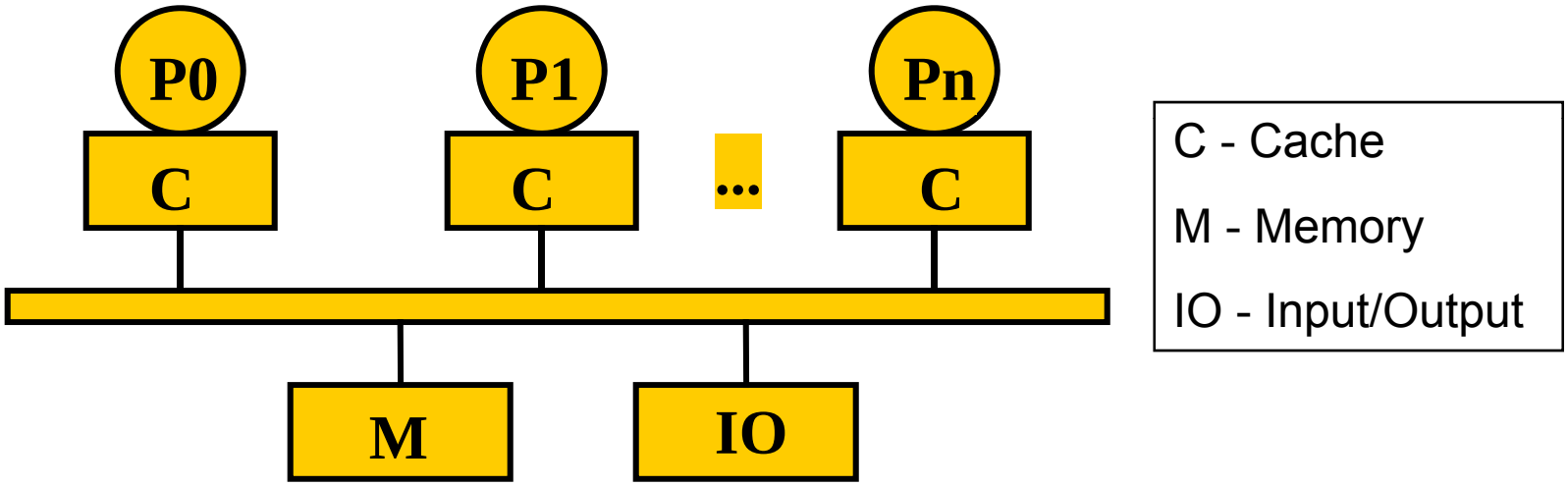
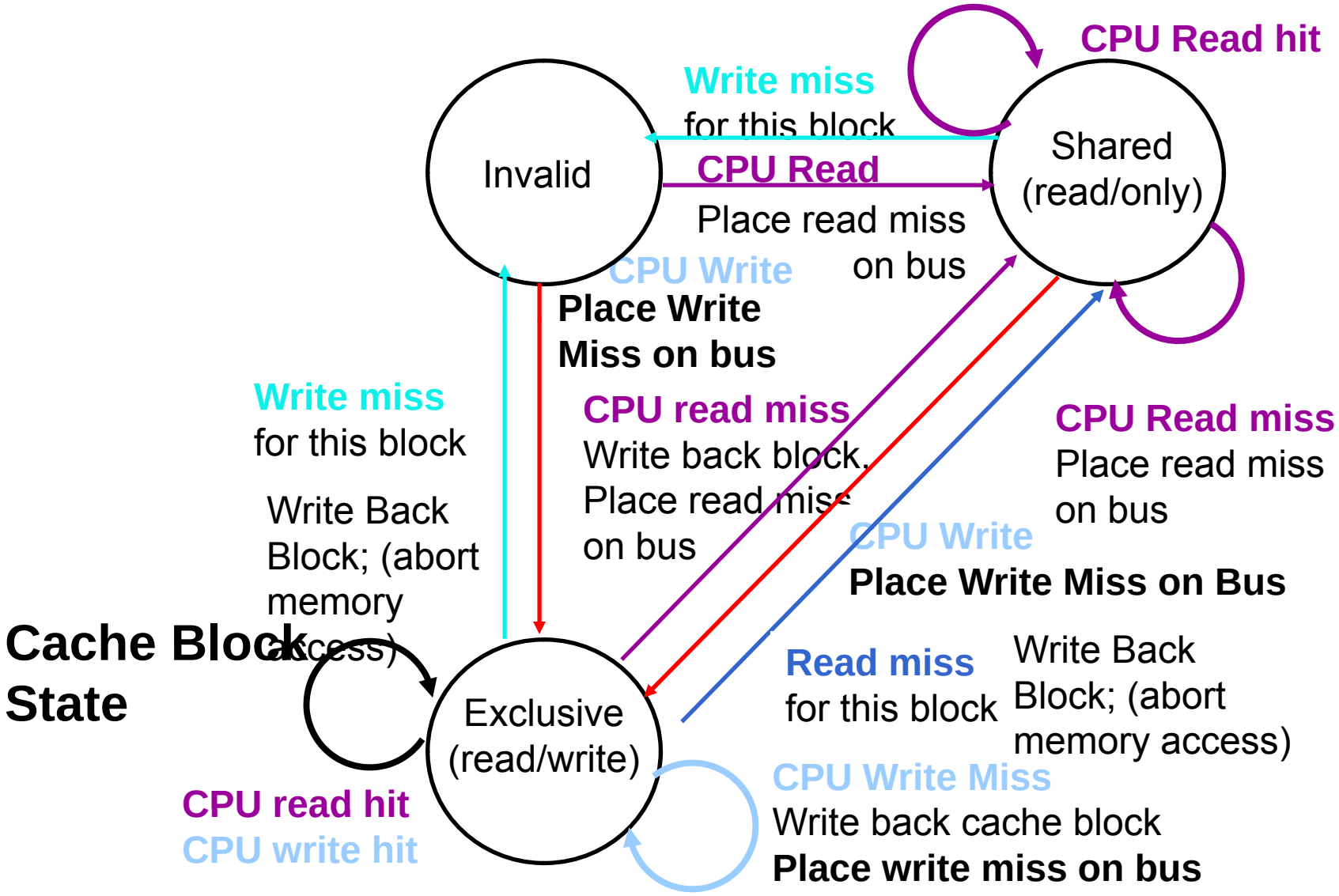
Multiprocessors Directory Protocols

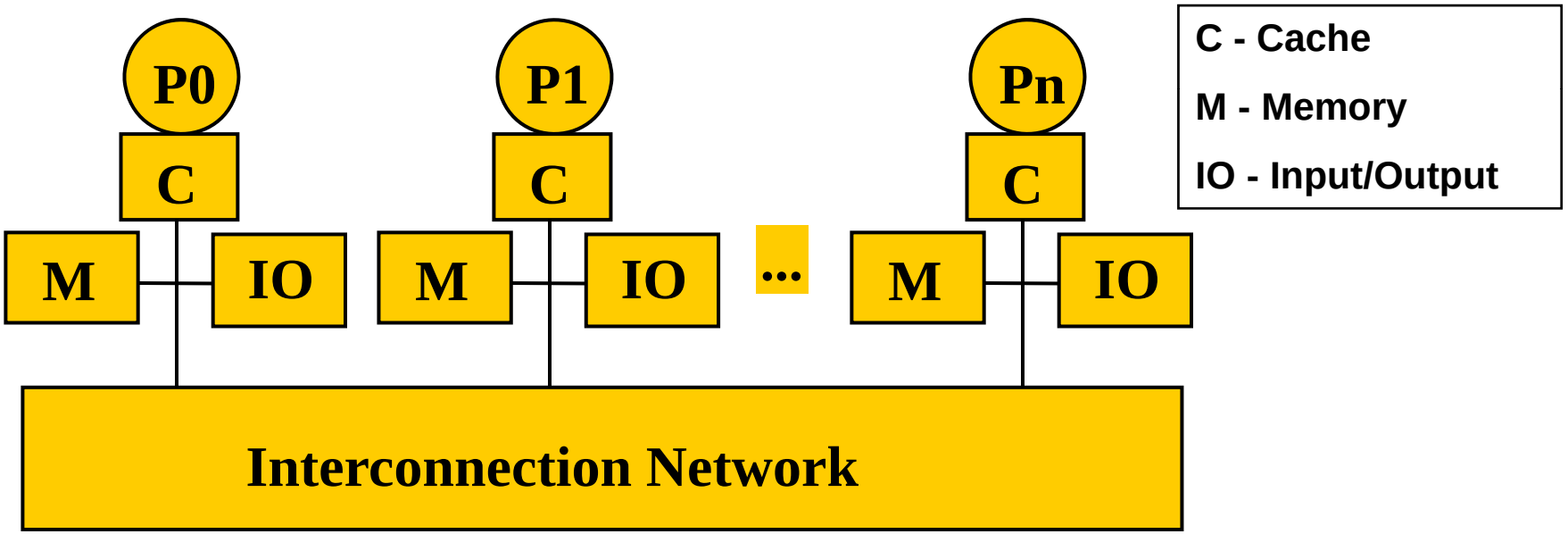
Review: Bus Snooping Topology



Snoopy-Cache State Machine



Distributed Directory MPs



Directory Protocol

- Similar to Snoopy Protocol: Three states
  - Shared: ≥ 1 processors have data, memory up-to-date
  - Uncached (no processor has it; not valid in any cache)
  - Exclusive: 1 processor (owner) has data;
    memory out-of-date
- In addition to cache state, must track which processors have data when in the shared state (usually bit vector, 1 if processor has copy)
- Keep it simple(r):
  - Writes to non-exclusive data
    => write miss
  - Processor blocks until access completes
  - Assume messages received
    and acted upon in order sent

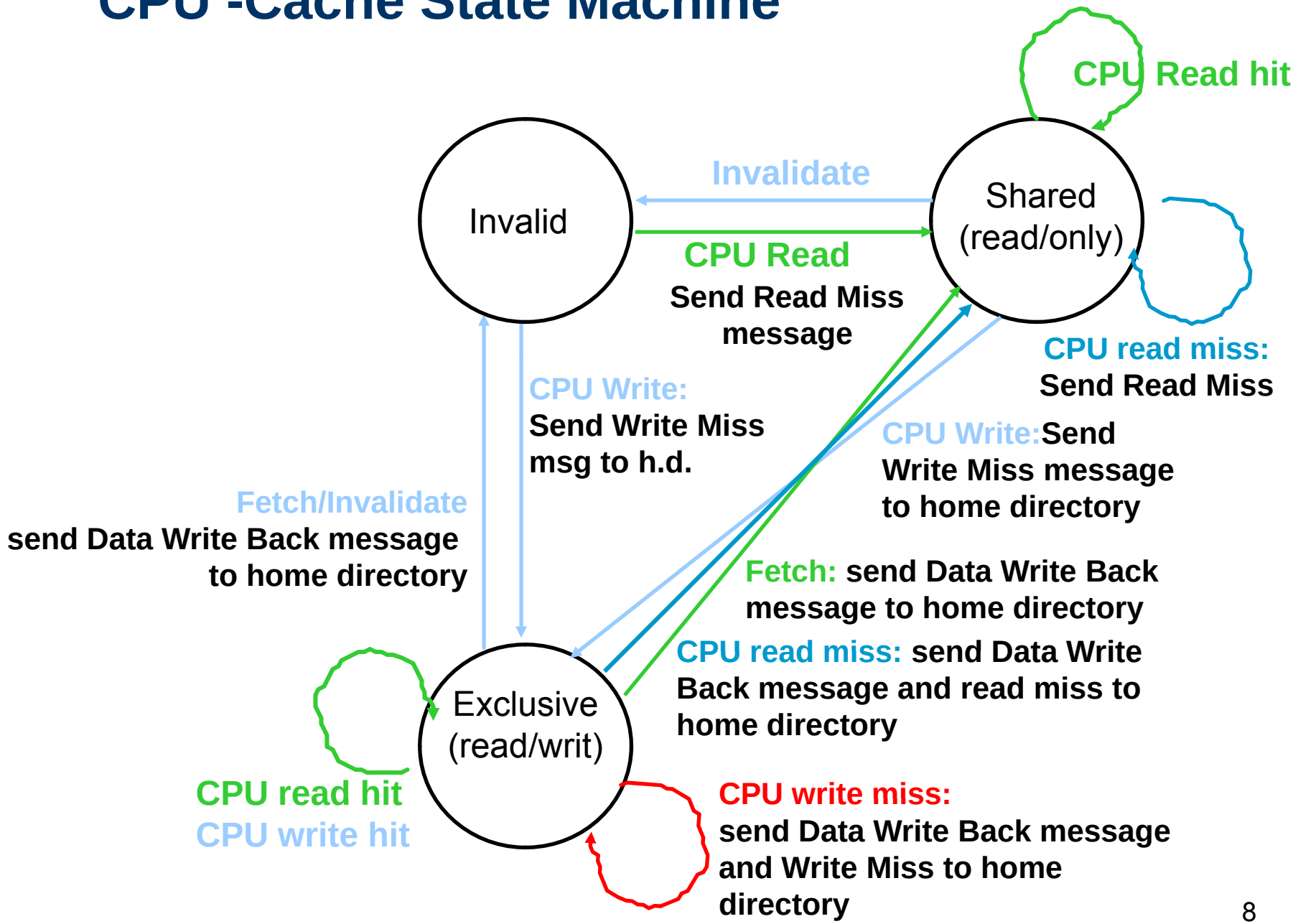
Directory Protocol

- No bus and don't want to broadcast:
  - interconnect no longer single arbitration point
  - all messages have explicit responses
- Terms: typically 3 processors involved
  - Local node where a request originates
  - Home node where the memory location of an address resides
  - Remote node has a copy of a cache block, whether exclusive or shared
- Example messages on next slide:
  P = processor number, A = address

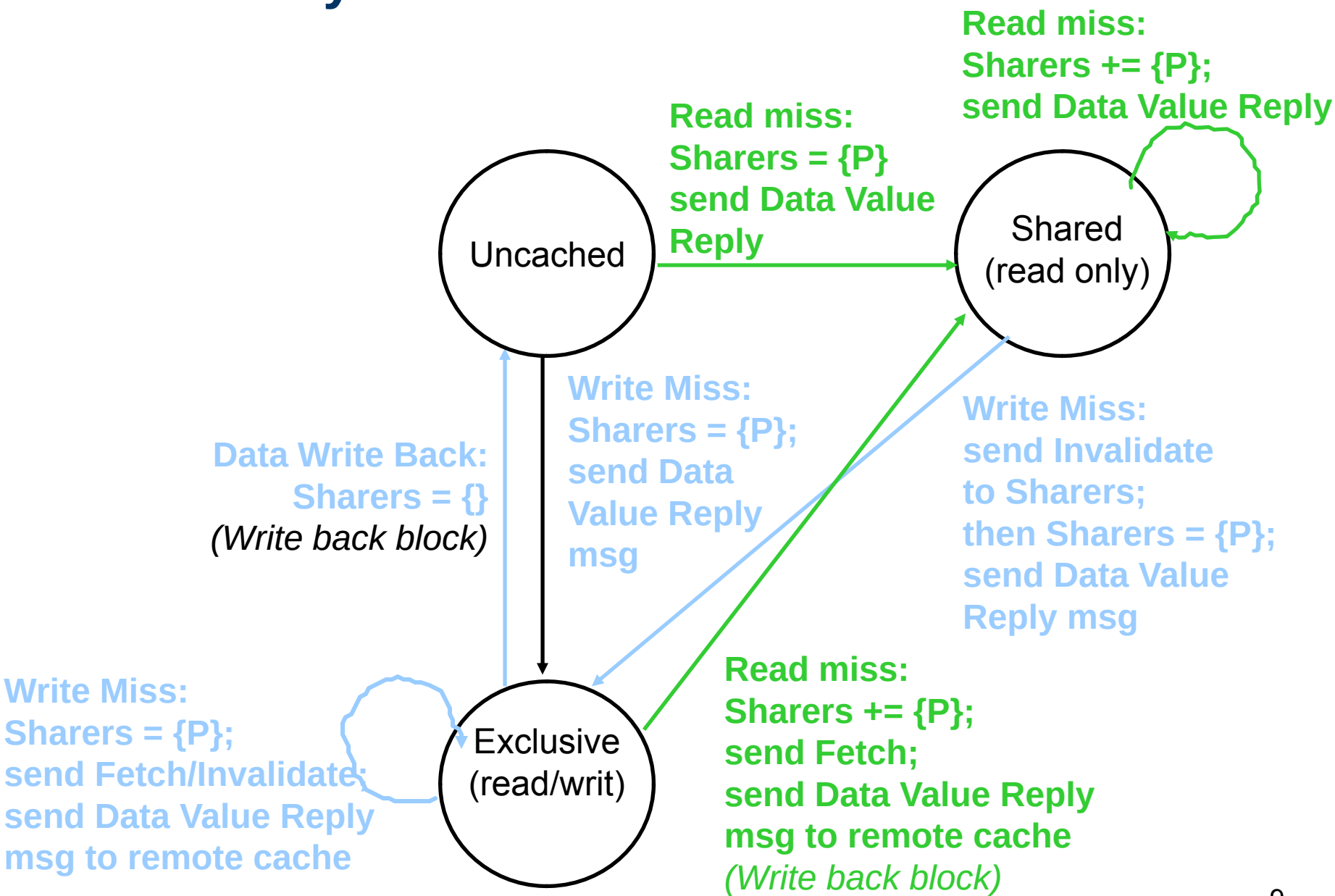
Directory Protocol Messages

Message type	Source	Destination	Msg
Read miss	Local cache	Home directory	P, A
Processor P reads data at address A; make P a read sharer and arrange to send data back			
Write miss	Local cache	Home directory	P, A
Processor P writes data at address A; make P the exclusive owner and arrange to send data back			
Invalidate	Home directory	Remote caches	A
Invalidate a shared copy at address A.			
Fetch	Home directory	Remote cache	A
Fetch the block at address A and send it to its home directory			
Fetch/Invalidate	Home directory	Remote cache	A
Fetch the block at address A and send it to its home directory; invalidate the block in the cache			
Data value reply	Home directory	Local cache	Data
Return a data value from the home memory (read miss response)			
Data write-back	Remote cache	Home directory	A, Data
Write-back a data value for address A (invalidate response)			

CPU -Cache State Machine



Directory State Machine



Parallel Program: An Example

```
/*
 * Title:       Matrix multiplication kernel
 * Author:      Aleksandar Milenkovic, milenkovic@computer.org
 * Date:        November, 1997
 *
 *-------------------------------------------------------------
 *    Command Line Options
 *     -pP: P = number of processors; must be a power of 2.
 *     -nN: N = number of columns (even integers).
 *     -h : Print out command line options.
 *-------------------------------------------------------------
 * */
void main(int argc, char*argv[]) {

     /* Define shared matrix */
   ma = (double **) G_MALLOC(N*sizeof(double *));
   mb = (double **) G_MALLOC(N*sizeof(double *));

   for(i=0; i<N; i++) {
     ma[i] = (double *) G_MALLOC(N*sizeof(double));
     mb[i] = (double *) G_MALLOC(N*sizeof(double));
   };
```

```
/* Initialize the Index */
   Index = 0;

/* Initialize the barriers and the lock */
   LOCKINIT(indexLock)
   BARINIT(bar_fin)

   /*  read/initialize data   */
   ...
   /* do matrix multiplication in parallel a=a*b
      */
   /* Create the slave processes. */
   for (i = 0; i < numProcs-1; i++)
      CREATE(SlaveStart)

   /* Make the master do slave work so we
      don't waste a processor */
   SlaveStart();


      ...

}
```

Parallel Program: An Example

```
/*====== SlaveStart ================*/
/* This is the routine that each processor will be
     executing in parallel */
void SlaveStart() {
   int myIndex, i, j, k, begin, end;
   double tmp;

   LOCK(indexLock); /* enter the critical section
     */
    myIndex = Index; /* read your ID */
    ++Index;              /* increment it, so the next
     will operate on ID+1 */
   UNLOCK(indexLock); /* leave the critical
     section */

   /* Initialize begin and end */
   begin = (N/numProcs)*myIndex;
   end = (N/numProcs)*(myIndex+1);
```

```
/* the main body of a thread */

    for(i=begin; i<end; i++) {

      for(j=0; j<N; j++) {
        tmp=0.0;
        for(k=0; k<N; k++) {
          tmp = tmp + ma[i][k]*mb[k][j];
        }
        ma[i][j] = tmp;
      }
    }

    BARRIER(bar_fin, numProcs);

}
```

Synchronization

- Why Synchronize? Need to know when it is safe for different processes to use shared data
- Issues for Synchronization:
  - Uninterruptable instruction to fetch and update memory (atomic operation);
  - User level synchronization operation using this primitive;
  - For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization

Uninterruptable Instruction to Fetch and Update Memory

- Atomic exchange: interchange a value in a register for a value in memory
  - 0 => synchronization variable is free
  - 1 => synchronization variable is locked and unavailable
  - Set register to 1 & swap
  - New value in register determines success in getting lock 0 if you succeeded in setting the lock (you were first)
    1 if other processor had already claimed access
  - Key is that exchange operation is indivisible
- Test-and-set: tests a value and sets it if the value passes the test
- Fetch-and-increment: it returns the value of a memory location and atomically increments it
  - 0 => synchronization variable is free

Lock&Unlock: Test&Set

```
/* Test&Set */
==============
               loadi R2, #1
lockit:        exch R2, location /* atomic operation*/
               bnez R2, lockit   /* test*/


unlock:        store location, #0 /* free the lock (write 0) */
```

Lock&Unlock: Test and Test&Set

```
/* Test and Test&Set */
======================

lockit:  load R2, location /* read lock varijable */
         bnz R2, lockit   /* check value  */
         loadi R2, #1
         exch R2, location /* atomic operation */
         bnz reg, lockit   /* if lock is not acquired, repeat */


unlock:  store location, #0 /* free the lock (write 0) */
```

Lock&Unlock: Test and Test&Set

```
/* Load-linked and Store-Conditional */
=====================================
lockit:  ll R2, location /* load-linked read  */
         bnz R2, lockit  /* if busy, try again */
         load R2, #1
         sc location, R2 /* conditional store */
         beqz R2, lockit /* if sc unsuccessful, try again */


unlock: store location, #0 /* store 0 */
```

Uninterruptable Instruction to Fetch and Update Memory

- Hard to have read & write in 1 instruction: use 2 instead
- Load linked (or load locked) + store conditional
  - Load linked returns the initial value
  - Store conditional returns 1 if it succeeds (no other store to same memory location since preceeding load) and 0 otherwise
- Example doing atomic swap with LL & SC:

```
try:  mov   R3,R4          ; mov exchange value
      ll    R2,0(R1); load linked
      sc    R3,0(R1); store conditional (returns 1, if Ok)
      beqz  R3,try         ; branch store fails (R3 = 0)
      mov   R4,R2          ; put load value in R4
```

- Example doing fetch & increment with LL & SC:

```
try:  ll    R2,0(R1); load linked
      addi  R2,R2,#1       ; increment (OK if reg-reg)
      sc    R2,0(R1)       ; store conditional
      beqz  R2,try         ; branch store fails (R2 = 0)
```

User Level Synchronization—Operation Using this Primitive

- Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock

```
                li      R2,#1
lockit:         exch    R2,0(R1)        ;atomic exchange
                bnez    R2,lockit       ;already locked?
```

- What about MP with cache coherency?
  - Want to spin on cache copy to avoid full memory latency
  - Likely to get cache hits for such variables
- Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic
- Solution: start by simply repeatedly reading the variable; when it changes, then try exchange ("test and test&set"):

```
try:            li      R2,#1
lockit:         lw      R3,0(R1)        ;load var
                bnez    R3,lockit       ;not free=>spin
                exch    R2,0(R1)        ;atomic exchange
                bnez    R2,try   ;already locked?
```

Barrier Implementation

```
struct BarrierStruct {
 LOCKDEC(counterlock);
 LOCKDEC(sleeplock);
 int sleepers;
};
...
#define BARDEC(B) struct BarrierStruct B;
#define BARINIT(B) sys_barrier_init(&B);
#define BARRIER(B,N) sys_barrier(&B, N);
```

Barrier Implementation (cont'd)

```
void sys_barrier(struct BarrierStruct *B, int N) {
  LOCK(B->counterlock)
      (B->sleepers)++;

      if (B->sleepers < N ) {
        UNLOCK(B->counterlock)

        LOCK(B->sleeplock)
            B->sleepers--;
            if(B->sleepers > 0) UNLOCK(B->sleeplock)
            else UNLOCK(B->counterlock)
      }
      else {
        B->sleepers--;
        if(B->sleepers > 0) UNLOCK(B->sleeplock)
        else UNLOCK(B->counterlock)
      }
}
```

Another MP Issue: Memory Consistency Models

- What is consistency? When must a processor see the new value? e.g., seems that

```
P1:  A = 0;                    P2:  B = 0;
     .....                          .....
     A = 1;                         B = 1;
L1:  if (B == 0) ...           L2:  if (A == 0) ...
```

- Impossible for both if statements L1 & L2 to be true?
  - What if write invalidate is delayed & processor continues?
- Memory consistency models: what are the rules for such cases?
- Sequential consistency: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved => assignments before ifs above
  - SC: delay all memory accesses until all invalidates done

Memory Consistency Model

- Schemes faster execution to sequential consistency
- Not really an issue for most programs;
  they are synchronized
  - A program is synchronized if all access to shared data are ordered by synchronization operations

```
write (x)
...
release (s) {unlock}
...
acquire (s) {lock}
...
read(x)
```

- Only those programs willing to be nondeterministic are not synchronized: "data race": outcome f(proc. speed)
- Several Relaxed Models for Memory Consistency since most programs are synchronized; characterized by their attitude towards: RAR, WAR, RAW, WAW
  to different addresses

Summary

- Caches contain all information on state of cached memory blocks
- Snooping and Directory Protocols similar; bus makes snooping easier because of broadcast (snooping => uniform memory access)
- Directory has extra data structure to keep track of state of all cache blocks
- Distributing directory
  => scalable shared address multiprocessor
  => Cache coherent, Non uniform memory access

Achieving High Performance in Bus-Based SMPs

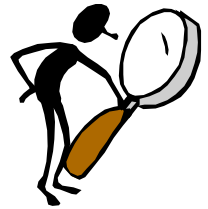
A. Milenkovic, "Achieving High Performance in Bus-Based Shared Memory Multiprocessors," *IEEE Concurrency* , Vol. 8, No. 3, July-September 2000, pp. 36-44.

Partially funded by Encore, Florida, done at the School of Electrical Engineering, University of Belgrade (1997/1999)

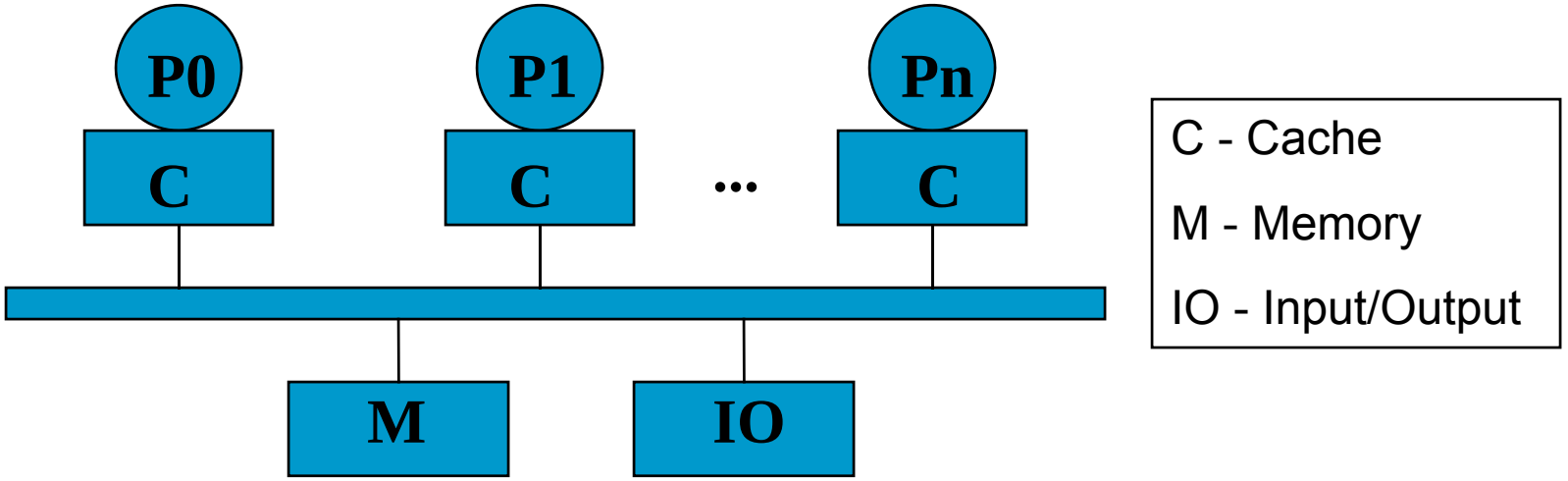
Outline

- Introduction
- Existing Solutions
- Proposed Solution: Cache Injection
- Experimental Methodology
- Results
- Conclusions

Introduction



- Bus-based SMPs:
  current situation and challenges



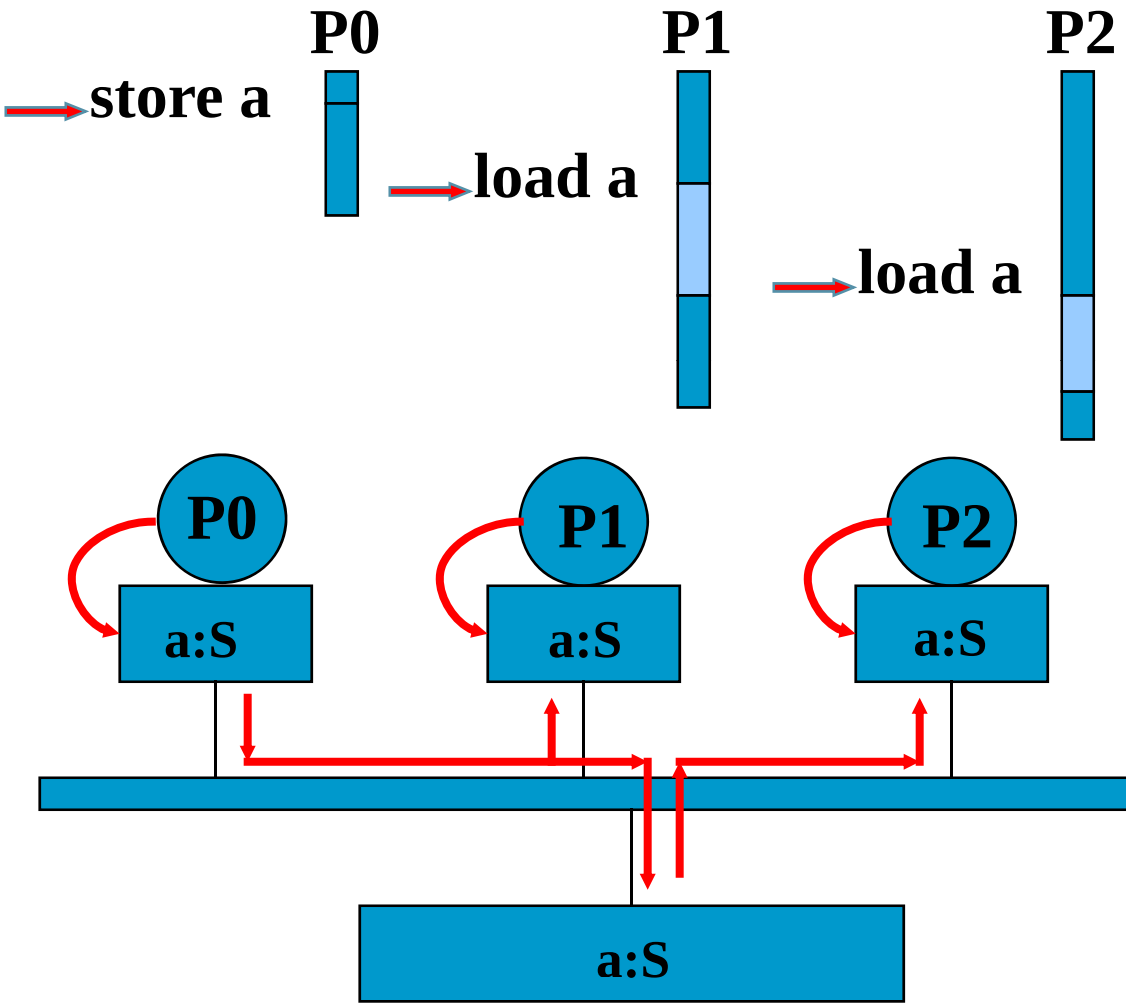
Introduction

- Cache misses and bus traffic are key obstacles
  to achieving high performance due to
  - widening speed gap between processor and memory
  - high contention on the bus
  - data sharing in parallel programs
- Write miss latencies: relaxed memory consistency models
- Latency of read misses remains
- Techniques to reduce
  the number of read misses

Existing solutions

- Cache Prefetching
- Read Snarfing
- Software-controlled updating

An Example



0. Initial state

1. P0: store a

2. P1: load a

3. P2: load a

M - Modified

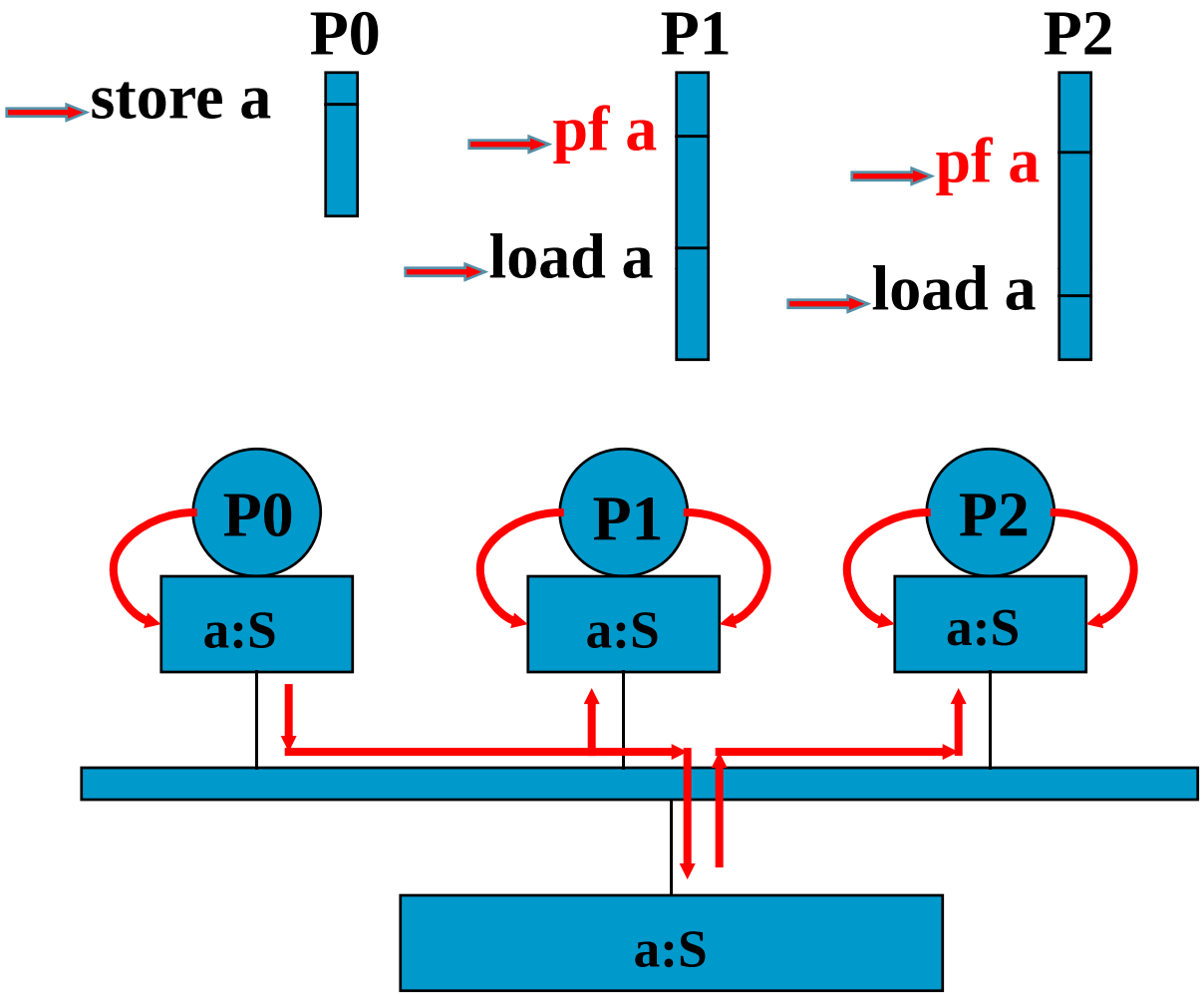
S - Shared

I – Invalid

- – Not present

Cache Prefetching

Exisiting solutions



0. Initial state

1. P0: store a

2. P1: pf a

3. P2: pf a

4. P1: load a

5. P2: load a

pf - prefetch

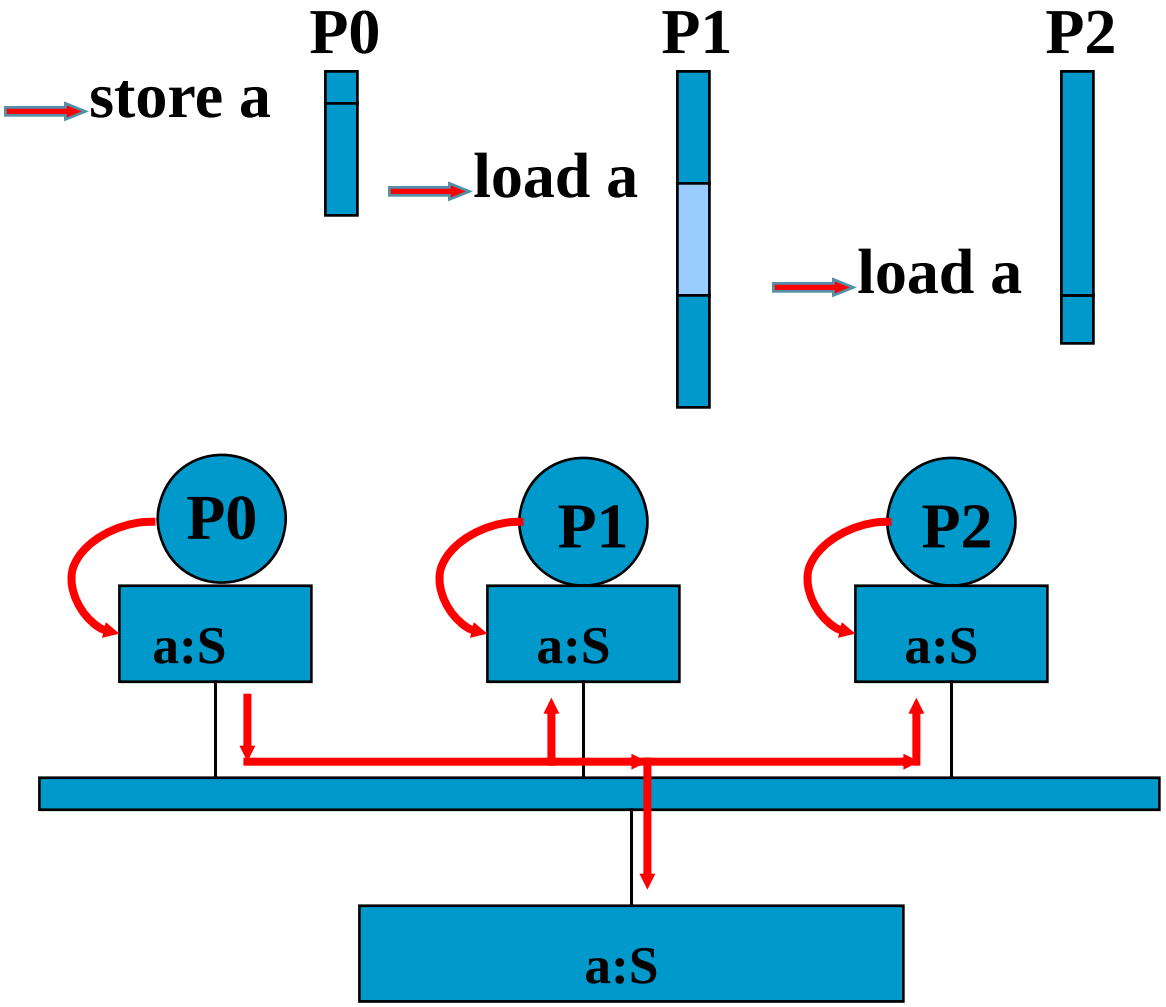
Cache Prefetching

Exisiting solutions

- Reduces all kind of misses (cold, coh., repl.)
- Hardware support: prefetch instructions + buffering of prefetches
- Compiler support
  [T. Mowry, 1994; T. Mowry and C. Luk, 1997]
- Potential of cache prefetching in BB SMPs [D. Tullsen, S. Eggers, 1995]

Read Snarfing

Exisitng solutions



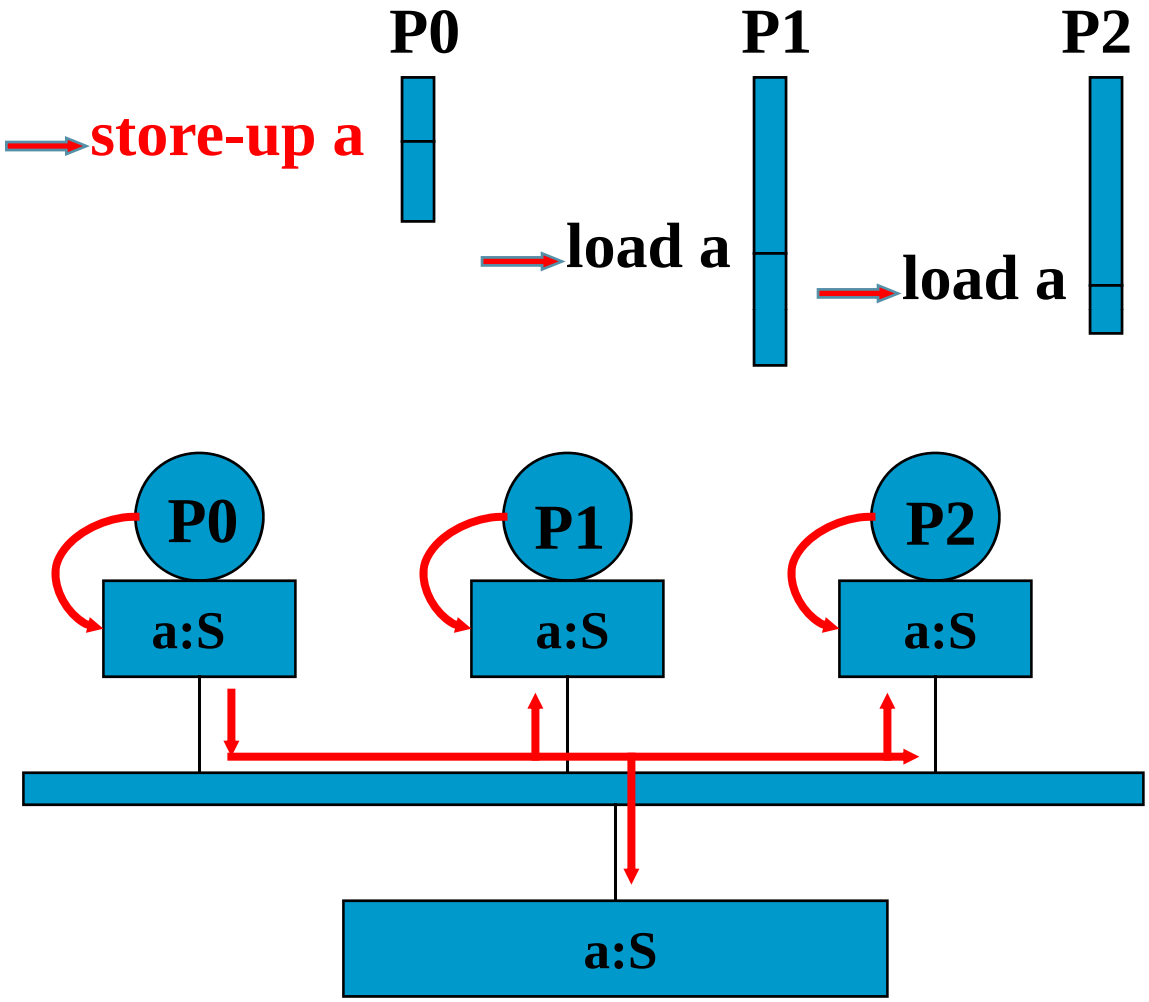
0. Initial state
1. P0: store a
2. P1: load a
3. P2: load a

Read Snarfing

Exisiting solutions

- Reduces only coherence misses
- Hardware support: negligible
- Compiler support: none
- Performance evaluation
  [C. Andersen and J.-L. Baer, 1995]
- Drawbacks

Software-controlled updating *Exisiting solutions*

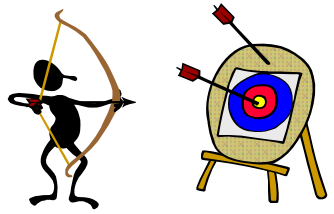


0. Initial state
1. P0: store-up a
2. P1: load a
3. P2: load a

Software-controlled updating *Exisiting solutions*

- Reduces only coherence misses
- Hardware support
- Compiler support:
  [J. Skeppstedt, P. Stenstrom, 1994]
- Performance evaluation
  [F. Dahlgren, J. Skeppstedt, P. Stenstrom, 1995]
- Drawbacks

CACHE INJECTION



- Motivation
- Definition and programming model
- Implementation
- Primena na prave deljene podatke (PDP)
- Primena na sinhro-primitive (SP)
- Hardverska podrška
- Softverska podrška

Motivation

- Overcome some of the other techniques' shortcomings such as
  - minor effectiveness of cache prefetching in reducing coherence cache misses
  - minor effectiveness of read snarfing and software-controlled updating in SMPs with relatively small private caches
  - high contention on the bus in cache prefetching and software-controlled updating

Definition

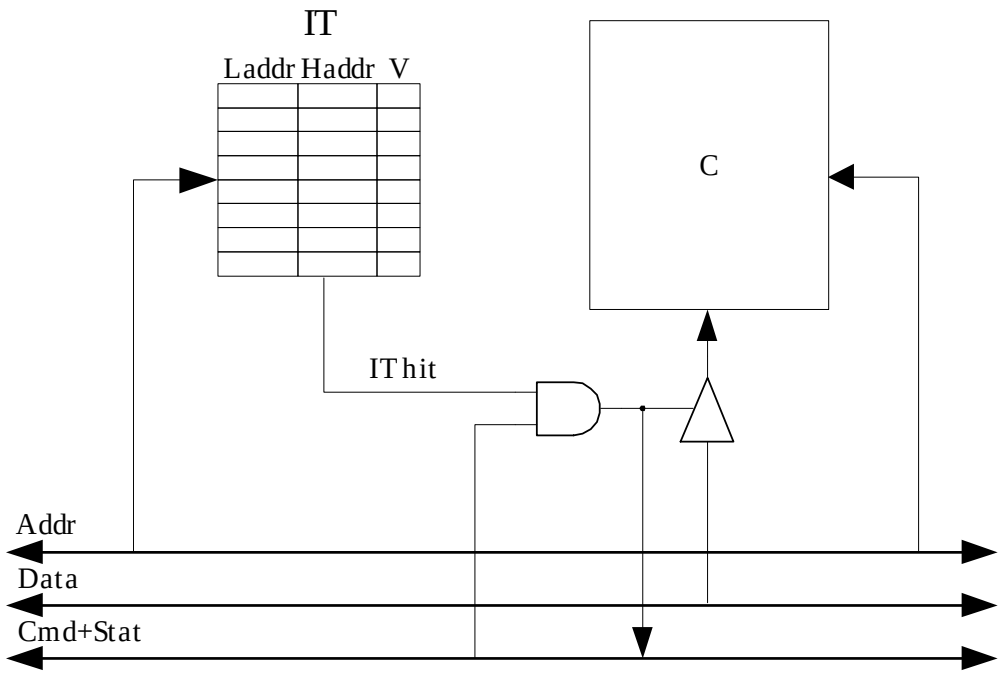
- Consumers predicts their future needs for shared data by executing an openWin instruction
- OpenWin Laddr, Haddr
- Injection table
- Hit in injection table $\Rightarrow$ cache injection

Definition

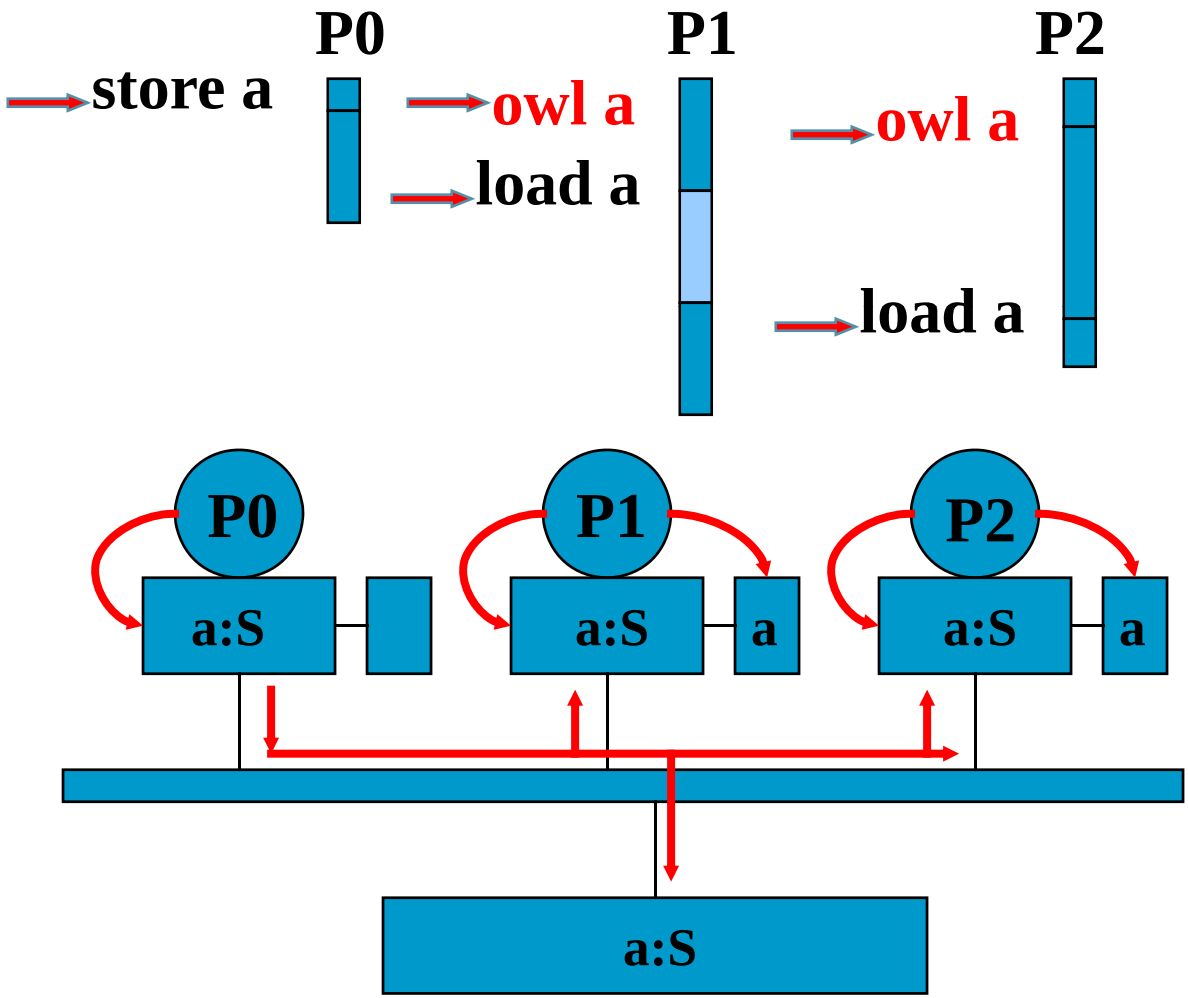
- Injection on first read
  - Applicable for read only shared data and 1-Producer-Multiple-Consumers sharing pattern
  - Each consumer initializes its local injection table
- Injection on Update
  - Applicable for 1-Producer-1-Consumer and 1-Producer-Multiple-Consumers sharing patterns or migratory sharing pattern
  - Each consumer initializes its local injection table
  - After data production, the data producer initiates an update bus transaction by executing an update or store-update instruction

Implementation

- OpenWin(Laddr, Haddr)
  - OWL(Laddr)
  - OWH(Haddr)
- CloseWin(Laddr)
- Update(A)
- StoreUpdate(A)

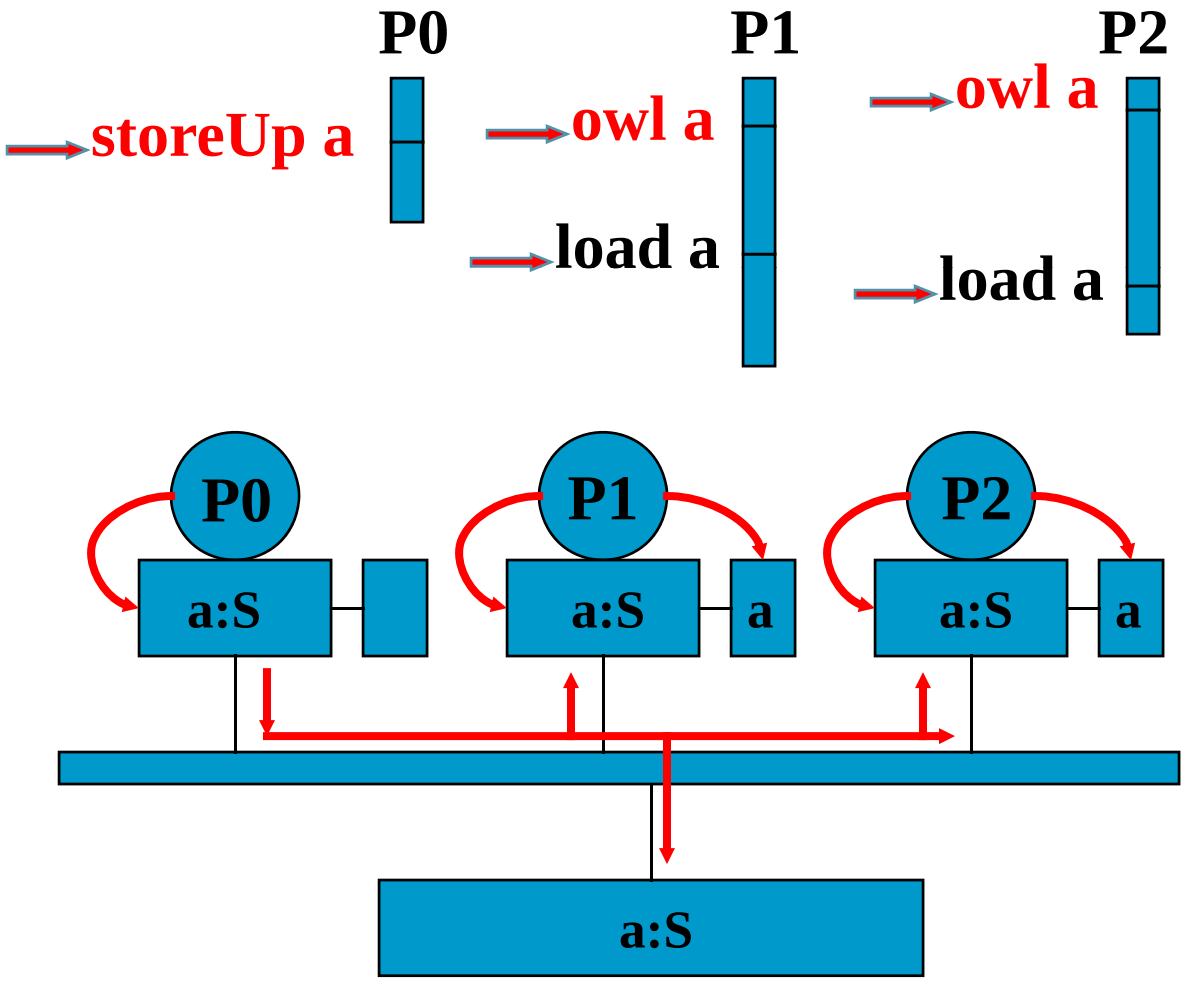


Injection on first read



0. Initial state
1. P0: store a
2. P1: owl a
3. P2: owl a
4. P1: load a
5. P2: load a

Injection on update



0. Initial state
1. P2: owl a
2. P1: owl a
3. P0: storeUp a
4. P1: load a
5. P2: load a

Injection for true shared data: PC

```
shared double A[NumProcs][100];
OpenWin(A[0][0], A[NumProcs-1][99]);
for(t=0; t<t_max; t++) {
   local double myVal =0.0
   for(p=0; p<NumProcs; p++) {
      for(i=0; i<100; i++)
         myVal+=foo(A[p][i], MyProcNum];
   }
   barrier(B, NumProcs);
   for(i=0; i<100; i++)
      A[MyProcNum][i]+=myVal;
   barrier(B, NumProcs);
}
CloseWin(A[0][0]);
```

Injection for true shared data: PC

	Base	Pref-Ex	Forw	Forw+ Pref-Ex	InjectFR	InjectWB	Inject+ Pref-Ex
T_{stall} [x10^3pclk]	304,5	≈0	34,5	≈0	22,5	4,5	2
Traffic [x10^6B]	2,576	2,5768	2,576	2,5768	0,2017	0,2017	0,2017
Code Complexity	0	>>	>>	>>	0	>	>>

Injection for *Lock* SP

- Base

```
lock(L);
   critical-section(d);
unlock(L);
```

- Inject

```
OpenWin(L);
lock(L);
   critical-section(d);
unlock(L);
CloseWin(L);
```

Injection for *Lock* SP

- Traffic
  - Test&exch Lock implementation

	RdC	RdXC	InvC	WbC
Base	**N^2**	**N(N+1)/2**	**N**	**-**
InjectFR	**2N-1**	**N(N+1)/2**	**N**	**-**
InjectWb	**1**	**-**	**N(N+1)/3**	**N(N+1)/3**

  - LL-SC Lock implementation

	RdC	RdXC	InvC	WbC
Base	**N^2**	**-**	**2N-1**	**-**
InjectFR	**2N-1**	**-**	**2N-1**	**-**
InjectWb	**1**	**-**	**2N-1**	**2N-1**

N – Number of processors;
RdC - Read; RdXC - ReadExclusive;
InvC - Invalidate; WbC - WriteBack

Injection for *Barrier* SP

- Base barrier implementation

```
struct BarrierStruct {
        LOCKDEC(counterlock); //semafor dolazaka
        LOCKDEC(sleeplock);   //semafor odlazaka
        int sleepers;};       //broj blokiranih
#define BARDEC(B)   struct BarrierStruct B;
#define BARINIT(B)  sys_barrier_init(&B);
#define BARRIER(B,N)sys_barrier(&B, N);
```

- Injection barrier implementation

```
BARDEC(B) BARINIT(B)
OpenWin(B->counterlock, B->sleepers);
....
BARRIER(B, N); ...;
BARRIER(B, N); ...;
CloseWin(B->counterlock);
```

Hardware support

- Injection table
- Instructions: OWL, OWH, CWL (Update, StoreUpdate)
- Injection cycle in cache controller

Software support

- Compiler and/or programmer are responsible for inserting instructions
- Sinhro
- True shared data

Experimental Methodology

- Limes (Linux Memory Simulator) – a tool for program-driven simulation of shared memory multiprocessors
- Workload
- Modeled Architecture
- Experiments

Workload

- Sinhronization kernels (SP)
  - LTEST (I=1000, C=200/20pclk, D=300pclk)
  - BTEST (I=100, Tmin=Tmax=40)
- Test applications (SP+PDP)
  - PC (I=20, M=128, N=128)
  - MM (M=128, N=128)
  - Jacobi (I=20, M=128, N=128)
- Applications from SPLASH-2
  - Radix (N=128K, radix=256, range={0-231})
  - LU (256x256, b=8)
  - FFT (N=216)
  - Ocean (130x130)

Modeled Architecture

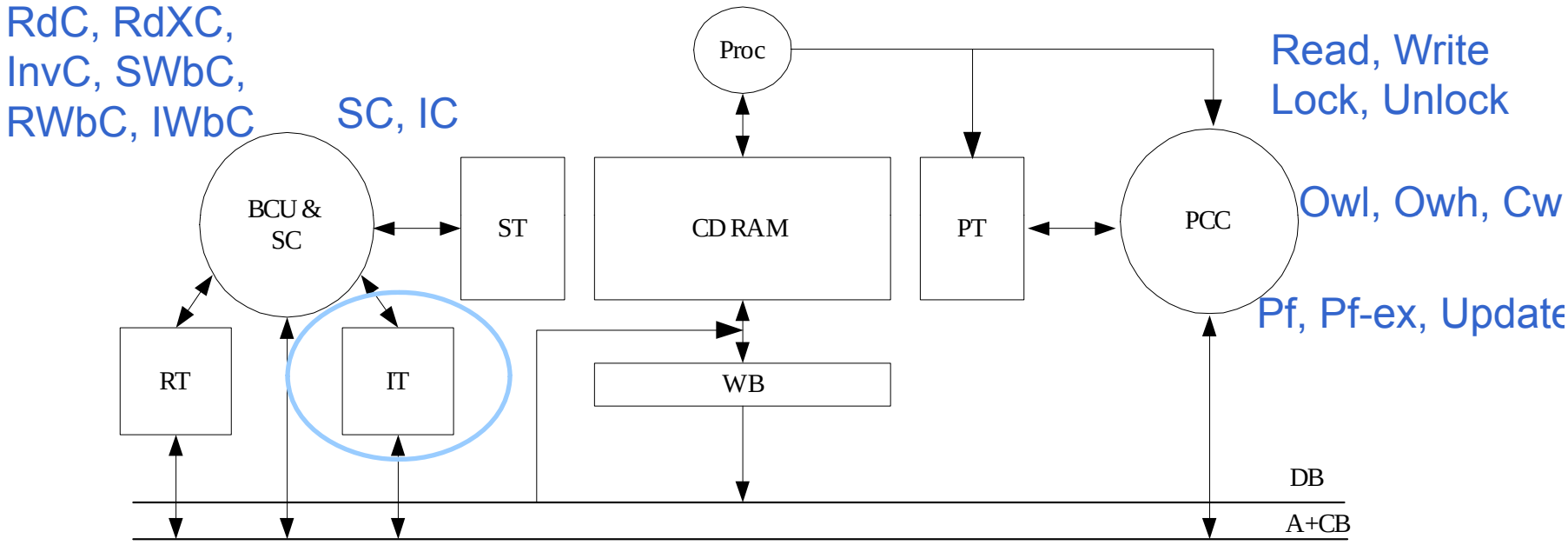
Experimental methodology

- SMP with 16 processors,
  Illinois cache coherence protocol
- Cache: first level 2-way set associative,
  128 entry injection table, 32B cache line size
- Processor model: single-issue, in-order, single cycle per instruction, blocking read misses, cache hit is solved without penalty
- Bus: split-transactions, round-robin arbitration, 64 bits data bus width, 2pclk snoop cycle, 20pclk memory read cycle

Modeled Architecture

Experimental methodology

Cache Controller

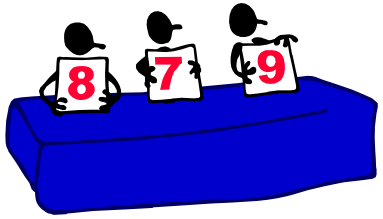


PCC - Processor Cache Controller
BCU&SC - Bus Control Unit&Snoop Controller
PT - Processor Tag, ST - Snoop Tag, WB - WriteBack Buffer
RT - Request Table, IT - Injection Table, CD - Cache Data
DB - Data Bus, A+CB - Address+Control Bus

Experiments

- Execution time
- Number of read misses and the bus traffic for
  B – base system
  S – read snarfing
  U – software-controlled updating
  I – cache injection

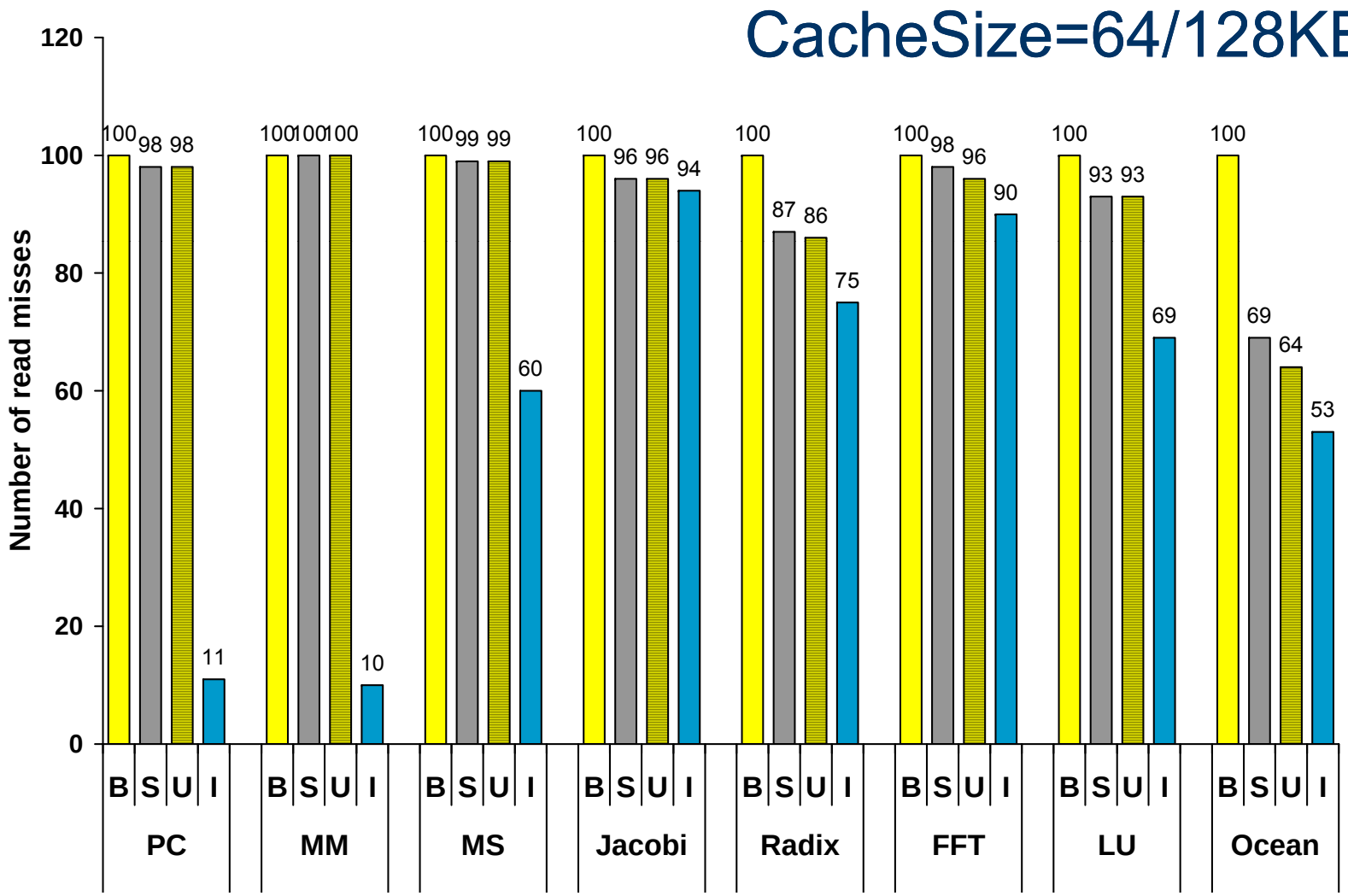
Results



- Number of read misses
  normalized to the base system in the system when the caches are relatively small and relatively large
- Bus traffic
  normalized to the base system in the system when the caches are relatively small and relatively large

Number of read misses

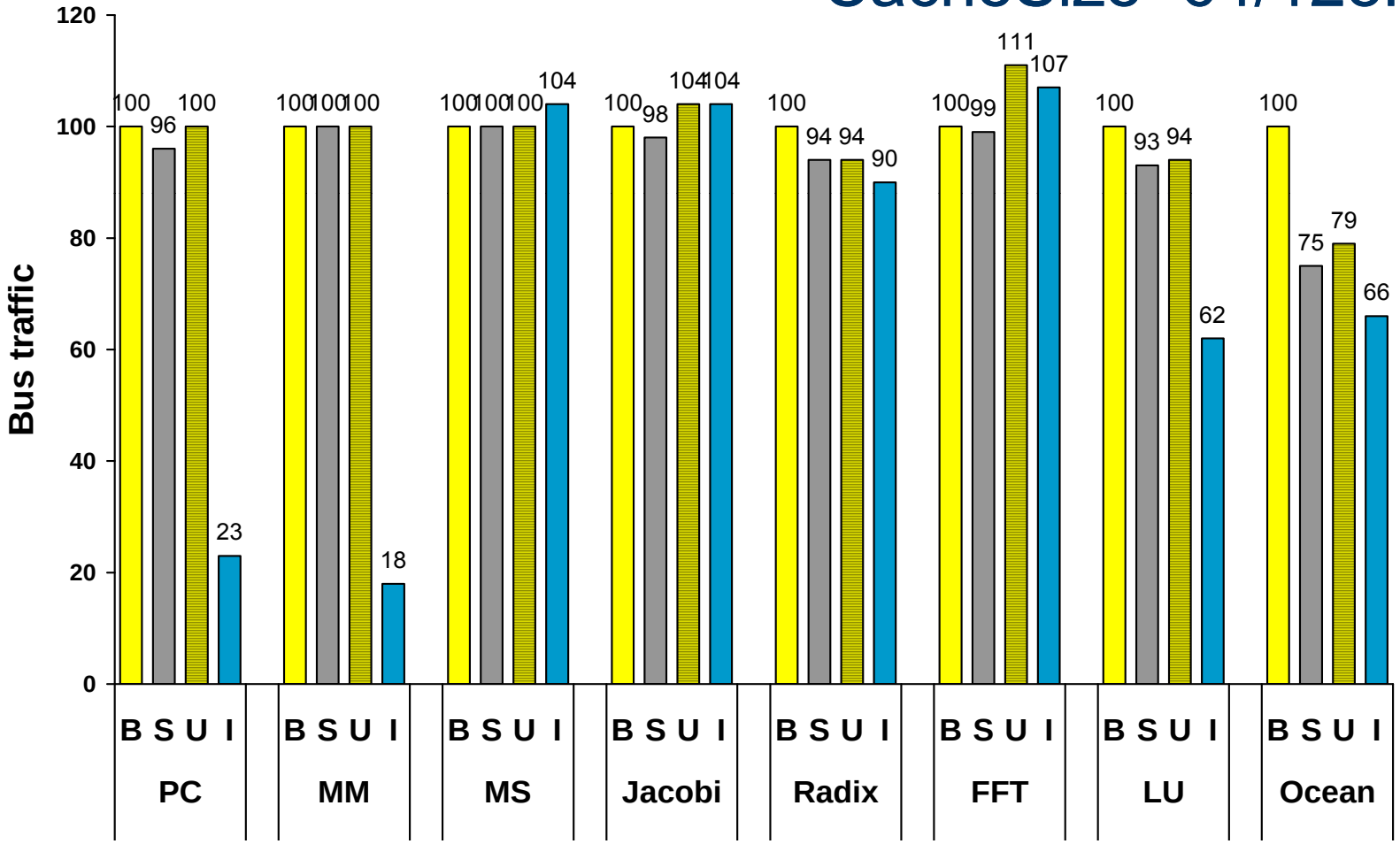
Results



Bus traffic

Results

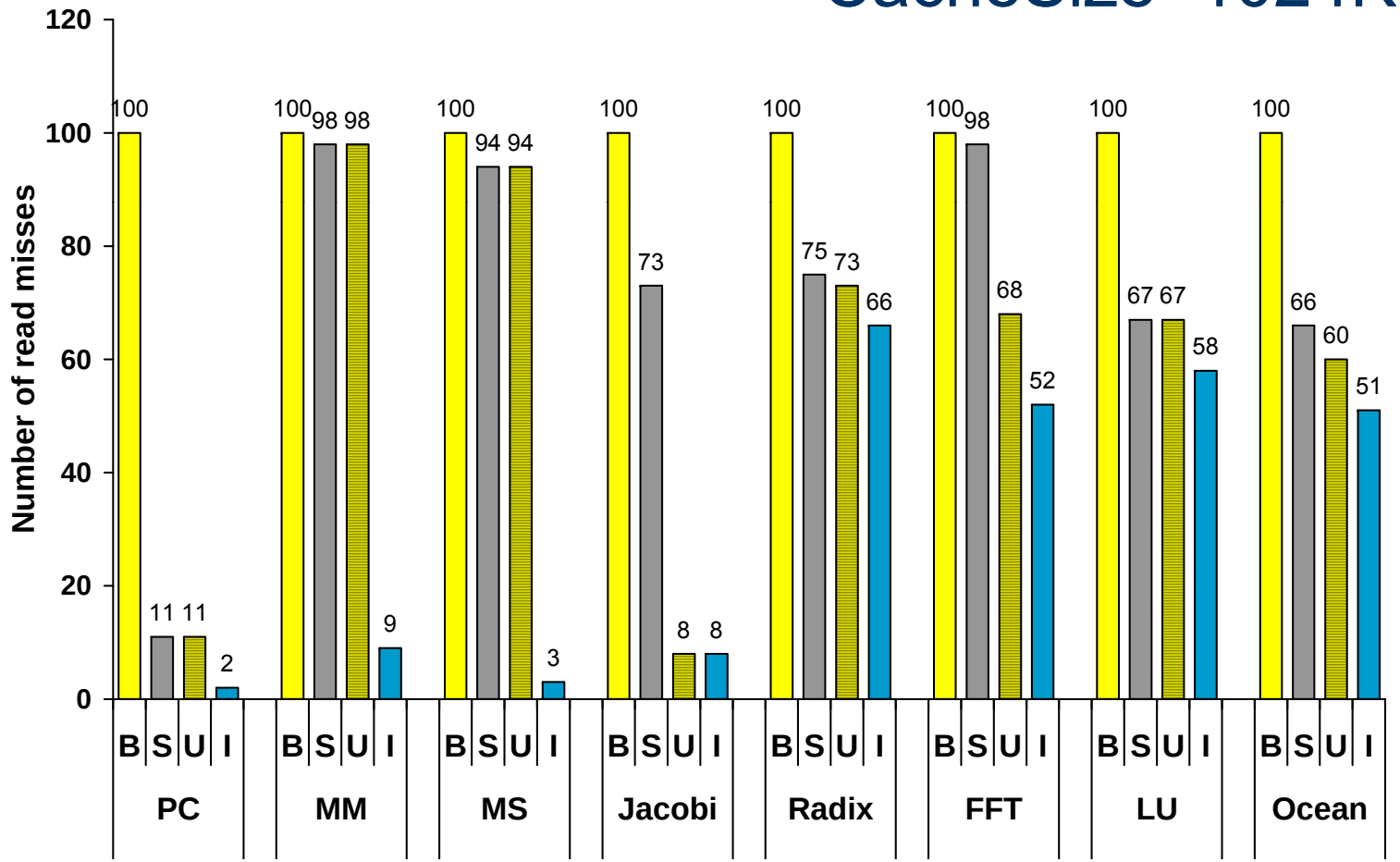
CacheSize=64/128KB



Number of read misses

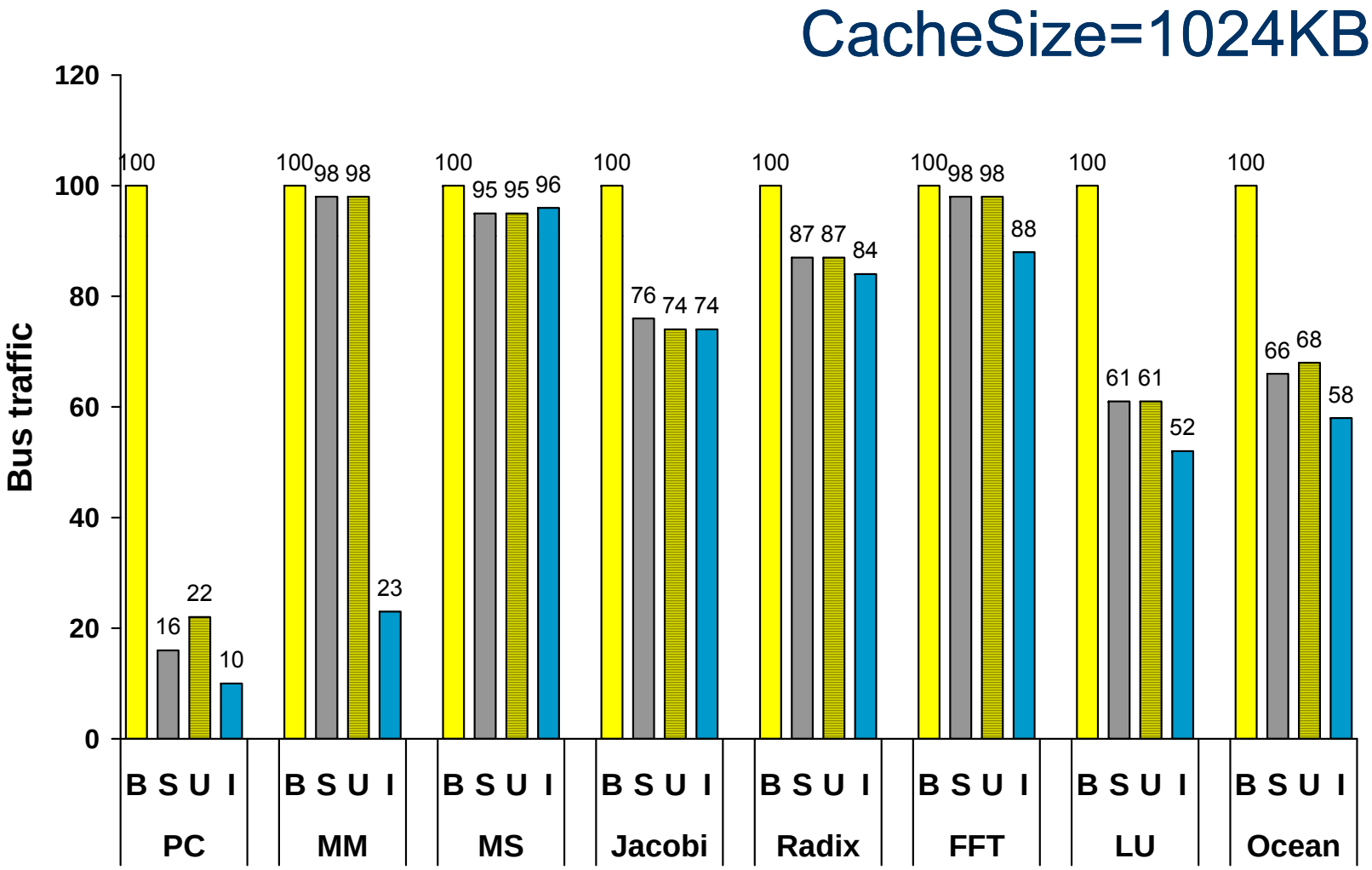
Results

CacheSize=1024KB



Bus traffic

Results



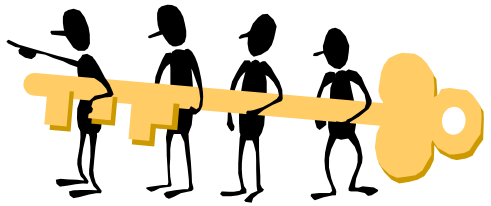
Conclusions

- Cache injection outperforms read snarfing and software-controlled updating
- It reduces the number of read misses by 6 to 90% (small caches), and by 27 to 98% (large caches)
- It reduces bus traffic for up to 82% (small caches), and up to 90% (large caches); it increases bus traffic for MS, Jacobi, and FFT in the system with small caches for up to 7%

Conclusions

- Effectiveness of cache injection relative to read snarfing and software-controlled updating is higher in the systems with relatively small caches
- Cache injection can be effective in reducing cold misses when there are multiple consumers of shared data (MM and LU)
- Software control of time window during which a block can be injected provides flexibility and adaptivity (MS and FFT)

Conclusions



- Cache injection further improves performance at minimal cost
- Cache injection encompasses the existing techniques read snarfing and software-controlled updating
- Possible future research directions
  - compiler algorithm to support cache injection
  - combining cache prefetching and cache injection
  - implementation of injection mechanism in scalable shared-memory cache-coherent multiprocessors