRUE
Meaning results may be incorrect!

This snapshot is taken from an animation illustrating the problem that non-sequentially consistent memory poses for Peterson's algorithm. The animation shows step-by-step execution of the given code trace by the two processors. The red arrow points to the statement that is currently being executed; the blue arrow points to the previously executed statement. Processor 1's writes to the variables *turn* and *interested*[1] hit the memory in reverse order, which allows both processors to enter the critical section concurrently.

Figure 7. Petersons's algorithm in non-sequentially consistent memory



Weak Ordering – Step 9

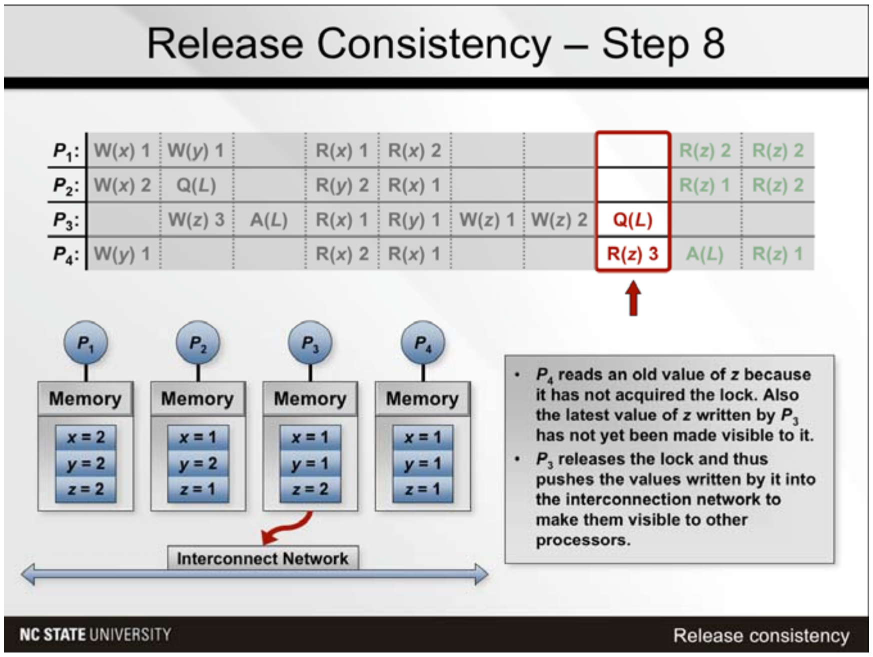
P_1:	W(x) 1	W(y) 1		R(x) 1	R(x) 2				R(z) 2	R(z) 2
P_2:	W(x) 2	S		R(y) 2	R(x) 2				R(z) 1	R(z) 2
P_3:		W(z) 3	S	R(z) 2	R(y) 1	W(z) 1	W(z) 2	S		
P_4:	W(y) 1			R(x) 2	R(x) 1			R(z) 3	S	R(z) 2

P_1 Memory	P_2 Memory	P_3 Memory	P_4 Memory
$x = 2$	$x = 2$	$x = 2$	$x = 2$
$y = 2$	$y = 2$	$y = 2$	$y = 2$
$z = 2$	$z = 1$	$z = 2$	$z = 2$

- P_4 is synchronized, and therefore *z* is updated with the latest value 2.
- P_1 happens to read an up-do-date value of *z*.
- P_2 reads an old value of *z* which is acceptable under weak-ordering.

This is from an animation illustrating the weak-ordering memory-consistency model. The grid at the top shows the sequence of memory accesses performed by the four processors (in a format due to Tanenbaum [5]). The horizontal axis represents time. Each slide illustrates the memory accesses performed in one time unit (highlighted in red). The different processors may see different values for the same variables (cf. *z* in this example). The notes give a brief explanation of why those values are acceptable under weak ordering. Also shown are the messages that need to be passed through the interconnection network in order to implement weak ordering.

Figure 8. Weak-ordering memory-consistency model



Release Consistency – Step 8

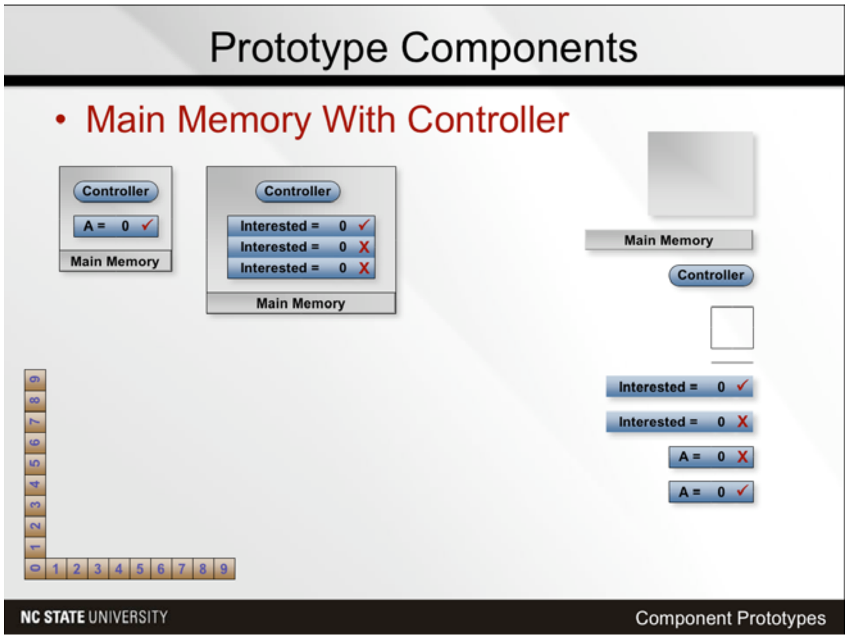
P_1:	W(x) 1	W(y) 1		R(x) 1	R(x) 2				R(z) 2	R(z) 2
P_2:	W(x) 2	Q(L)		R(y) 2	R(x) 1				R(z) 1	R(z) 2
P_3:		W(z) 3	A(L)	R(x) 1	R(y) 1	W(z) 1	W(z) 2	Q(L)		
P_4:	W(y) 1			R(x) 2	R(x) 1			R(z) 3	A(L)	R(z) 1

- P_4 reads an old value of z because it has not acquired the lock. Also the latest value of z written by P_3 has not yet been made visible to it.
- P_3 releases the lock and thus pushes the values written by it into the interconnection network to make them visible to other processors.

NC STATE UNIVERSITY Release consistency

The above snapshot is taken from an animation illustrating release consistency. The grid on top shows the sequence of memory accesses performed by the four processors. The horizontal axis represents time. The memory accesses performed in the current time-step are highlighted in red. Values seen by the various processors are shown. The notes explain why those values are acceptable under release consistency. Also shown are the messages that need to be passed through the interconnection network in order to achieve release consistency.

Figure 9 Release-Consistency Model.



Prototype Components

- Main Memory With Controller

NC STATE UNIVERSITY Component Prototypes

A large library of PowerPoint components is available for faculty and students in other classes to build their own animations. Shown are widgets for creating diagrams of memory or cache locations; other templates are available for arrows, buses, operation traces, and connectors that reshape themselves as they are dragged to illustrate a path between two components.

Figure 10. Components for Building Additional Animations