Shared Memory

SMP Architectures and Programming

Why work with shared memory parallel programming?

- Speed
- Ease of use
- CLUMPS
- Good starting point

Shared Memory

- Processes or threads share memory
- No explicit communication is needed, only synchronization
- Easy to program

”No explicit communication needed”

- Can a process work at any time?
  - No – we still need communication
- Barriers, mutex and signals are much more important in SMP than in distributed memory architectures
  - Swap
  - Test-and-set
  - Interrupts/systemcalls

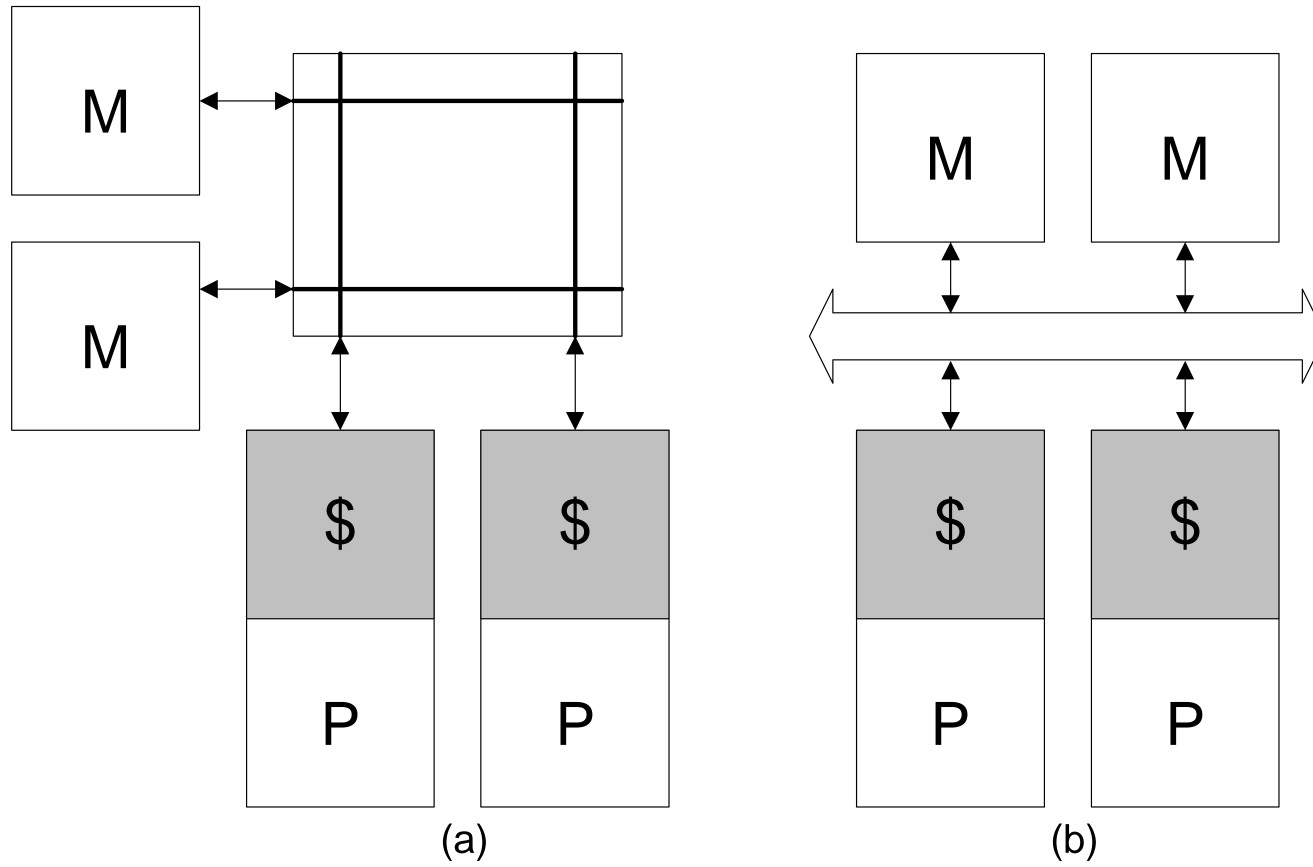
Shared Memory Architecture

- UMA
  - Uniform Memory Architecture
- NUMA
  - Non Uniform Memory Architecture

SMP Architectures

- Shared bus
  - Very simple
  - Fairly Cheap
  - Poor scalability
- Crossbar Switch
  - More complex
  - More expensive
  - Scales better

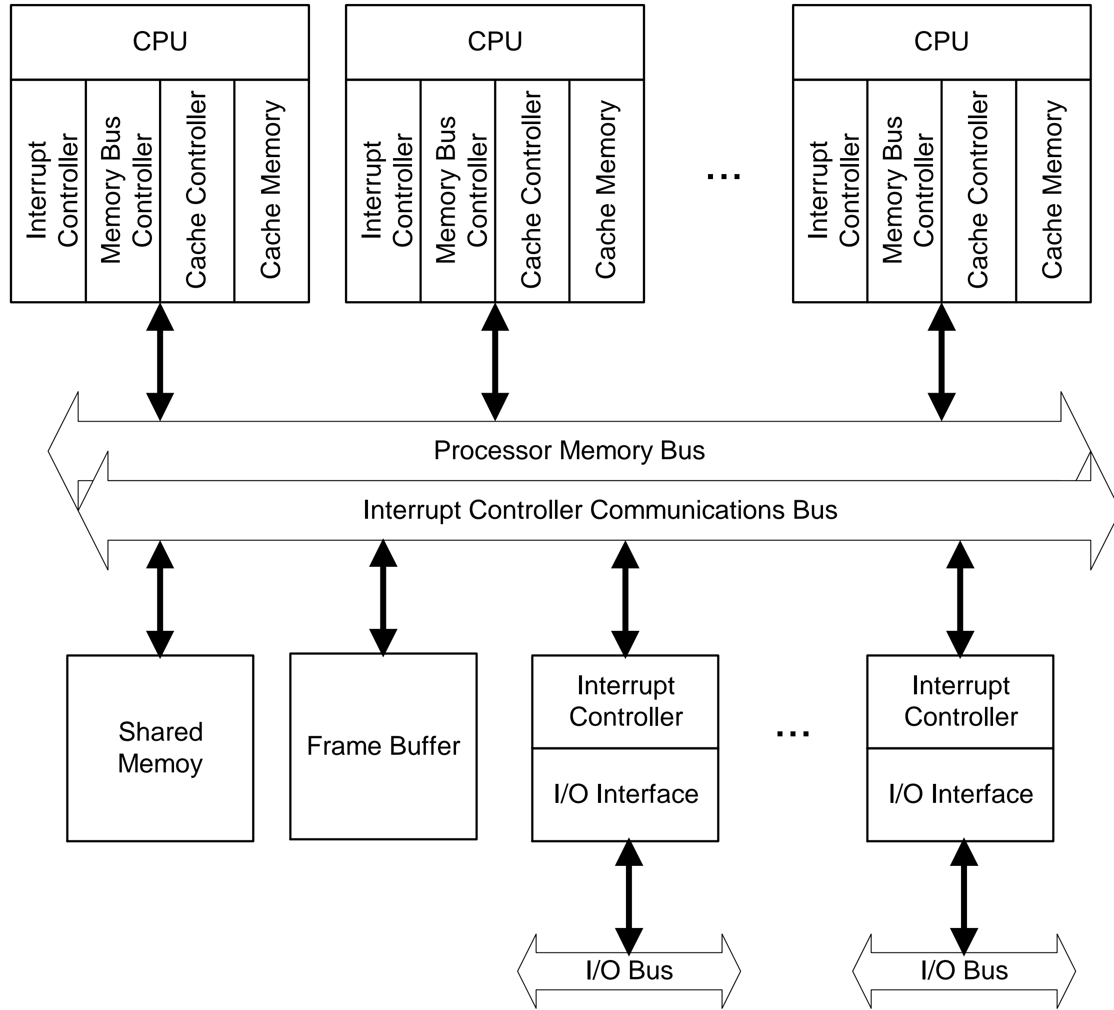
SMP HW



SHMP – a quick refresh

- Shared bus
  - Rather simple
  - Very cheap
  - Only scales to a few processors
  - Maintains the standard memory view

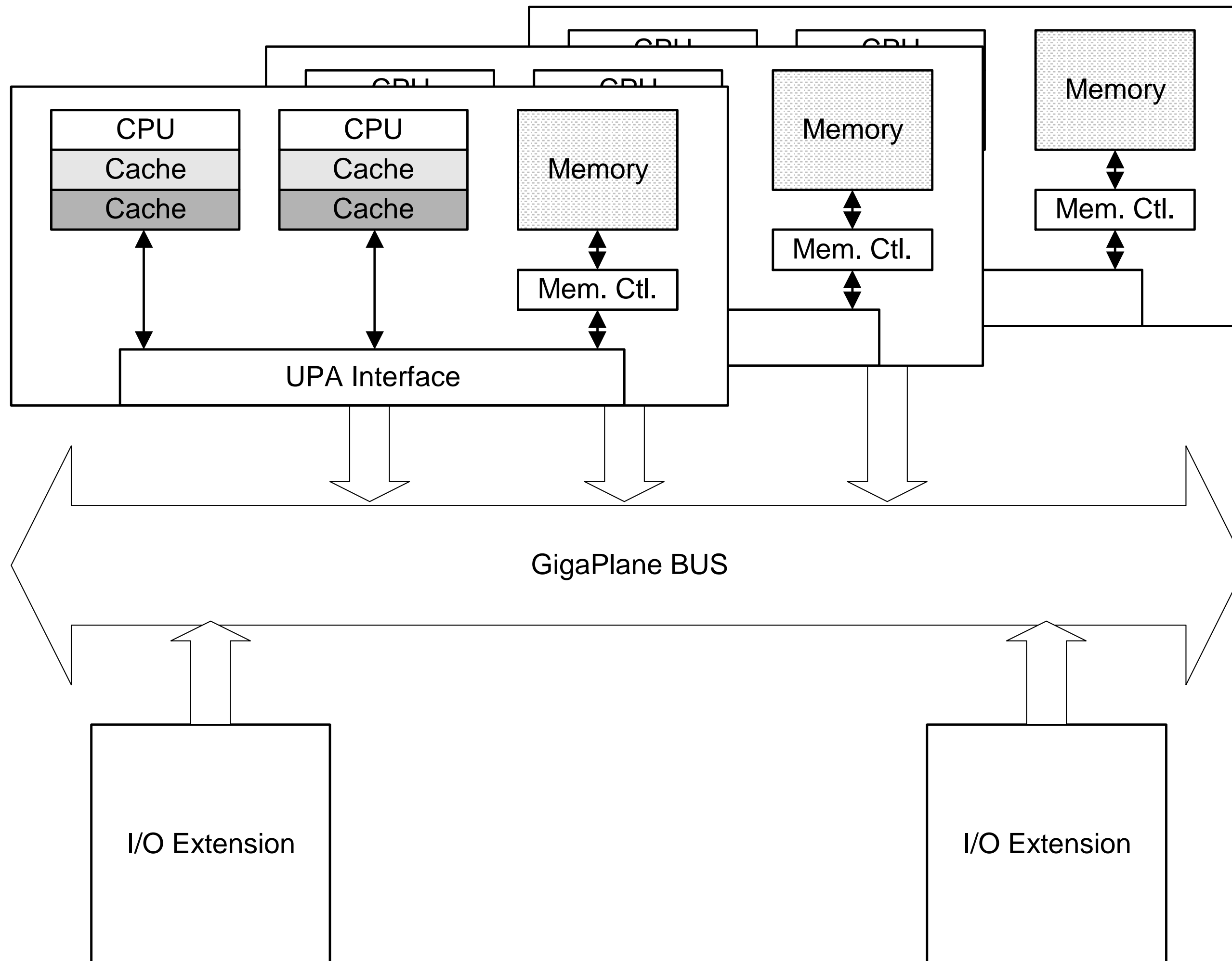
Shared Bus



SHMP – a quick refresh

- Crossbar switched
  - Rather complex
  - Quite expensive
  - Can scale to tens of processors
  - Needs a relaxed memory consistency protocol

Crossbar switch



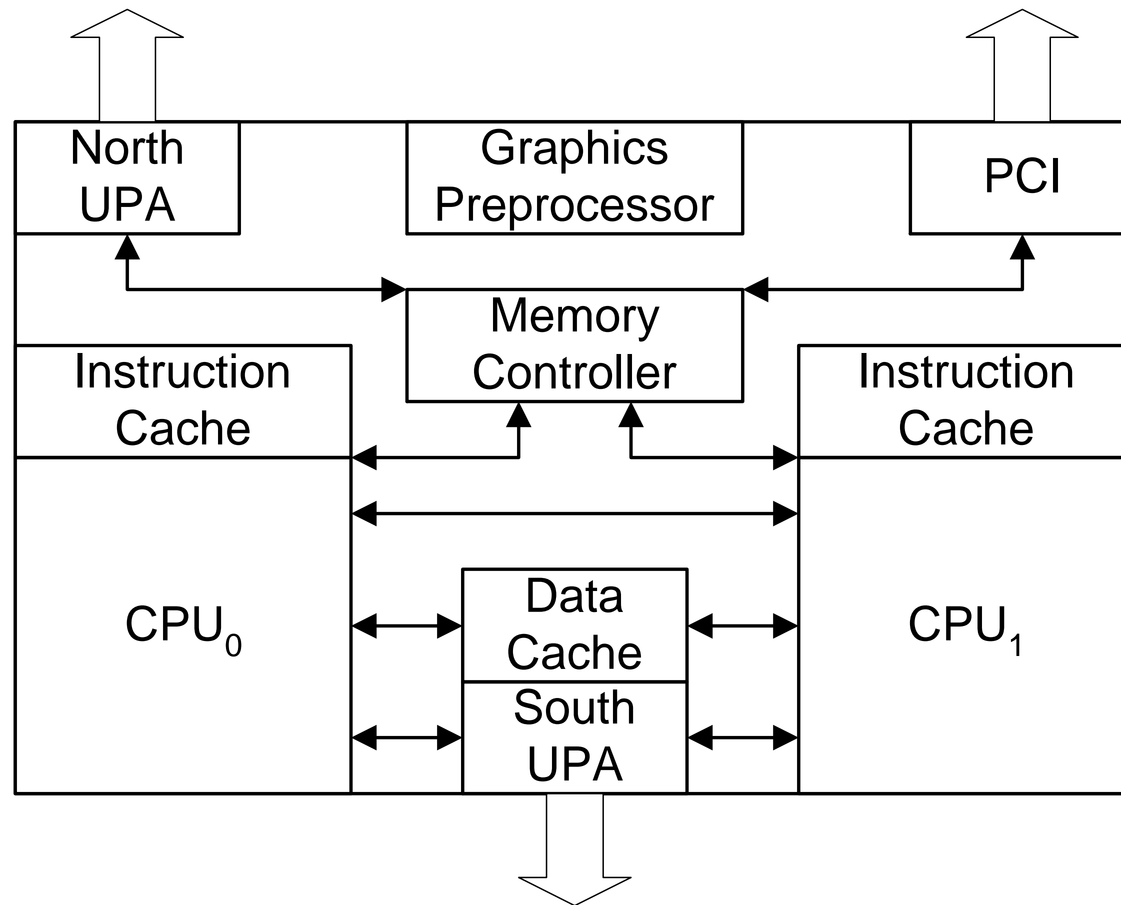
COMA Architectures

- Eliminates all ”main memory”
- Cache entries are the only thing in the system
- Problems:
  - Invalidating all copies on write
  - Not eliminating the last copy

SHMP – a quick refresh

- MP on a chip
  - Extremely simple
  - Extremely cheap
  - Only very few processors per chip
  - Allows the CPUs to work together more closely

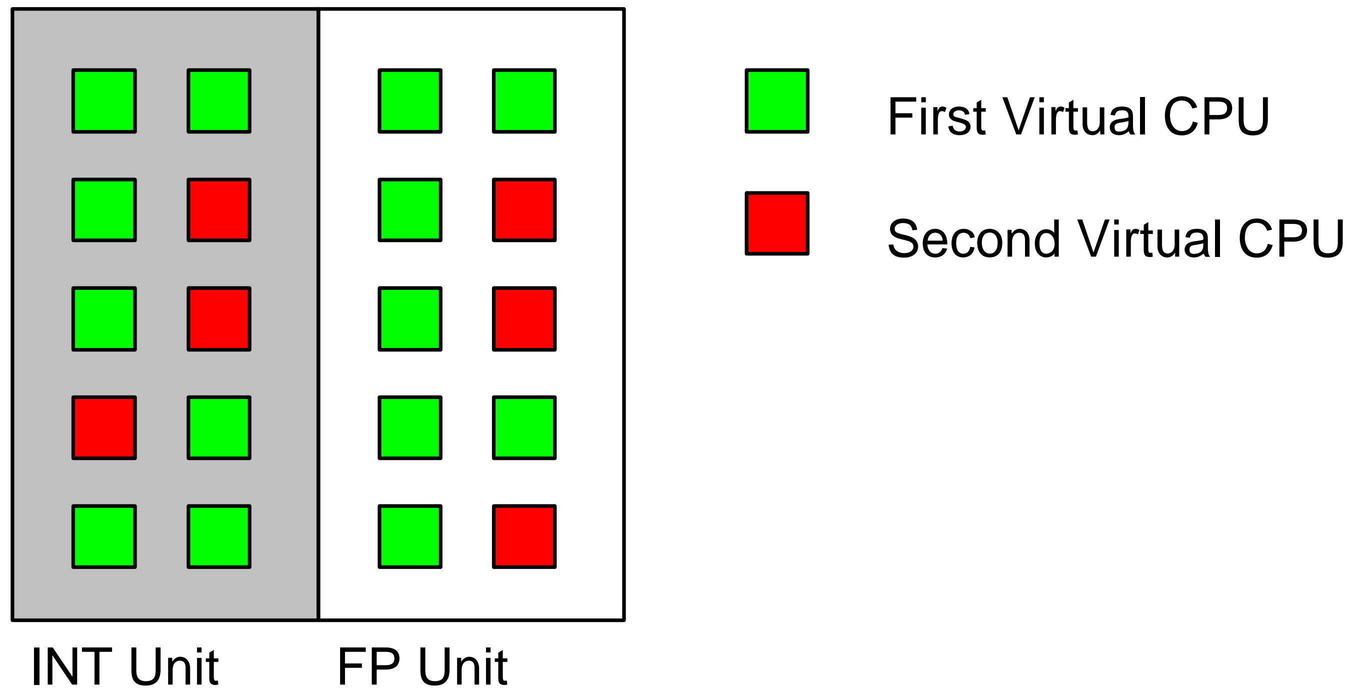
MP on a chip (Majc)



SHMP – a quick refresh

- Virtual MP on a chip
  - Named Hyper-threading
  - Extremely cheap – only an extra register-file per VP and some control logic
  - Virtual depth can be quite large but few applications may take advantage of it
  - Allows us much better utilization of the CPU area

MP on a chip



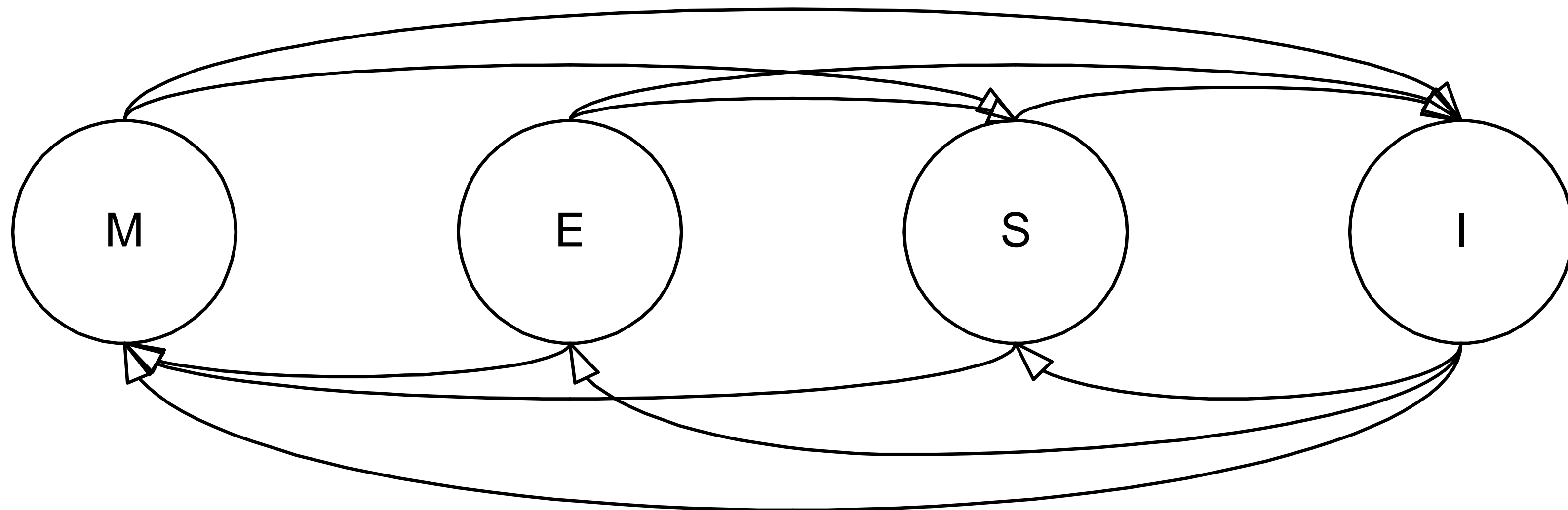
Processes and Threads

- A thread is simply a process which shares the address space of the process it resides in with the other threads in that process
- What is the importance of Threads vs Processes in SMP?

The MESI Protocol

- Common protocol for ensuring sequential consistency
- States are
  - Modified
  - Exclusive
  - Shared
  - Invalid

MESI Protocol



False Sharing

- Unfortunate memory layout may cause performance problems due to ’false sharing’

```
Int cnt[2];
A: cnt[0]++;
B: cnt[1]++;
```

cacheline

False Sharing: A test!

```
volatile int count[2];

#ifdef FALSE
worker(void * arg)
{
  int i,index=(int)arg;
  for(i=0;i<100000000; i++)
    count[index]++;
}
#else
worker(void * arg)
{
  int i,index=(int)arg;
  int temp=0;

  for(i=0;i<100000000; i++)
    temp++;

    count[index]+=temp;
}
#endif
```

```
main()
{
  pthread_t t;

  pthread_create(&t,NULL,worker,NULL);
  worker((void *)1);

  printf("%d %d\n",count[0],count[1]);

}
```

```
False Sharing
13.190u 0.020s 0:06.79 194.5%   0+0k 0+0io 245pf+0w

No False Sharing
  2.690u 0.000s 0:01.36 197.7%    0+0k 0+0io 245pf+0w
```

Scaling Shared Memory

- MESI requires broadcast of memory operations
- To scale SHM architectures above a few tens of processors we may relax memory consistency

Defining Memory Consistency

A Memory Consistency Model defines a set of constraints that must be meet by a system to conform to the given consistency model. These constraints define a set of rules that define how memory operations are viewed relative to:

- *Real time*
- *Each other*
- *Different nodes*

Why Relax the Consistency Model

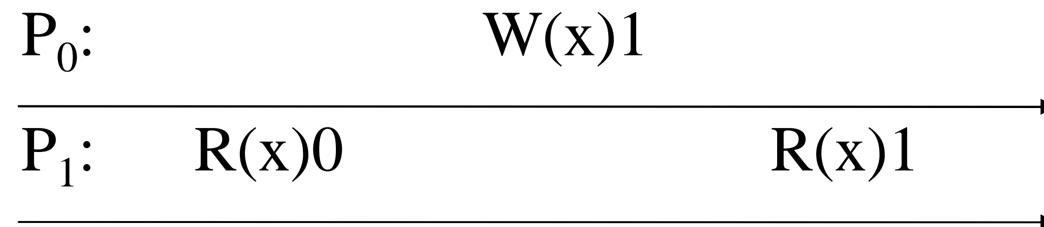
- To simplify bus design on SMP systems
  - More relaxed consistency models requires less bus bandwidth
  - More relaxed consistency requires less cache synchronization
- To lower contention on DSM systems
  - More relaxed consistency models allows better sharing
  - More relaxed consistency models requires less interconnect bandwidth

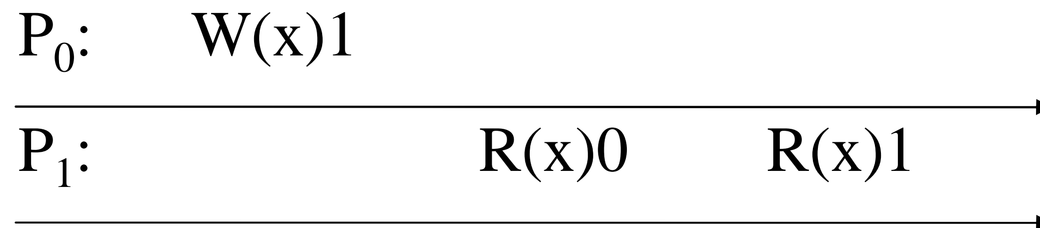
Strict Consistency

Any read to a memory location x returns the value stored by the most resent write to x.

- Performs correctly with race conditions
- Can’t be implemented in systems with more than one CPU

Strict Consistency



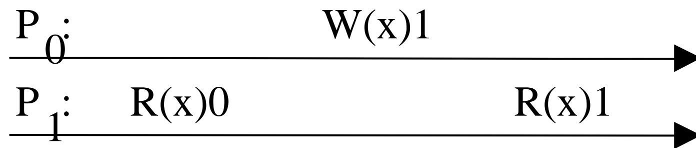


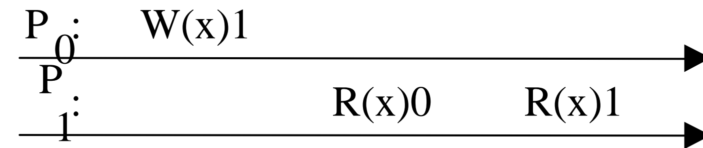
Sequential Consistency

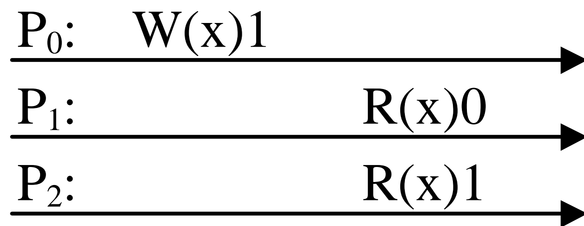
[A multiprocessor system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appears in this sequence in the order specified by its program.

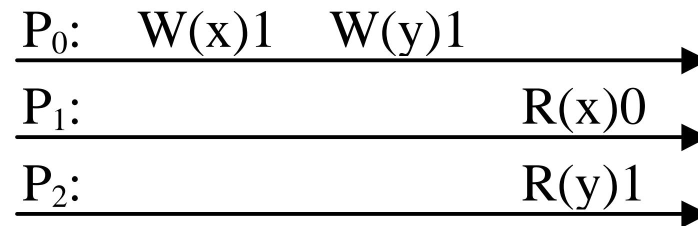
- Handles all correct code, except race conditions
- Can be implemented with more than one CPU

Sequential Consistency







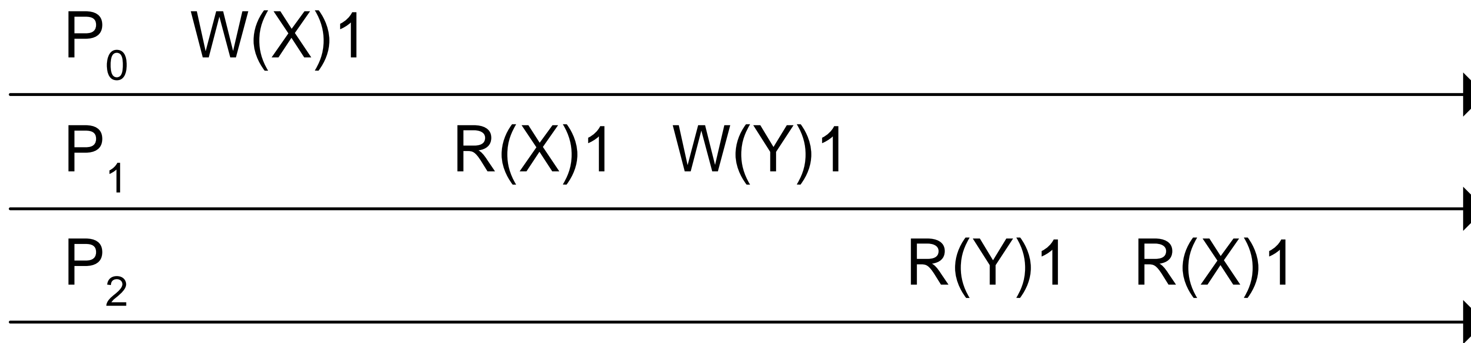


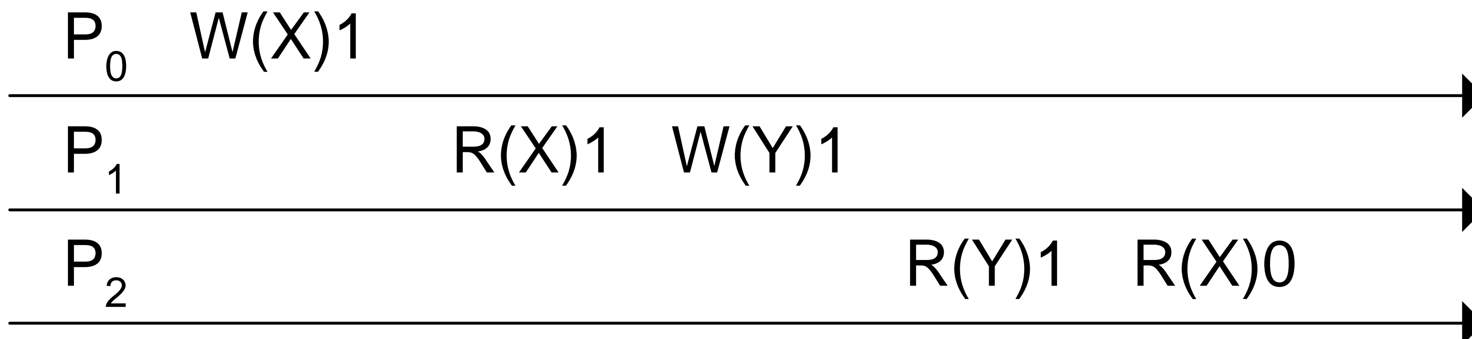
Causal Consistency

Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

- Still fits programmers idea of sequential memory accesses
- Hard to make an efficient implementation

Causal Consistency



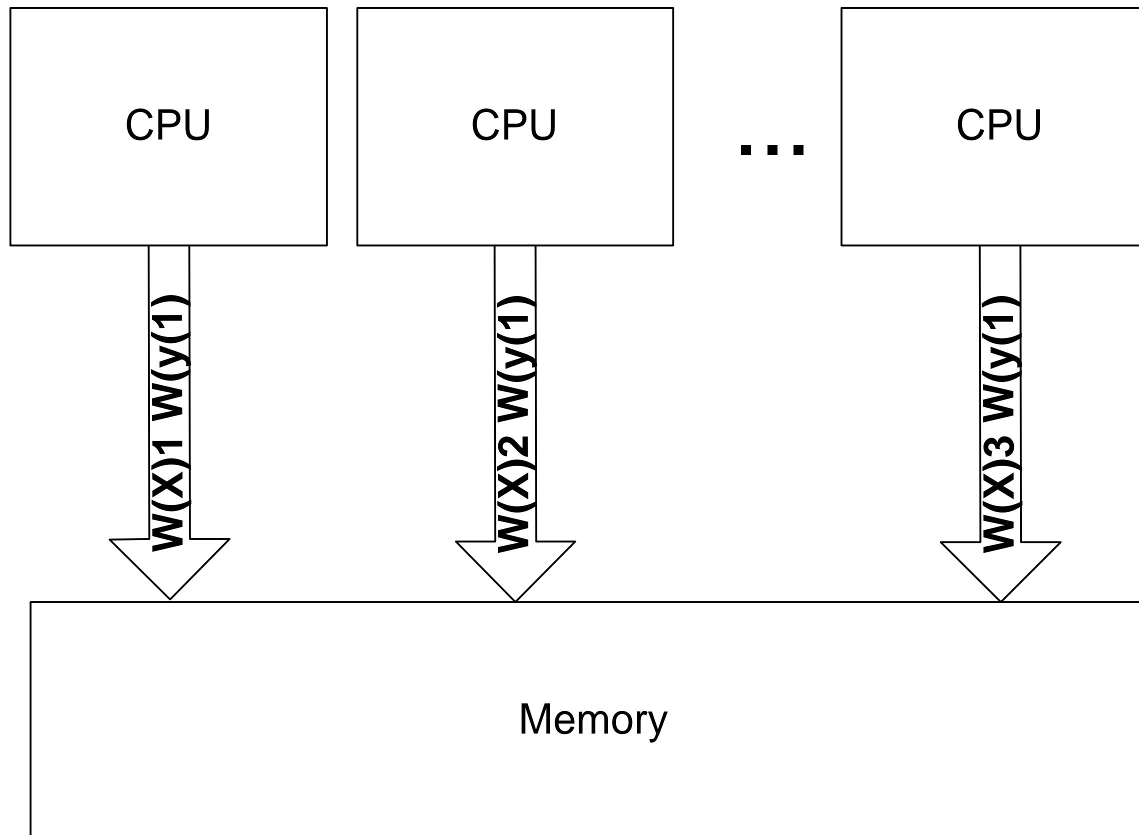


PRAM Consistency

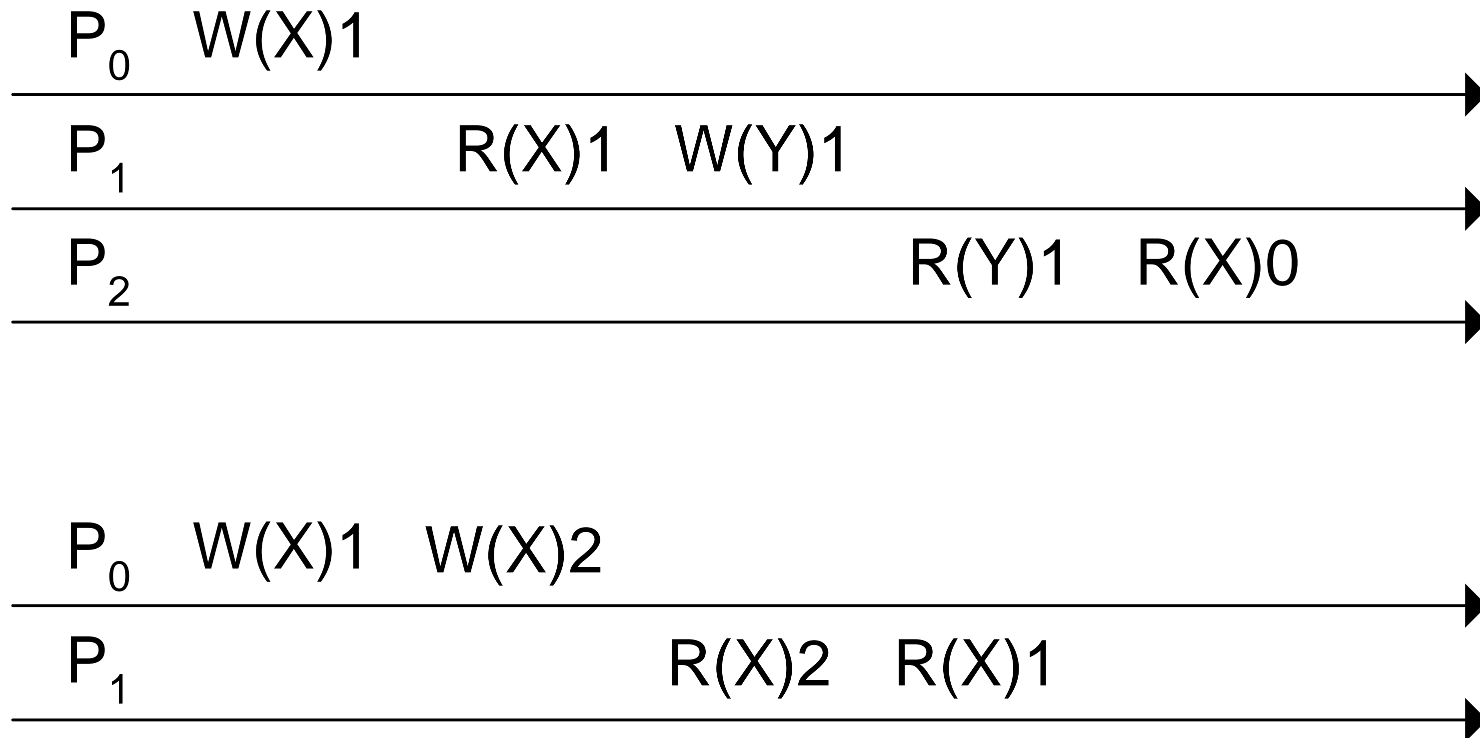
Writes done by a single process are received by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.

- Operations from one node can be grouped for better performance
- Does not comply with ordinary memory conception

PRAM Model



PRAM Consistency



Processor Consistency

1. *Before a read is allowed to perform with respect to any other processor, all previous reads must be performed.*
2. *Before a write is allowed to perform with respect to any other processor, all other accesses (read and write) must be performed.*

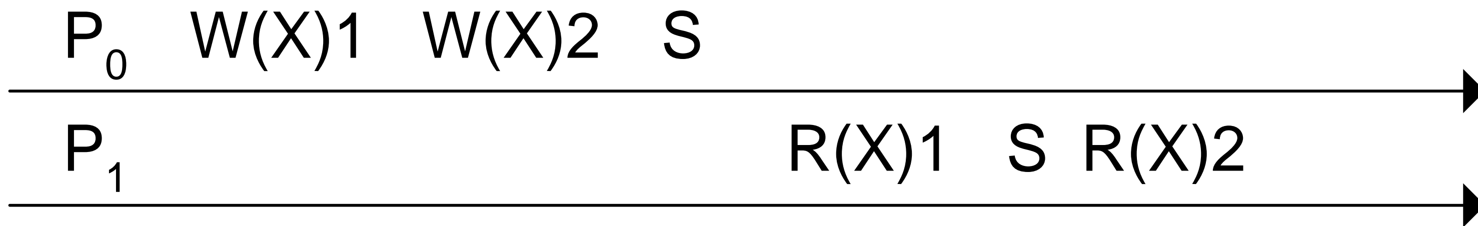
- Slightly stronger than PRAM
- Slightly easier than PRAM

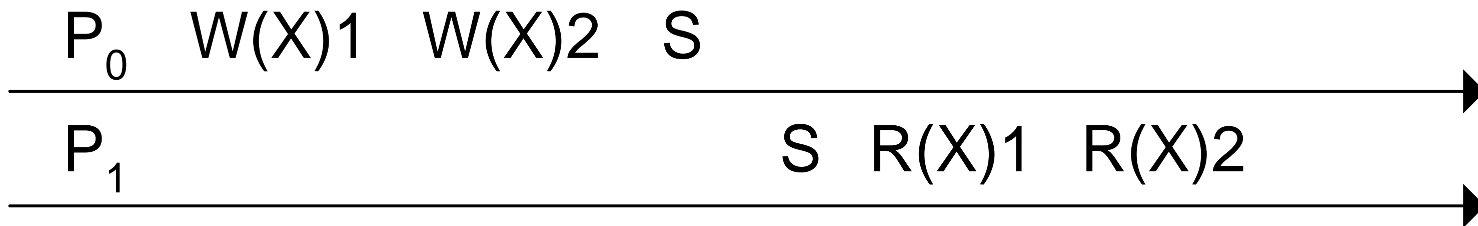
Weak Consistency

1. *Accesses to synchronization variables are sequentially consistent.*
2. *No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.*
3. *No data access (read or write) is allowed to be performed until all previous accesses to synchronization variables have been performed.*

- Synchronization variables are different from ordinary variables
- Lends itself to natural synchronization based parallel programming

Weak Consistency



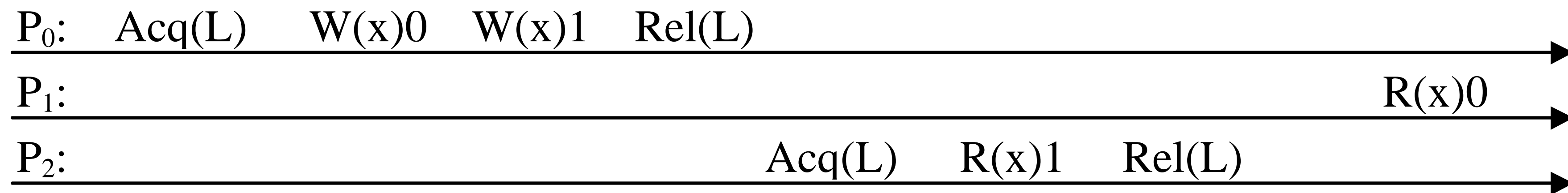


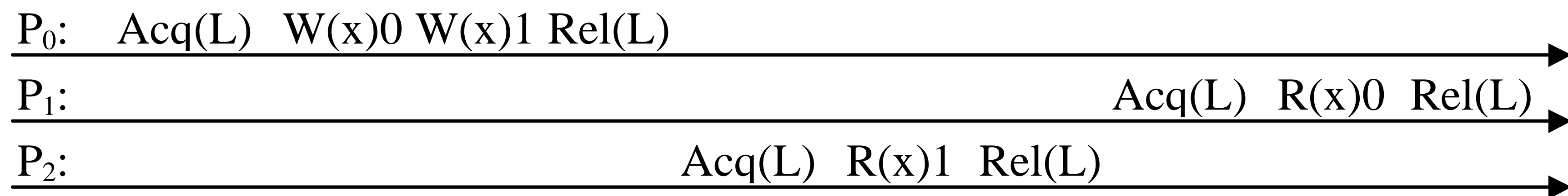
Release Consistency

1. *Before an ordinary access to a shared variable is performed, all previous acquires done by the process must have completed successfully.*
2. *Before a release is allowed to be performed, all previous reads and writes done by the process must have completed.*
3. *The acquire and release accesses must be processor consistent.*

- Synchronization's now differ between Acquire and Release
- Lends itself directly to semaphore synchronized parallel programming

Release Consistency





Lazy Release Consistency

- Differs only slightly from Release Consistency
- Release dependent variables are not propagated at release, but rather at the following acquire
- This allows Release Consistency to be used with smaller granularity

Entry Consistency

1. *An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.*
2. *Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in non-exclusive mode.*
3. *After an exclusive mode access to a synchronization variable has been performed, any other process' next non-exclusive mode access to that synchronization variable may not be performed until it has been performed with respect to that variable's owner.*

- Associates specific synchronization variables with specific data variables

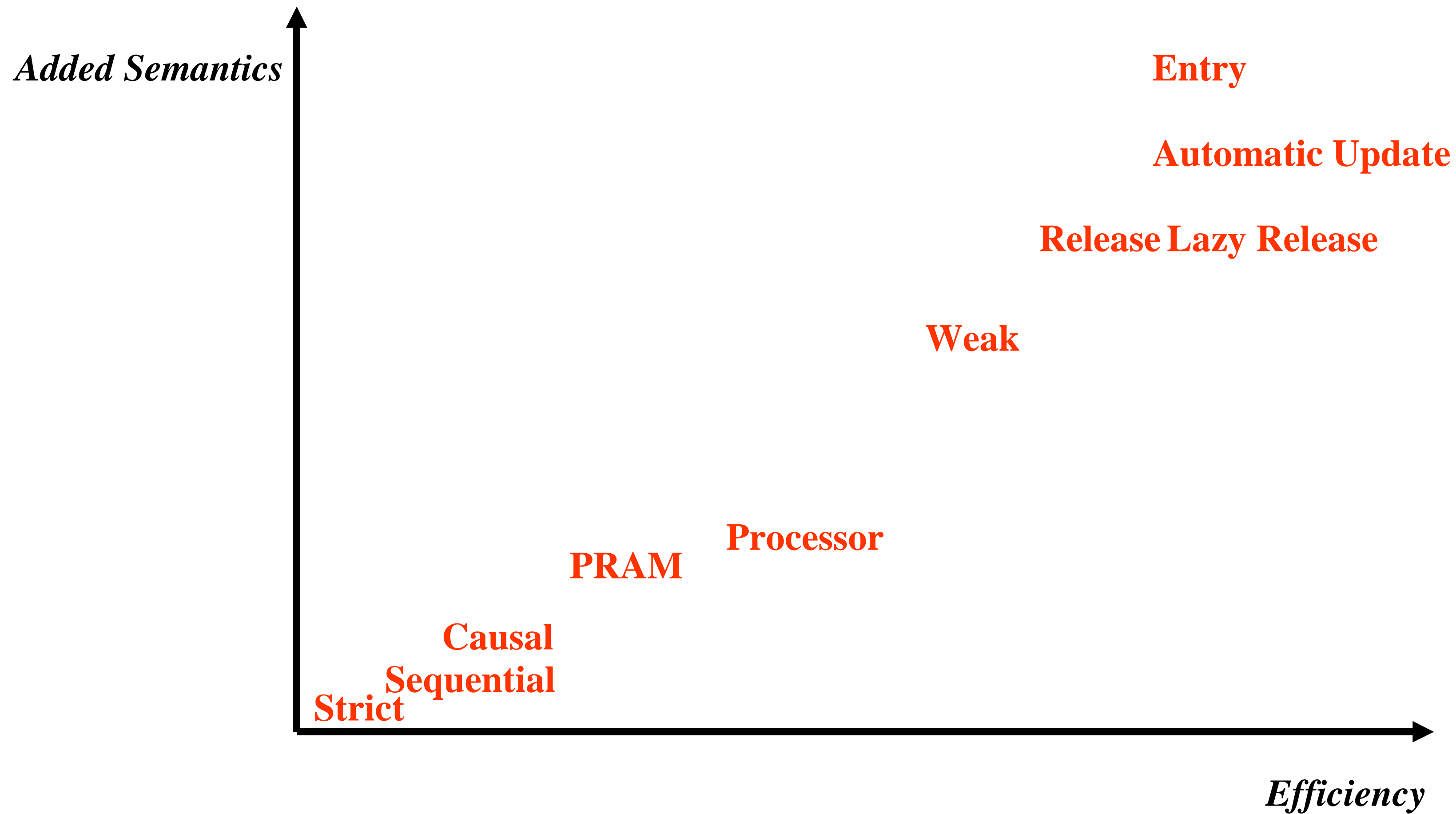
Automatic Update

Automatic update consistency has the same semantics as lazy release consistency, and adding:

> *Before performing a release all automatic updates must be performed.*

- Lends itself to hardware support
- Efficient when two nodes are sharing the same data often

Comparing Consistency models



Working with Relaxed Consistency Models

- Natural tradeoff between efficiency and added work
- Anything beyond Causal Consistency requires the consistency model to be explicitly known
- Compiler knowledge of the consistency model can hide the relaxation from the programmer

CASE Studies

- Intel MP Specification
- Sequent NUMA-Q
- SGI Origin-2000
- Cray T3E

Intel MP Specification

- Shared bus approach
- Sequential Consistency
- MESI Protocol
- Scales to 4 processors directly
- 8 processors by gluing two 4 processor busses together
- UMA

Sequent NUMA-Q

- Based on Intel MP Spec and SCI bus
  - Scalable Coherent Bus
- 4 way SMP boards keept coherent by MESI
- Boards keeps coherent by SCI
- The SCI ring is implemented in one chip
- UMA/NUMA

SGI Origin-2000

- Extention to Stanford DASH
- Provides sequential consistency
- Boards with one R12000 or two R10000 processors (MESI)
- LARGE interface to the world keeps memory coherent
- NUMA (cc-NUMA)

Cray T3E

- Alpha processors with local memory
- May access the memory on other processors
- Memory is note keept coherent in any way!
- Special functionalities for synchronization
- NUMA

Summary

- Relaxing memory consistency is necessary for any system with more than one processor
- Simple relaxation can be hidden
- Strong relaxation can achieve better performance