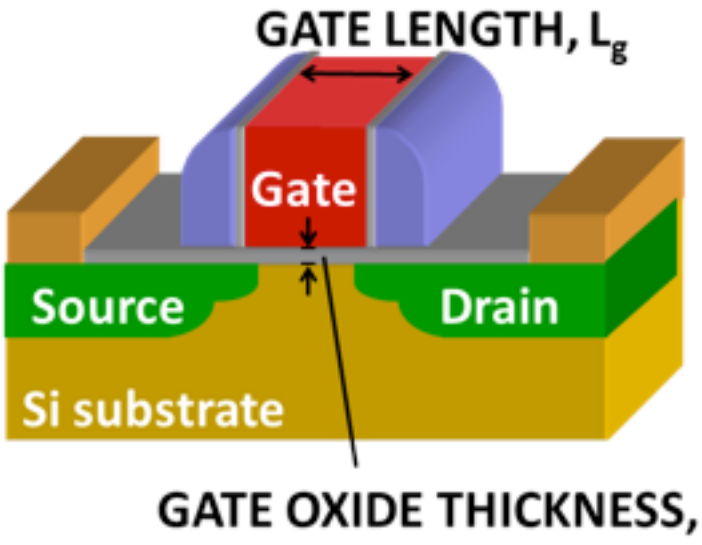
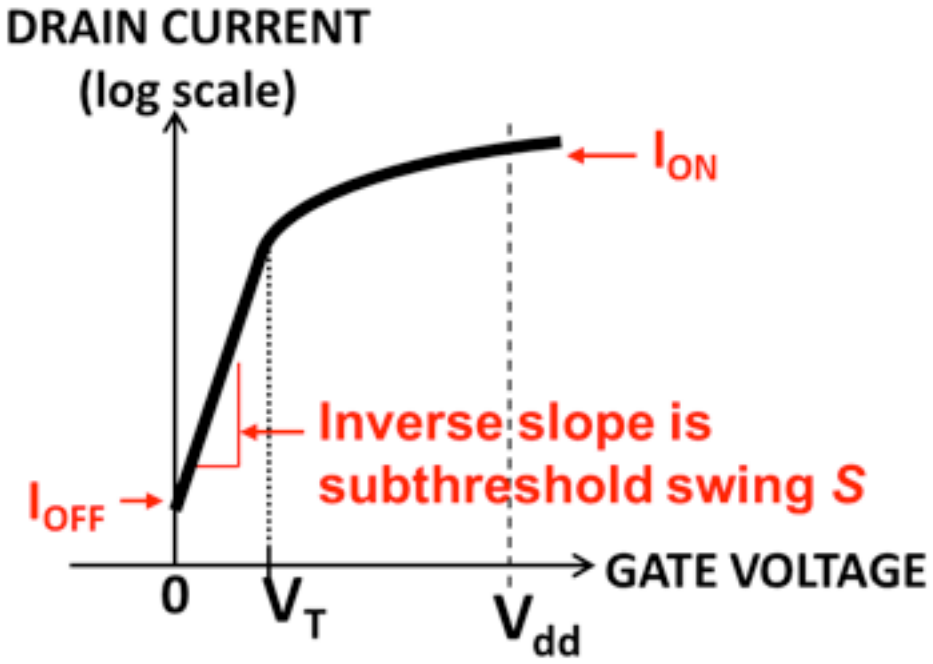
eetimes

1) IBM Fabrication
2) Reneseas: big/little
3) Reducing AMD power with gating
4) Flash on DDR4

Metal Oxide Semiconductor Field-Effect Transistor:





DARPA UPSIDE

[August 17, 2012]

Digital Processors Limited by Power; What's The Upside?

(Targeted News Service Via Acquire Media NewsEdge) WASHINGTON, Aug. 14 -- The U.S. Department of Defense's Defense Advanced Research Projects Agency issued the following news release: Today's Defense missions rely on a massive amount of sensor data collected by intelligence, surveillance and reconnaissance (ISR) platforms. Not only has the volume of sensor data increased exponentially, there has also been a dramatic increase in the complexity of analysis required for applications such as target identification and tracking. The digital processors used for ISR data analysis are limited by power requirements, potentially limiting the speed and type of data analysis that can be done. A new, ultra-low power processing method may enable faster, mission critical analysis of ISR data.

Chapter 5: Multiprocessors

High-Performance Computer Architecture

Prof. Ben Lee

Oregon State University

School of Electrical Engineering and Computer Science

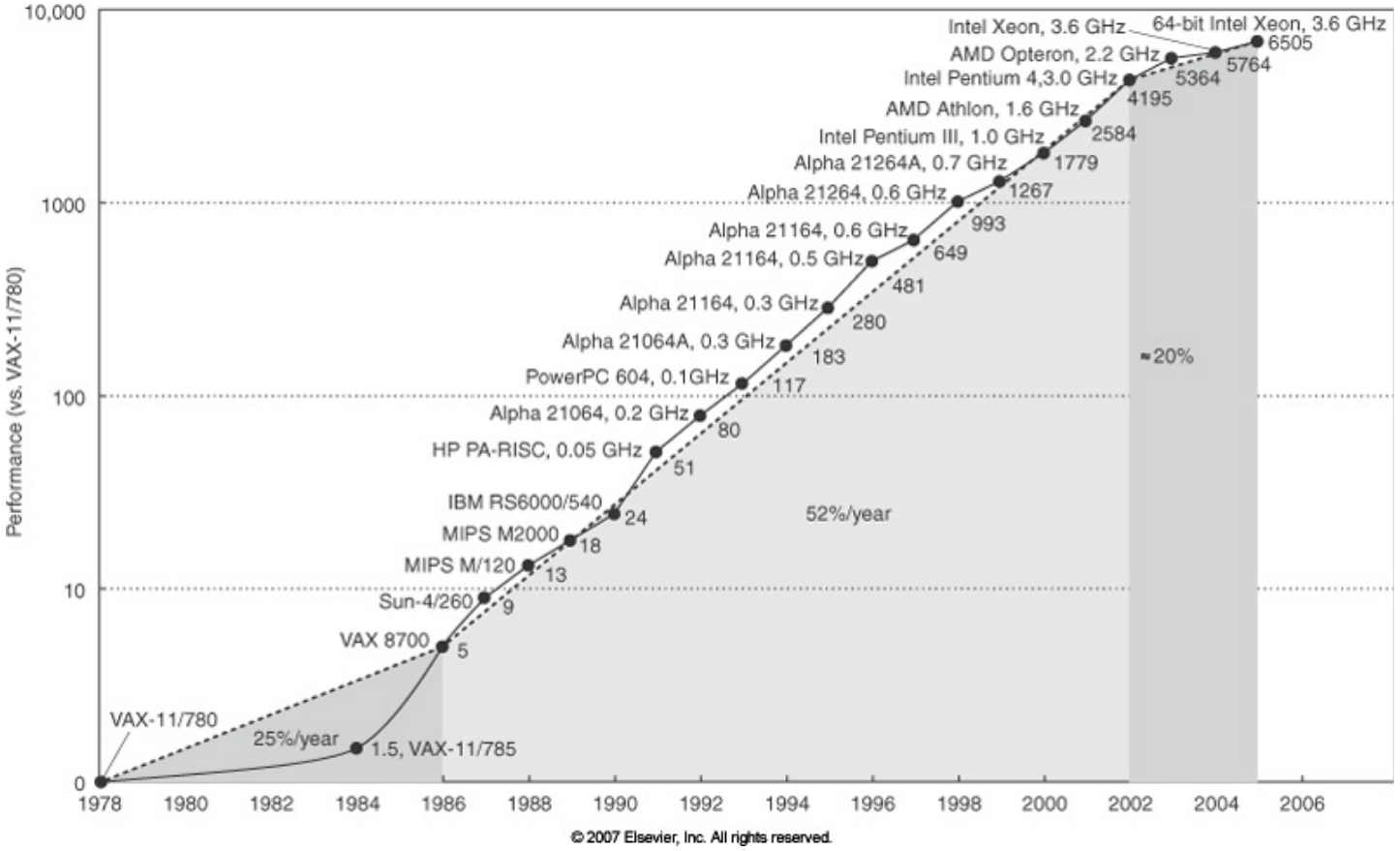
Chapter Outline

- Introduction
- Parallel Computers
- Shared-Memory Programming
- Synchronization
- Cache Coherence

Introduction

- Growth in data-intensive applications.
  - Data bases, file servers, …
- Growing interest in servers, server performance.
- Increasing desktop performance less important.
  - Outside of graphics
- Improved understanding in how to use multiprocessors effectively.
  - Especially servers where significant natural TLP
- Advantage of leveraging design investment by replication => CMPs or Multicores.
  - Rather than unique design

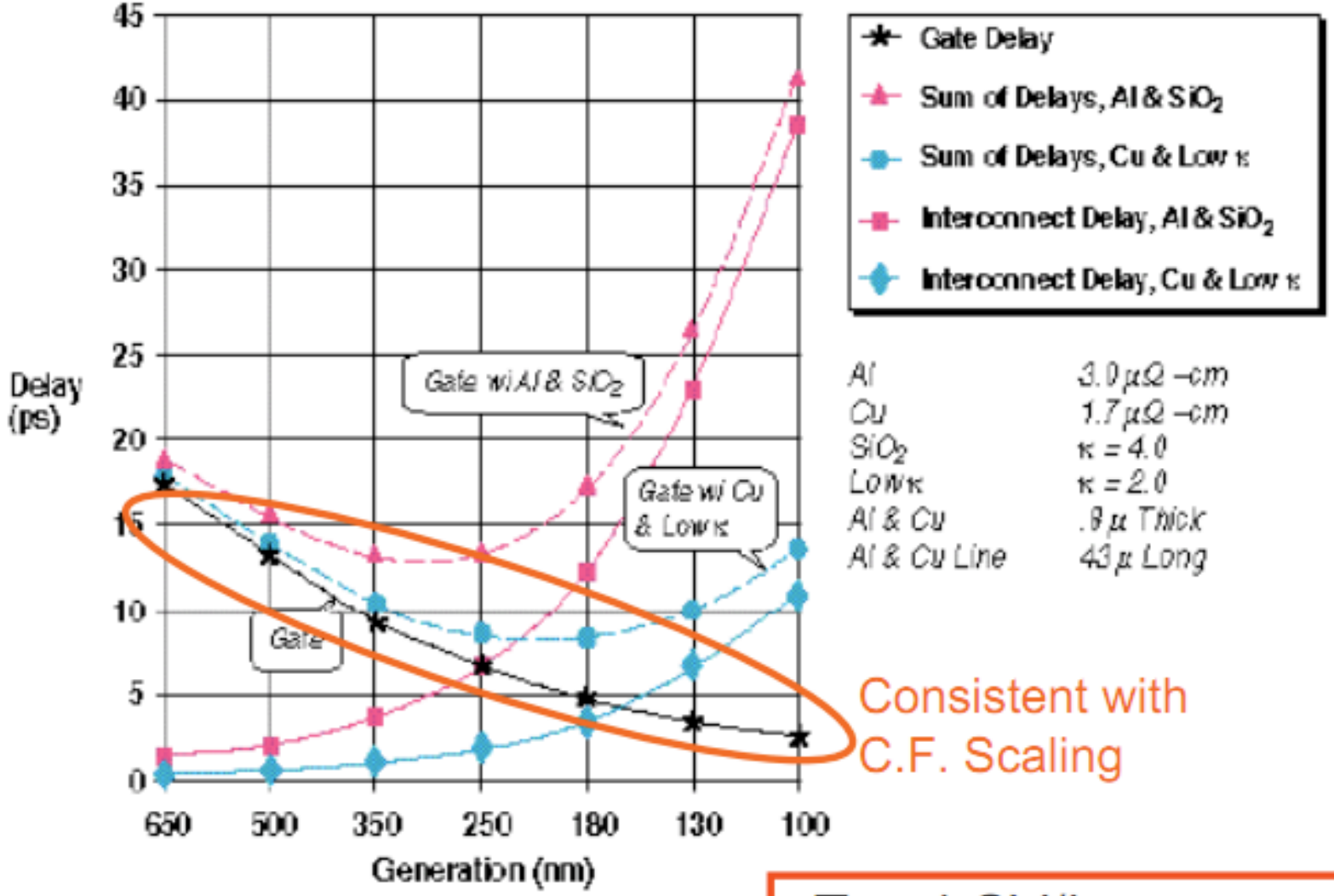
Another Reason



Limit to Processor Performance

- Complexity of exploiting ILP.
  - Difficult to support large instruction window and number of in-flight instructions.
- On-chip wires are becoming slower than logic gates.
  - Only a fraction of the die will be reachable within a single clock cycle.
- Cooling and packaging will be a real challenge due heat release.
- Memory and processor performance gap will continue to be a challenge.

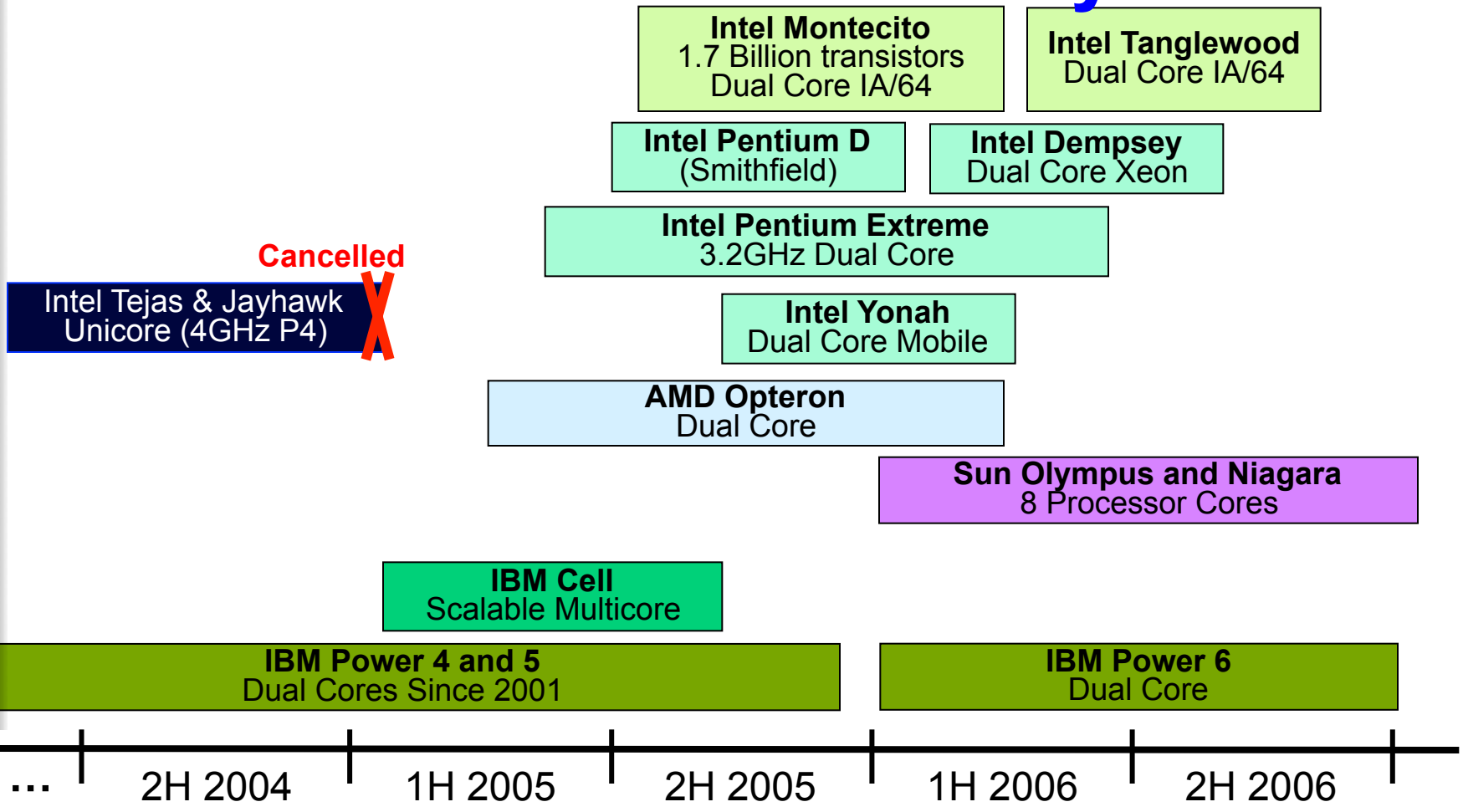
Gate Delay Trends



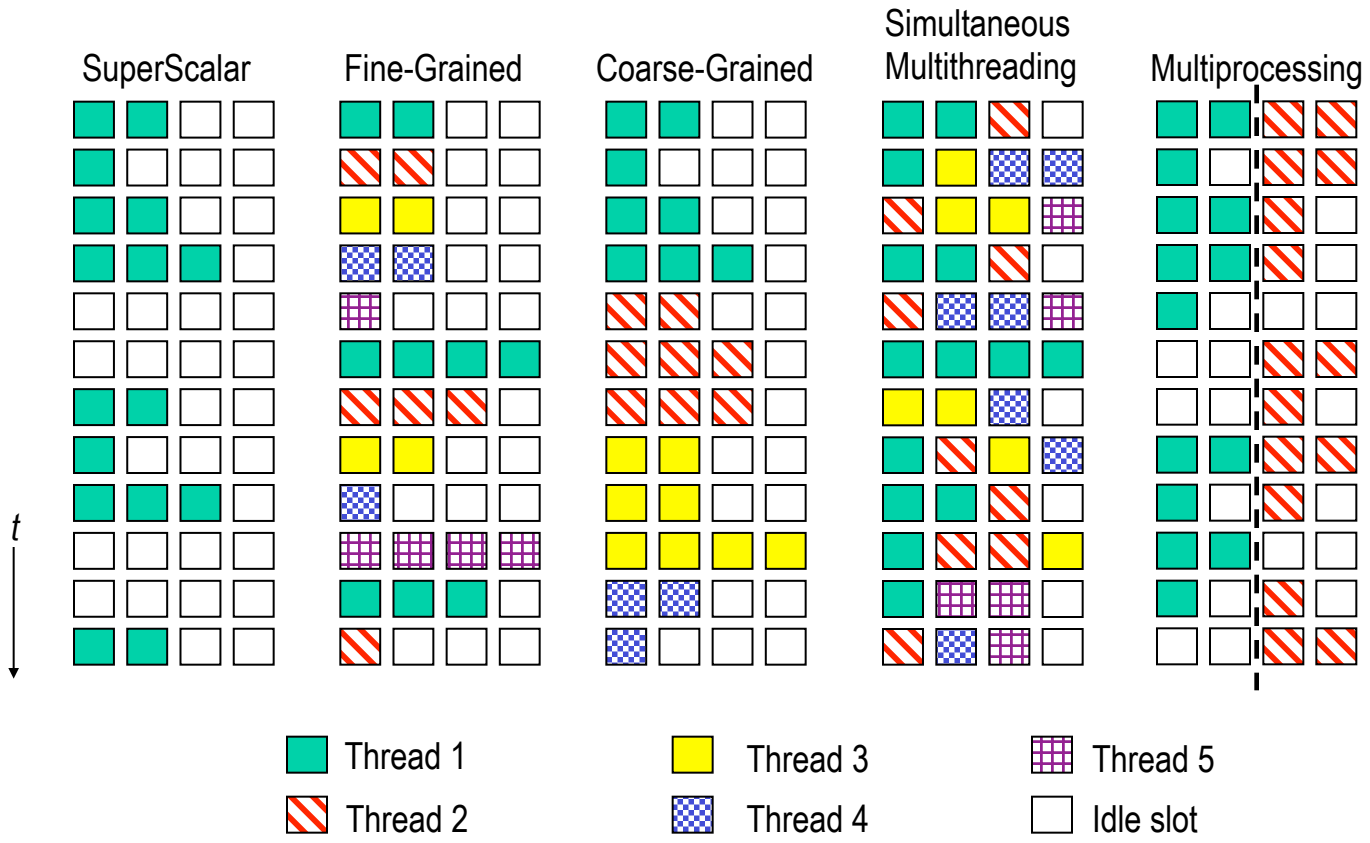
Each technology generation, gate delay reduced about 30% (src: ITRS '01)

$$T_d = kCV/I = kCV/(V_{dd}-V_t)^{\alpha}$$

Multicores are ~~Coming~~ Already Here!

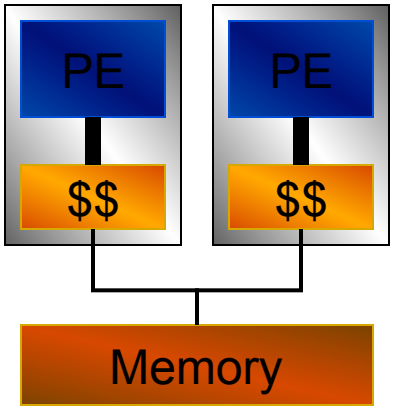


SS, MT, SMT, & MP

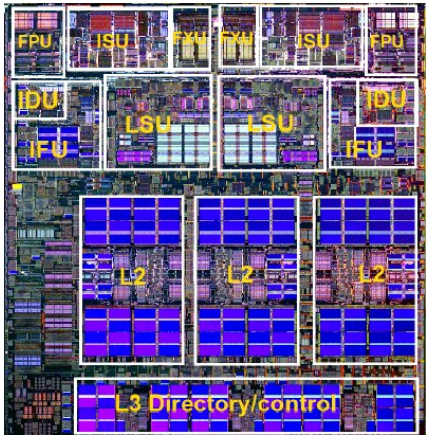

SuperScalar
Fine-Grained
Coarse-Grained
Simultaneous Multithreading
Multiprocessing
t
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Idle slot

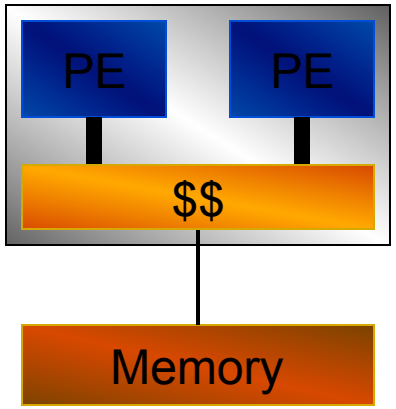
What is a Multicore?

Traditional Multiprocessor

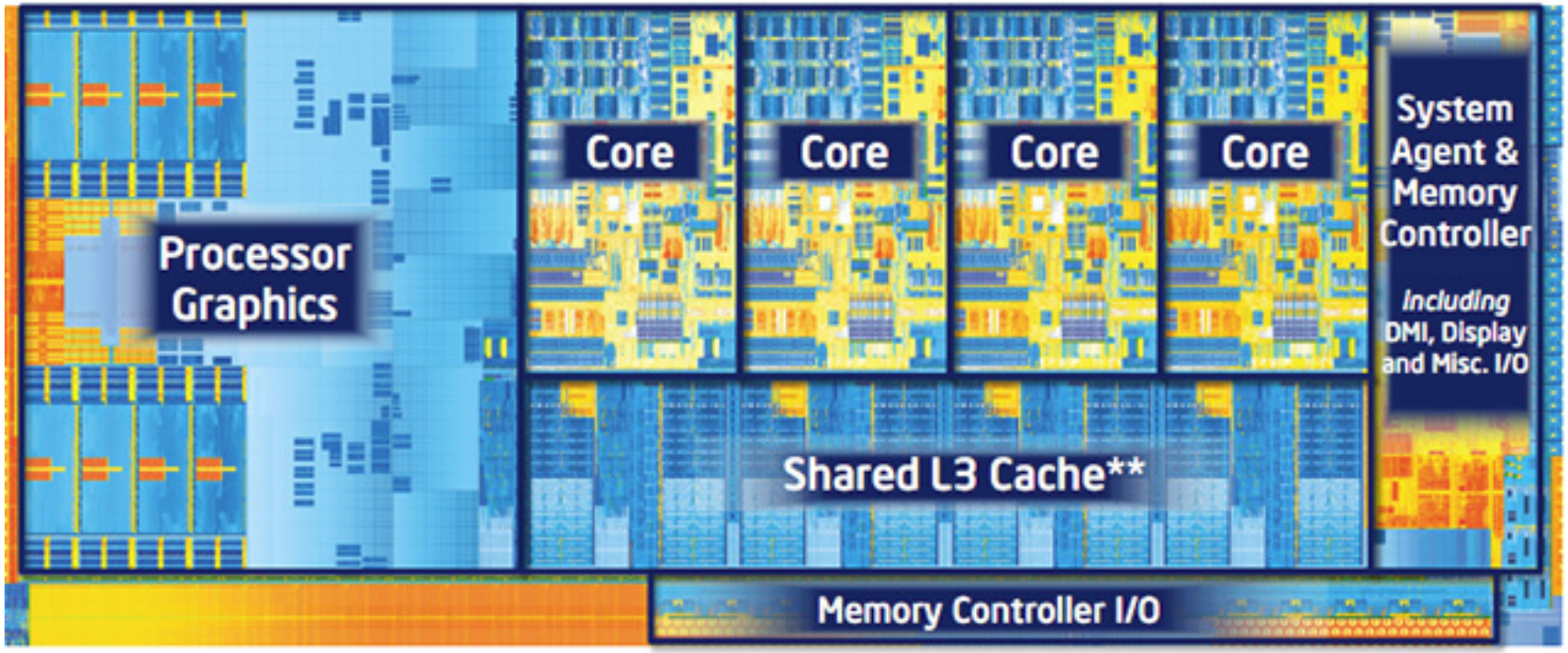


Basic Multicore IBM Power5



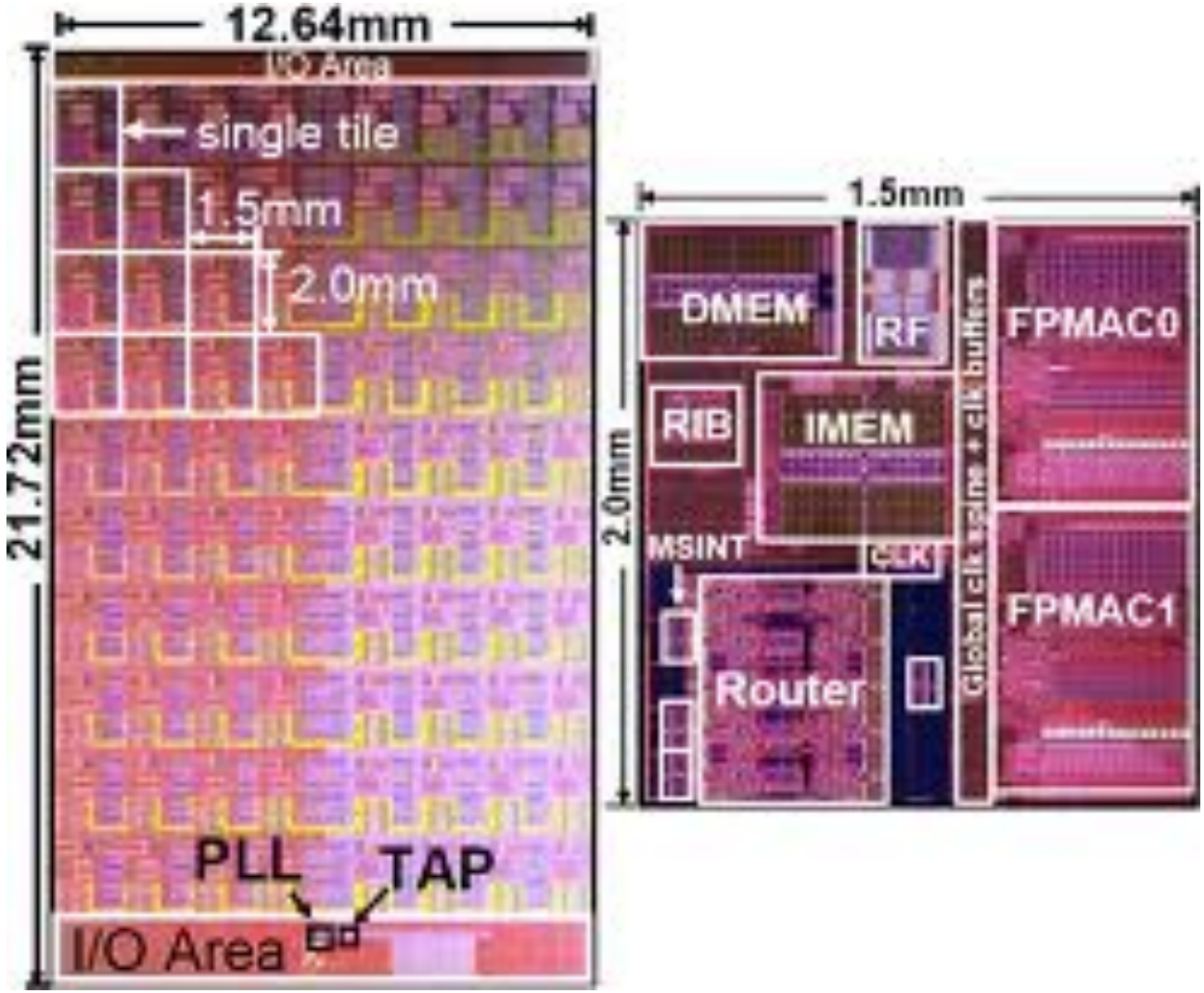


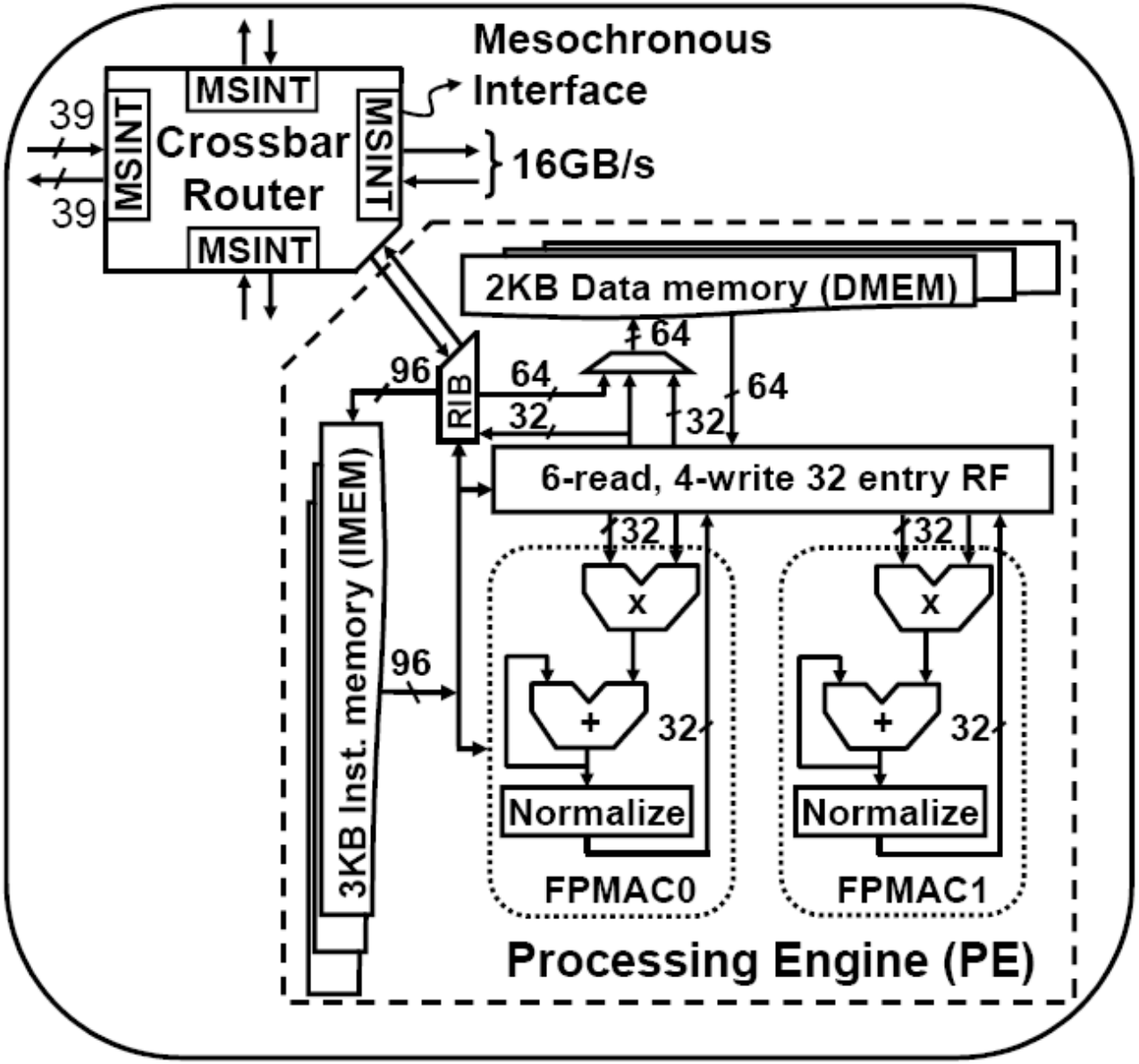
3rd Generation Intel® Core™ Processor: 22nm Process

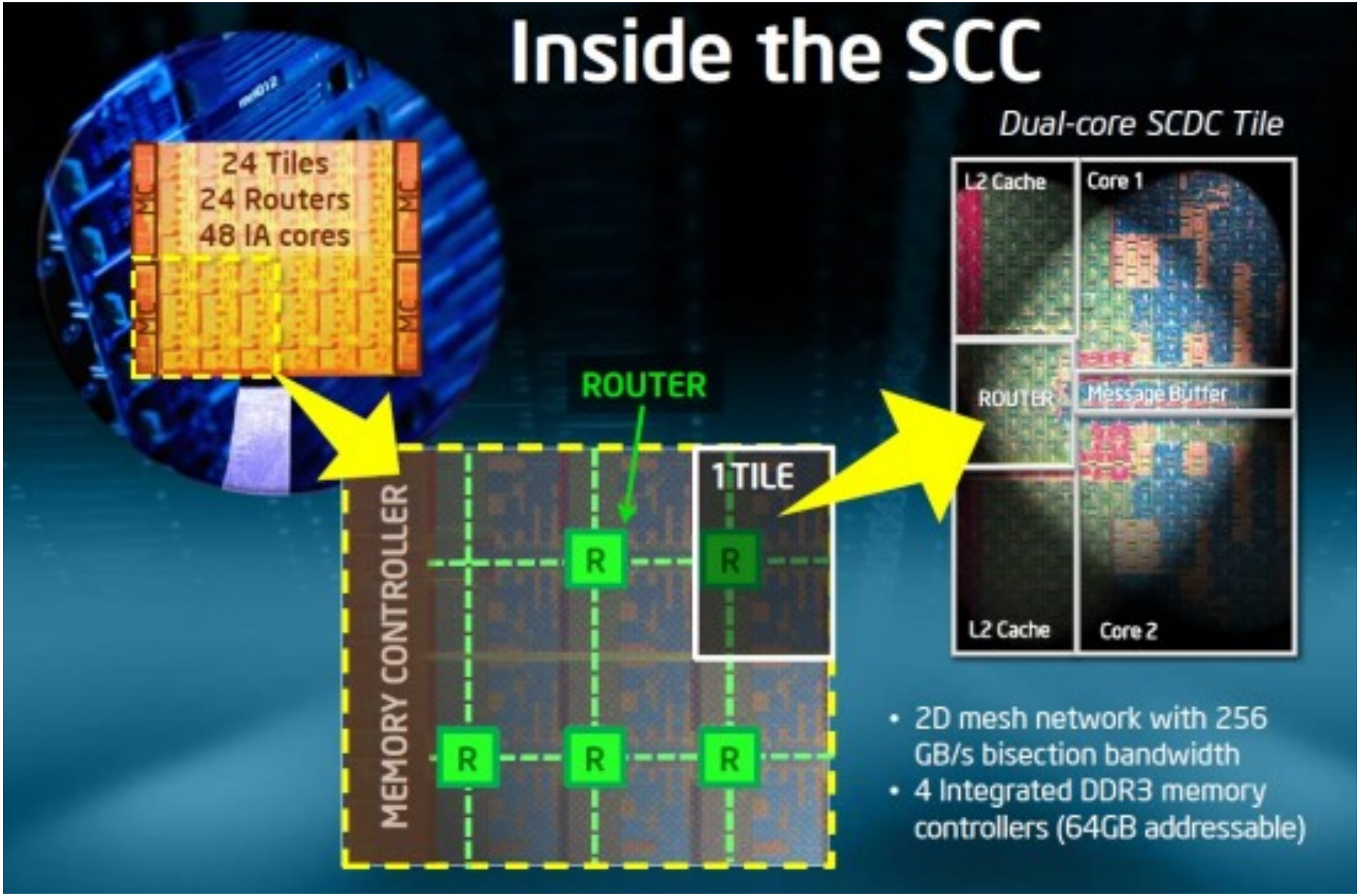


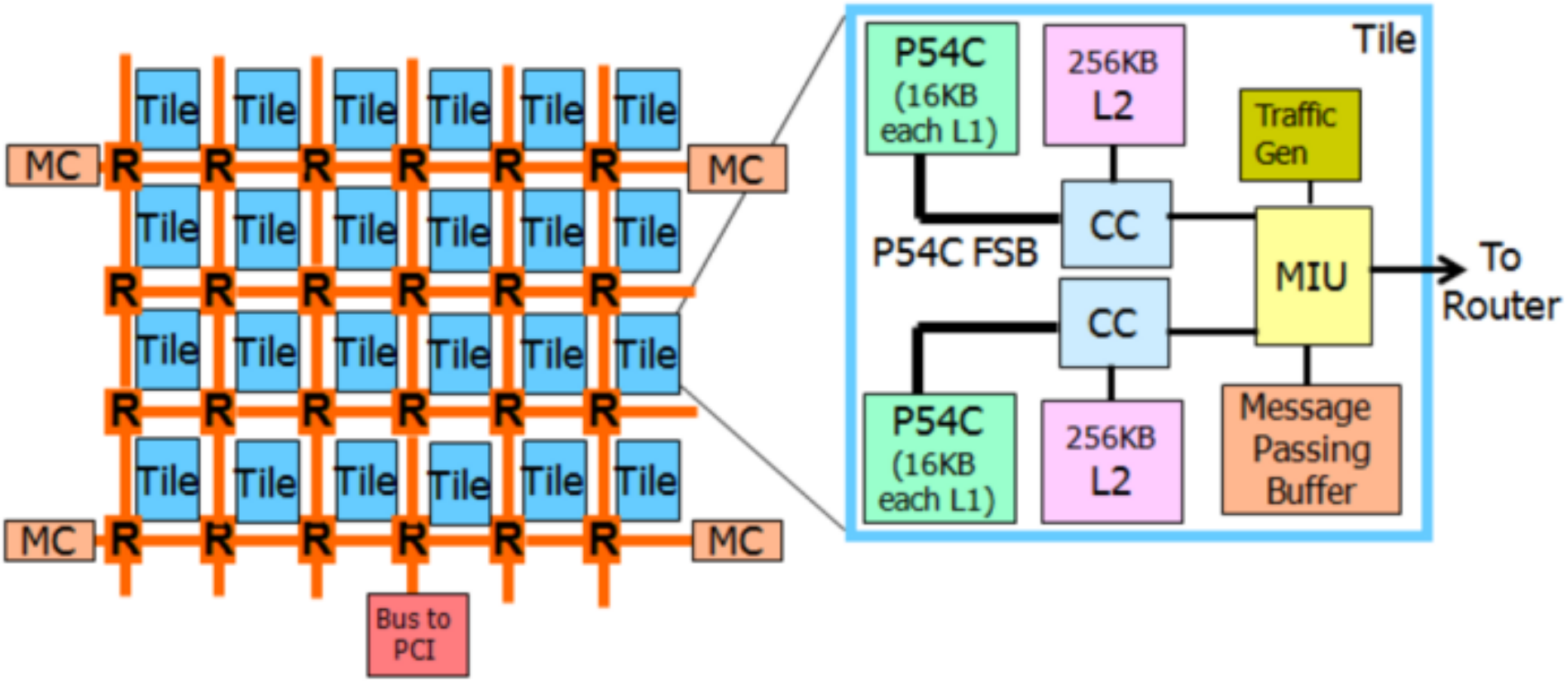
New architecture with shared cache delivering more performance and energy efficiency

Quad Core die with Intel® HD Graphics 4000 shown above
Transistor count: 1.4Billion Die size: 160mm²
** Cache is shared across all 4 cores and processor graphics


12.64mm
I/O Area
single tile
1.5mm
2.0mm
21.72mm
PLL
TAP
I/O Area
1.5mm
2.0mm
DMEM
RF
FPMAC0
RIB
IMEM
MSINT
CLK
FPMAC1
Router
Global clk spine + clk buffers


Mesochronous
Interface
MSINT
MSINT
MSINT
MSINT
39
39
Crossbar
Router
16GB/s
2KB Data memory (DMEM)
64
96
64
64
RIB
32
32
6-read, 4-write 32 entry RF
32
32
3KB Inst. memory (IMEM)
96
X
X
+
+
32
32
Normalize
Normalize
FPMAC0
FPMAC1
Processing Engine (PE)


Inside the SCC
24 Tiles
24 Routers
48 IA cores
MC
MC
MC
MC
ROUTER
1 TILE
MEMORY CONTROLLER
R
R
R
R
R
Dual-core SCDC Tile
L2 Cache
Core 1
ROUTER
Message Buffer
L2 Cache
Core 2
• 2D mesh network with 256 GB/s bisection bandwidth
• 4 Integrated DDR3 memory controllers (64GB addressable)


Tile
MC
R
Bus to PCI
P54C (16KB each L1)
256KB L2
Traffic Gen
CC
P54C FSB
MIU
To Router
Message Passing Buffer

Knights Corner

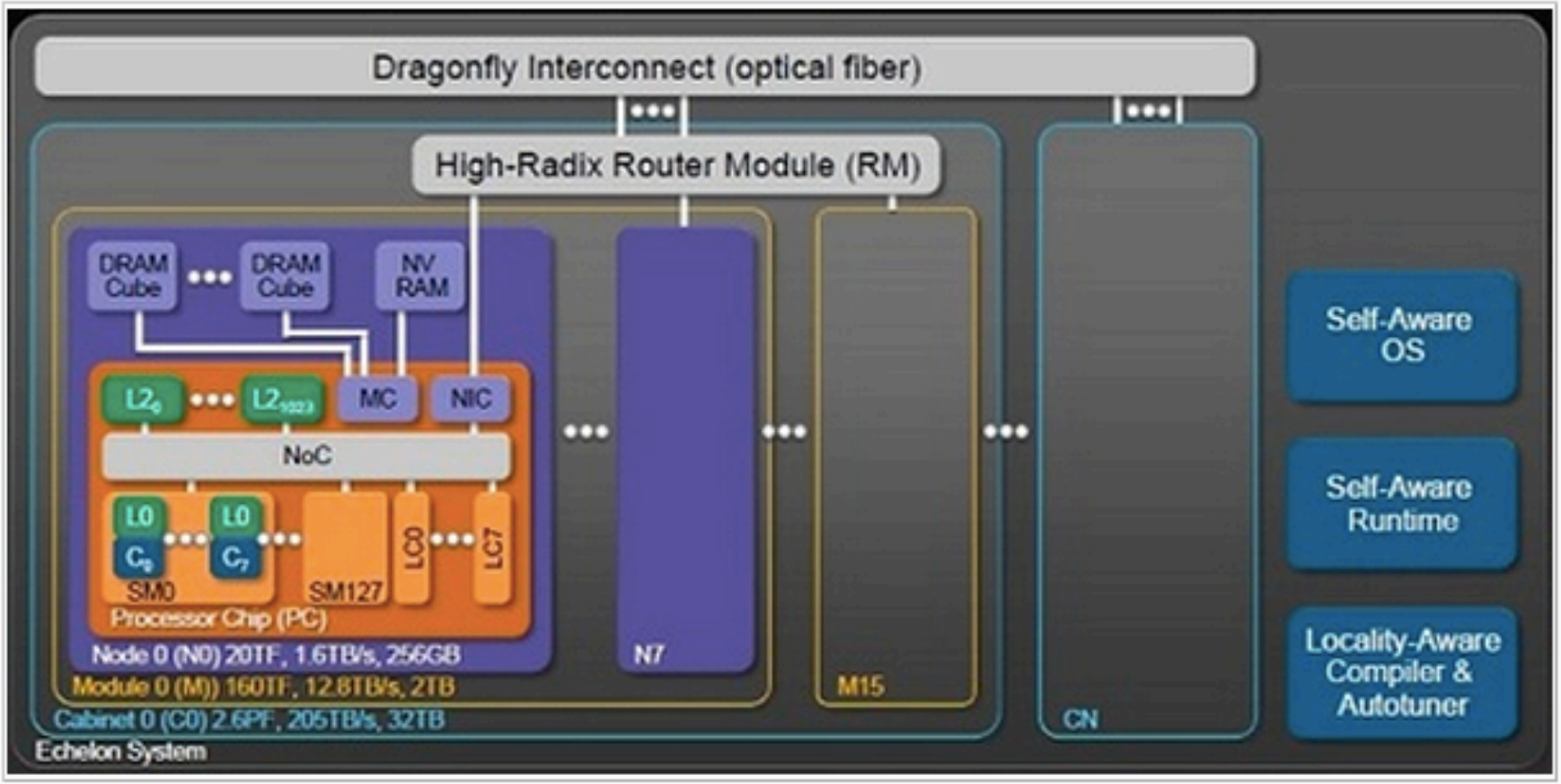


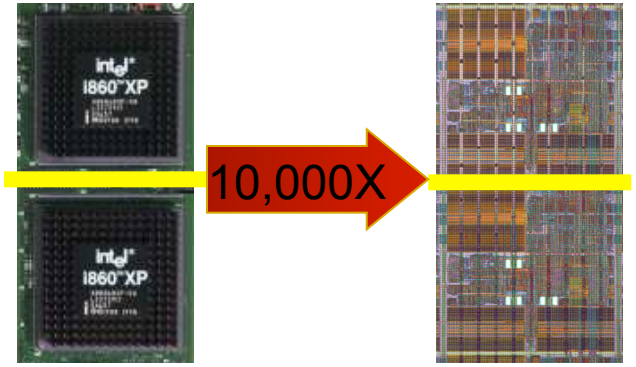
Figure 3: nVIDIA's Echelon Architecture

Why Move to Multicores

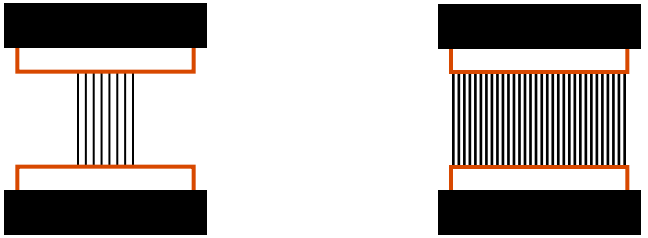
- Many issues with scaling a unicore
  - Power
  - Efficiency
  - Complexity
  - Wire Delay
  - Diminishing returns from optimizing a single instruction stream

Impact of Multicores

- How much data can be communicated between two cores?
- What changed?
  - Number of Wires
    - I/O is the true bottleneck
    - On-chip wire density is very high
  - Clock rate
    - I/O is slower than on-chip
  - Multiplexing
    - No sharing of pins
- Impact on programming model?
  - Massive data exchange is possible
  - Data movement is not the bottleneck
    - Locality is not that important

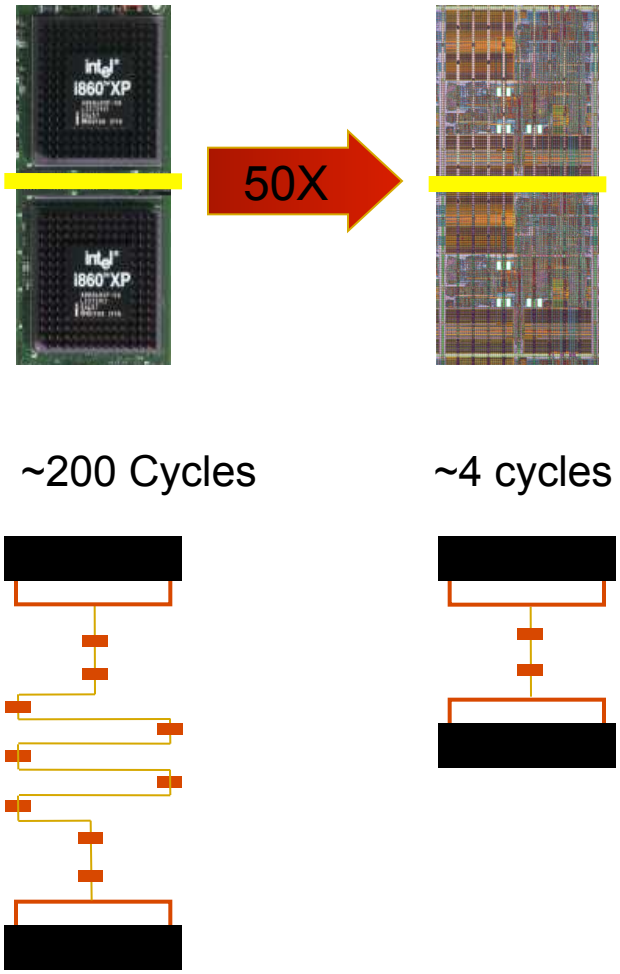


32 Giga bits/sec ~300 Tera bits/sec



Impact of Multicores

- How long does it take for a round trip communication?
- What changed?
  - Length of wire
    - Very short wires are faster
  - Pipeline stages
    - No multiplexing
    - On-chip is much closer
- Impact on programming model?
  - Ultra-fast synchronization
  - Can run real-time apps on multiple cores

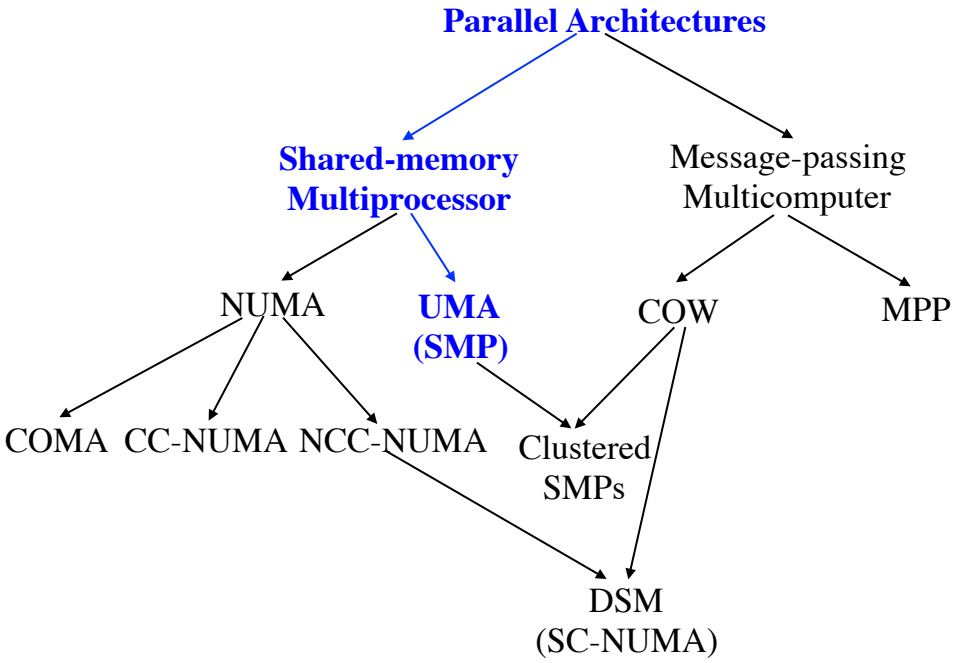


Chapter Outline

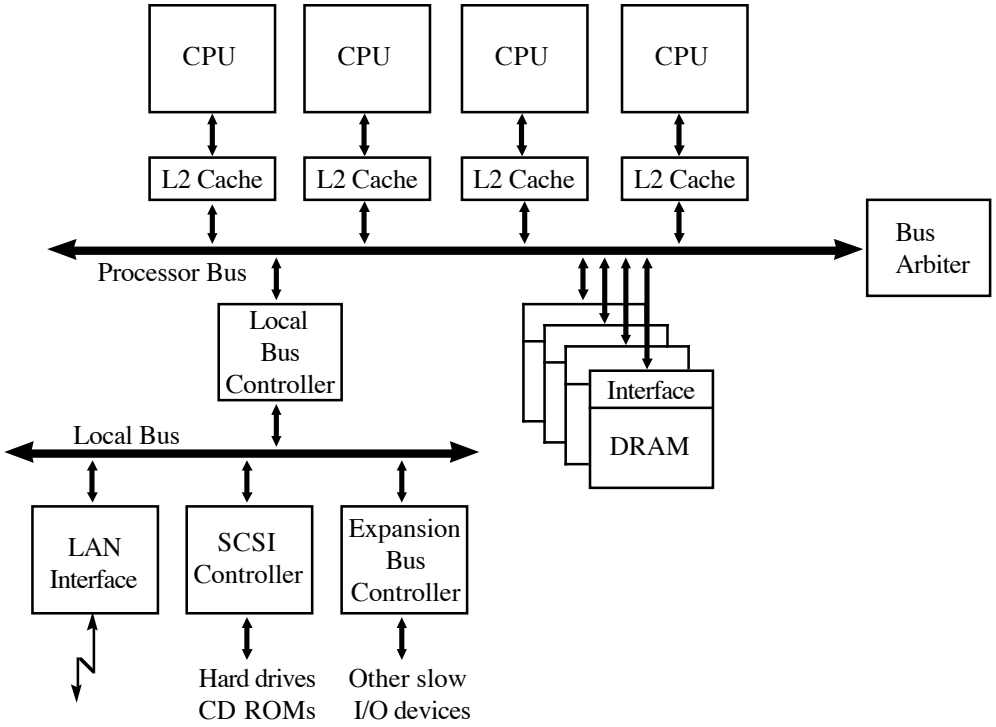
- Introduction
- Parallel Computers
- Shared-Memory Programming
- Synchronization
- Cache Coherence

Parallel Computers

- "A parallel computer is a collection of processing elements that *cooperate* and *communicate* to solve large problems fast."
- Taxonomy of parallel computers:



Shared-Memory Multiprocessors



Issues in Multiprocessors

- Programming
  - Need to explicitly define parallelism
- Hardware
  - Interconnection
    - Bus
    - Network
  - Synchronization
    - Test-and-set
    - Barrier synchronization
    - Fetch and add
  - Cache Coherence
    - Snoopy protocols
    - Directory-based

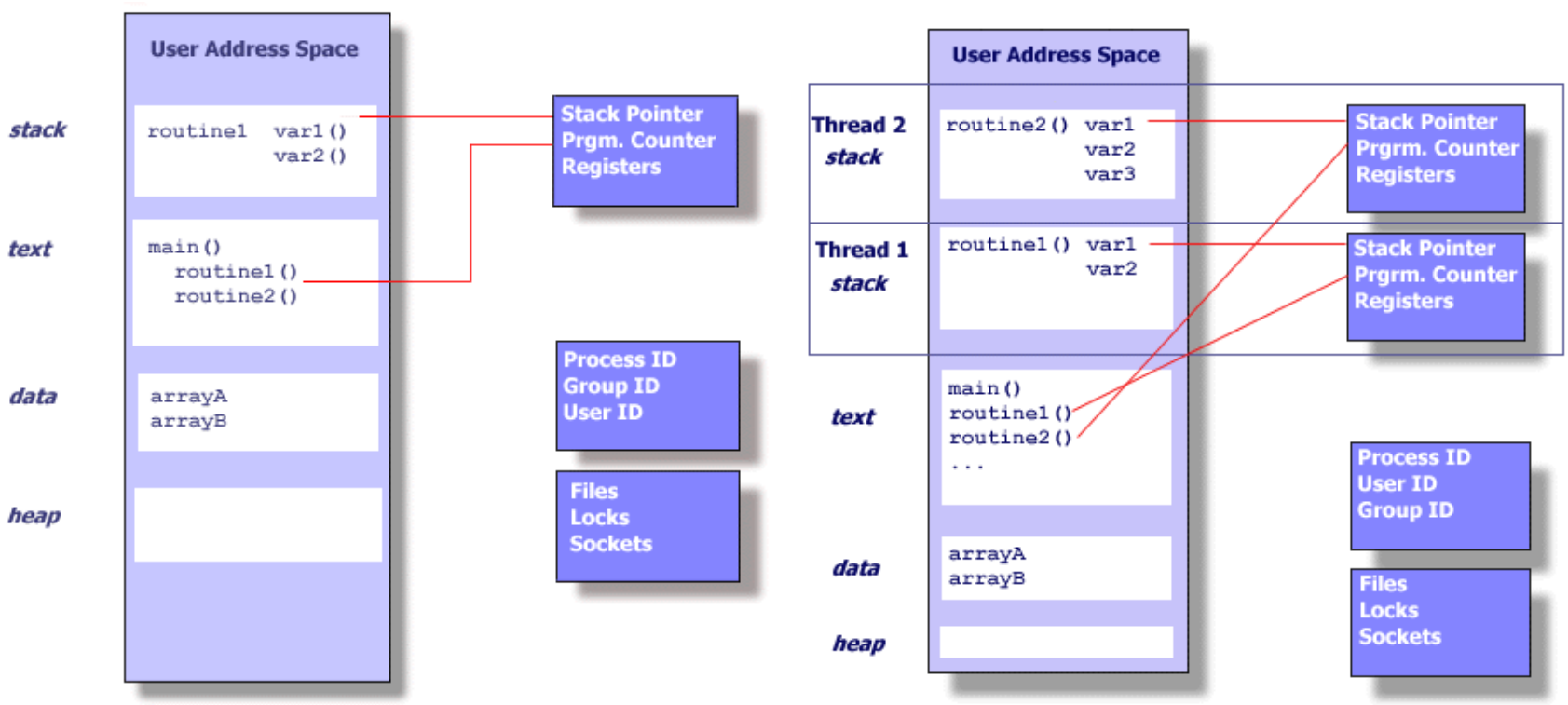
Chapter Outline

- Introduction
- Parallel Computers
- Shared-Memory Programming
- Synchronization
- Cache Coherence

Shared-Memory Programming

- Many vendors have implemented their own proprietary versions of threads.
- A standardized C language threads programming interface, POSIX Threads or *Pthreads*
- Threaded applications offer potential performance gains and practical advantages:
  - Overlapping CPU work with I/O
  - Priority/real-time scheduling
  - Asynchronous event handling
  - Parallelization on SMPs
- Pthreads provide Over 60 routines for
  - Thread management - thread create, join, schedule, etc.
  - Mutexes - mutual exclusion
  - Conditional variables - provides communication between threads

Threads

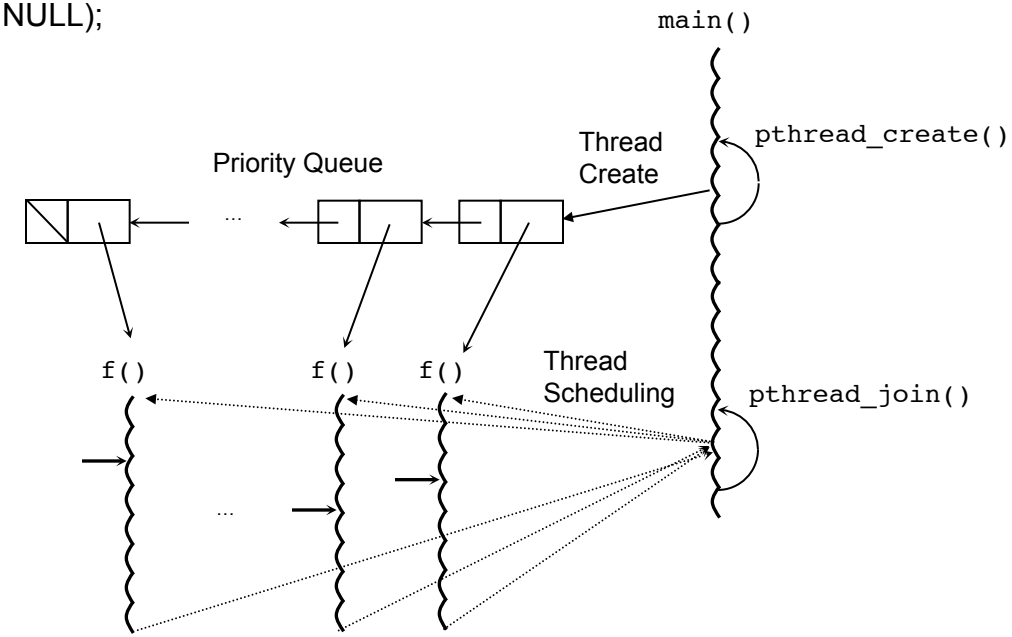


Unix Process

Threads within a process

Thread Creation

```
main()
{
      for (i=0;i<=n;i++)
        pthread_create(&thread_id[i],NULL,f,(void *)&args[i]);
      ...
      for (i=0;i<=n;i++)
        pthread_join(thread_id[i], NULL);
}


void *f(void *param)
{
      ...
      do something
      ...
```



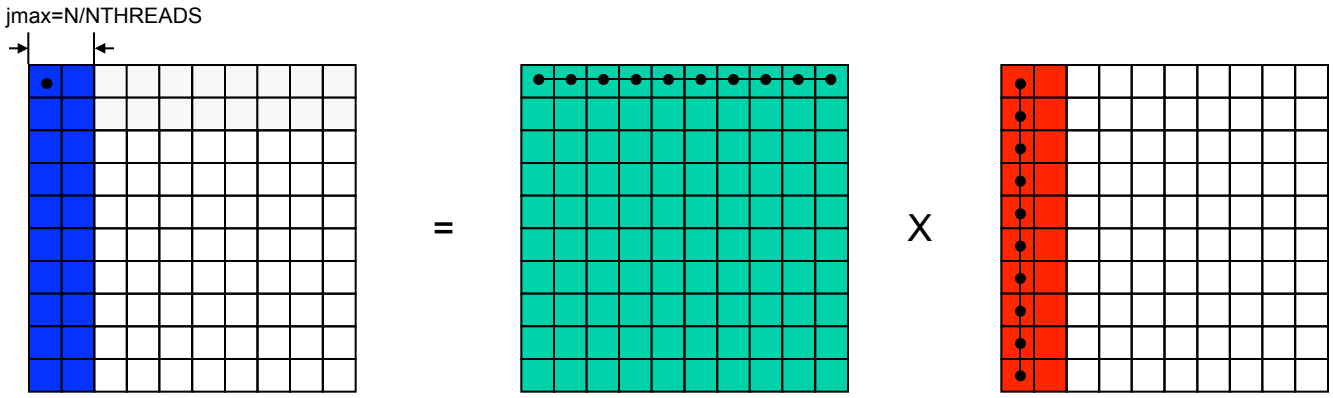
Matrix Multiply

```
/******************************************************************************
Simple Multi-threaded matrix multiplication
compile with cc -mt -xO3 -D_SOLARIS_2 thmm.c -lpthread
or
cc -mt -xO3 -xtarget=ultra2 -xcache=16/32/1:4096/64/1 -D_SOLARIS_2
        -xunroll=4 thmm.c -lpthread
******************************************************************************/

#include <stdio.h>
#include <pthread.h>
#include <sys/time.h>
#ifdef _SOLARIS_2
#include <thread.h>
#endif

#define N 500
#define NTHREADS 100
int jmax = N/NTHREADS;

/* function prototypes */
void* matMult( void* );

/* global matrix data */
double a[N][N], b[N][N], c[N][N];
int count=0;
```

```
void main( void )
{    pthread_t thr[NTHREADS];
     int i, j;

     #ifdef _SOLARIS_2
     thr_setconcurrency(NTHREADS);
     #endif

     for( i = 0; i < NTHREADS; ++i ) {
          pthread_create(&thr[i], NULL, matmult, (void*)i);
     }
     for( i = 0; i < NTHREADS; ++i ) {
          pthread_join(thr[i], NULL);
     }
}

void* matmult(void* thread_id)
{    int i, j, k;
     int offset, column;
     offset = jmax*(int)thread_id;

     for(j = 0; j < jmax; j++) {
          column = j+offset;
          for(i = 0; i < N; i++) {
               for(k = 0; k < N; k++) {
                         c[i][column] = c[i][column] + a[i][k]*b[k][column];
               }
          }
      }
     return NULL;
}
```

Matrix Multiply

```
void* matmult(void* thread_id)
{       int i, j, k;
        int offset, column;
        offset = jmax*(int)thread_id;
        for(j = 0; j < jmax; j++) {
                column = j+offset;
                for(i = 0; i < N; i++) {
                        for(k = 0; k < N; k++) {
                                c[i][column] = c[i][column] + a[i][k]*b[k][column];
                        }
                }
         }
        return NULL;
}
```



Chapter Outline

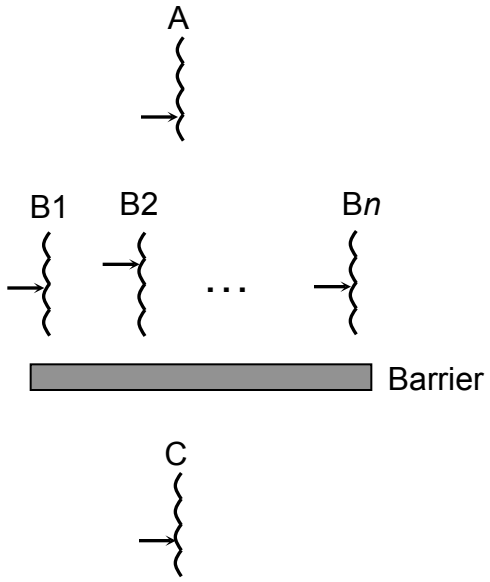
- Introduction
- Parallel Computers
- Shared-Memory Programming
- Synchronization
- Cache Coherence

Synchronization

- Synchronization enforces correct sequencing of processors and ensures mutually exclusive access to shared writable data.
  - Sequence control
  - Access control

Sequence Control

- Ensures correct timing among cooperating threads/ processes.
  - e.g., barrier synchronization



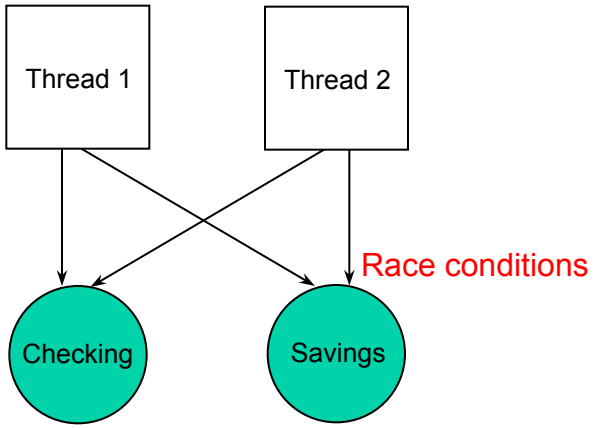
Access Control

```
/* Savings to checking transfer */

struct account {
    int checking;
    int savings;
};

void
savings_to_checking(struct account *ap,
    int amount)
{
    ap->savings -=amount;
    ap->checking += amount;
}

int
total_balance(struct account *ap)
{
    int balance;
    balance = ap->checking + ap->saving;
    return balance;
}
```

Critical Section

Critical Section

Thread 1	Thread 2
	`ap->savings -=amount;`
`balance = ap->checking +`	
`ap->saving;`	
	`ap->checking += amount`
`return balance;` Wrong balance!	



Requires mutual exclusion between `saving_to_checking` and `total_balance`

Pthreads Synchronization

- Need a synchronization variable allocated in memory.
  - Lock/unlock
- All threads need to check the synchronization variable.
  - If in use, wait until it becomes available.
- Pthread uses mutexes (mutual exclusions):
  - Creating and destroying a mutex
    - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
    - `int pthread_mutex_init`
    - `int pthread_mutex_destroy`
  - Locking and unlocking a mutex
    - `int pthread_mutex_lock`
    - `int pthread_mutex_unlock`

Previous Example

```
/* Savings to checking transfer */

struct account {
     int checking;
     int savings;
};
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void
savings_to_checking(struct account *ap, int
     amount)
{
     pthread_mutex_lock(&lock);
     ap->savings -=amount;
     ap->checking += amount;
     pthread_mutex_unlock(&lock);
}

int
total_balance(struct account *ap)
{
     int balance;
     pthread_mutex_lock(&lock);
     balance = ap->checking + ap->saving;
     pthread_mutex_unlock(&lock);
     return balance;
}
```

Critical Section

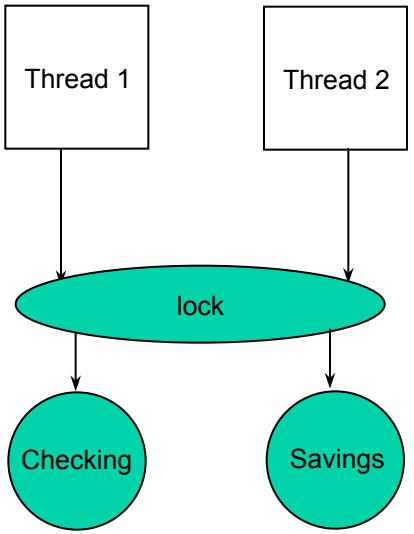
Critical Section

Thread 1

```
balance = ap->checking +
          ap->saving;

return balance;
```

Thread 2

```
ap->savings -=amount;

ap->checking += amount;
```



Synchronization Primitives

- `pthread_mutex_lock` and `_unlock` are high-level function calls.
- At some point within these calls, `lock` has to be read-test/modify-write atomically.
- Older (CISC-based) multiprocessors provided synchronization primitives:
  - Test-and-Set
  - Fetch-and-Add
  - Compare-and-Swap
  - Exchange (x86)

Test-and-Set

- Test-and-Set serializes access to a CS by forcing only one of several threads trying to access the CS. A thread is allowed to enter the CS only if Test-and-Set observes that lock =0.

```
Test-and-Set(lock)
{    temp ← lock;
     lock ← 1;
     return temp}

/* Example code */
loop:    lock=Test-and-Set(lock)
         if lock==1 then goto loop
         {
             Critical Section
         }
         Reset(lock){lock ← 0}
```

Fetch-and-Add

- Fetch-and-Add is more power than Test-and-Set in the sense that it allows arbitrary number of threads to share a counter. Each thread after executing Fetch-and-Add will acquire a unique number, which can then be used in number of different ways.

```
Fetch-and-Add(x,a)
{    temp ← x;
     x← temp+a;
     return temp}

/* Example code */
count = 1;
while (count <= n){
     count=fetch&add(i,1);
     if(count <= n)
         {/*  execute iteration i of the loop  */}
}
```

Hardware Support for Sync

- Modern processors are RISC-based.
  - RISC ISAs do not allow read-test/modify-write operations to be performed in a single instruction.
  - Need to provide special instructions.
  - Exception: x86 (uses `xchg`)
- Solution: A pair of special load and store instructions.
  - MIPS: `ll` (load-linked or load-locked) and `sc` (store-conditional)
  - PowerPC: `lwarx` (load word and reserve indexed) and `stwcx.` (store word conditional indexed)

MIPS Support for Sync

- MIPS processor provides LL-SC pair:
  - `ll` (load-linked or load-locked) - Loads a value and stores the address of the value in a *link register*. If an interrupt occurs (e.g., context switch), or if the cache block matching the address in the link register is invalidated (e.g., by another sc), the link register is cleared.
  - `sc` (store-conditional) - Stores a value only if address matches the address in the link register, and is valid. Returns 1 if it succeeds and 0 otherwise.

MIPS Support for Sync

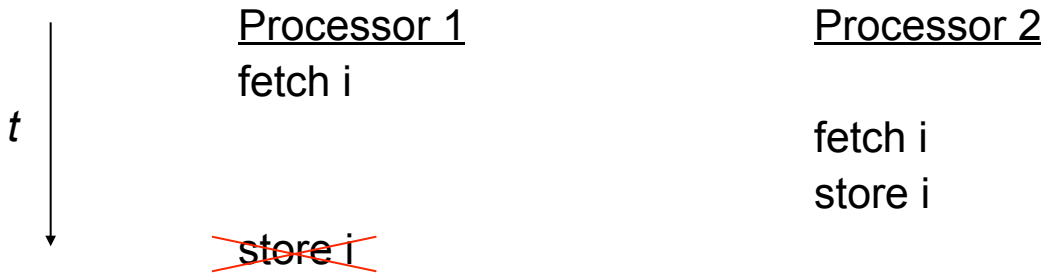
```
; Example of Test-and-Set implementation
lockit:  ll       R2, 0(R1)         ; load-linked
         bnez     R2, lockit        ; not available => spin
         daddui   R2, R0, #1        ; locked value
         sc       R2, 0(R1)         ; conditional store
         beqz     R2, lockit        ; branch if SC fails
```

```
; Example of Fetch-and-Add implementation
try:     ll       R2, 0(R1)         ; load-linked
         addi     R3, R2, #1        ; increment
         sc       R23, 0(R1)        ; store conditional
         beqz     R3, try           ; branch if store fails
```

MIPS Support for Sync

- Example: A parallel loop of *n* iterations

```
count = 1;
while (count <= n){
     count=fetch&add(i,1);
     if(count <= n)
            {/*  execute iteration i of the loop  */}
}
```



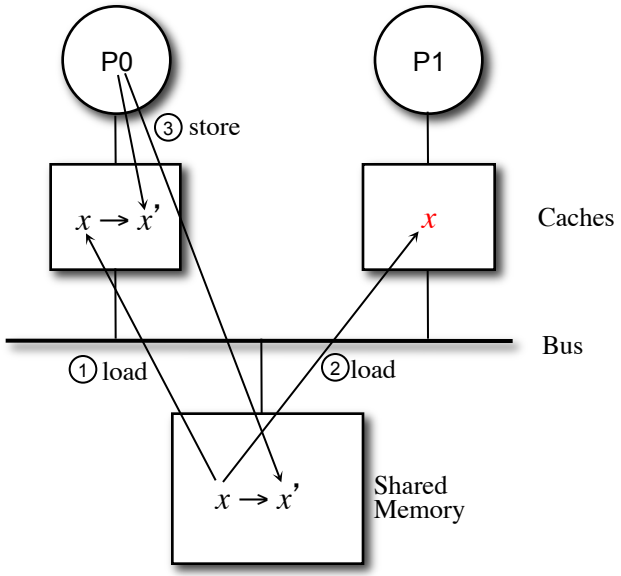
- Since the store comes after another store, the reservation is no longer intact therefore store fails.

Chapter Outline

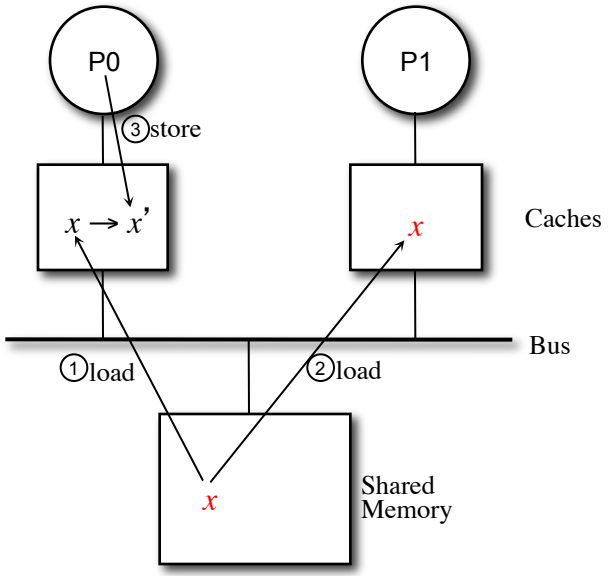
- Introduction
- Parallel Computers
- Shared-Memory Programming
- Synchronization
- Cache Coherence

Cache-Coherence Problem

Sharing of Writable Data



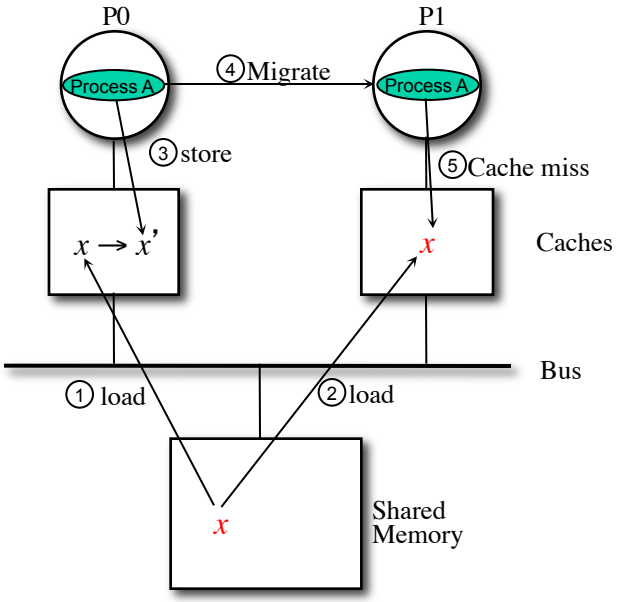
Write-through



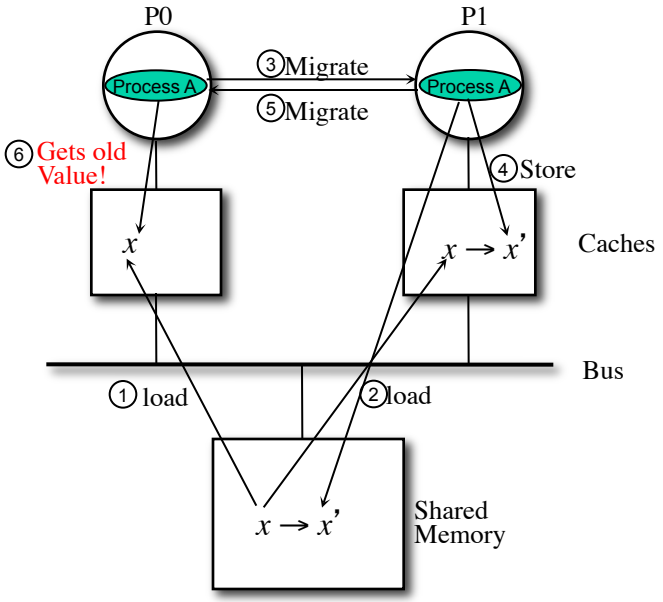
Write-back

Cache-Coherence Problem

Process Migration



Write-back

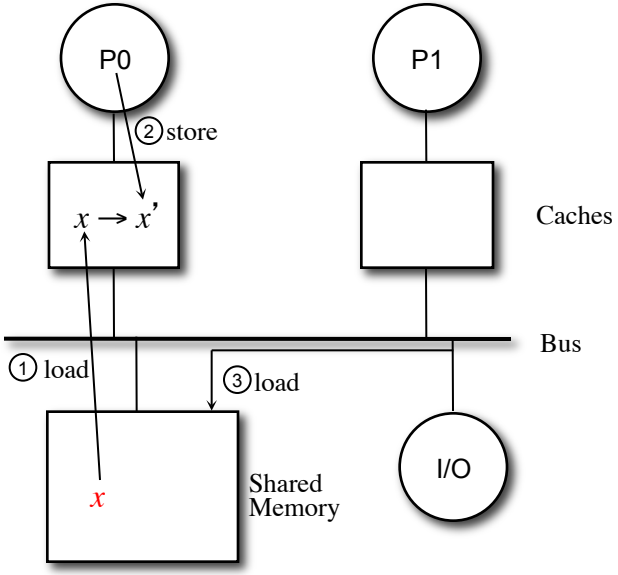


Write-through

Cache-Coherence Problem

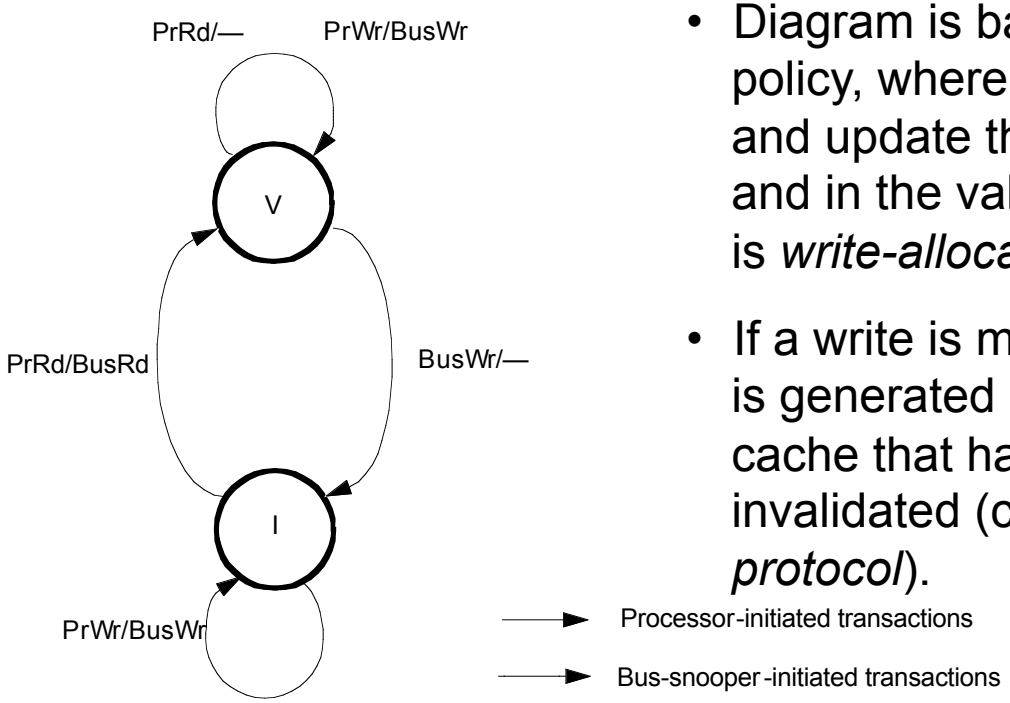
I/O Activity

I/O devices, such as DMA, will access memory directly. However, for this to work, I/O must read memory after it is written back.



Write-back

Cache Coherence for Write-Through



- All processors monitor (snoop) the bus for writes.
- Diagram is based on *no-write allocate* policy, where writes update the memory and update the block only if it is present and in the valid state. The other option is *write-allocate*.
- If a write is made, an invalidate signal is generated on the bus, and other cache that have this block are invalidated (called *write-invalidate protocol*).

CC for Write-Back: MSI Protocol

States

- Invalid (I)
- Shared (S): one or more
- Dirty or Modified (M)

Processor Events:

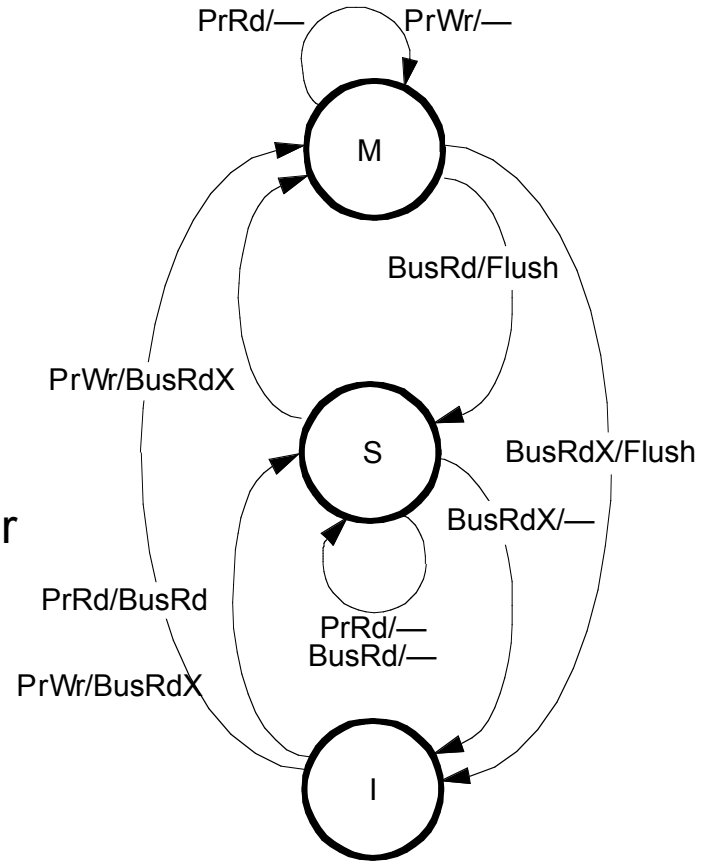
- PrRd (read)
- PrWr (write)

Bus Transactions

- BusRd: Request for a copy with no intent to modify
- BusRdX: Request for a copy with intent to modify. Invalidates all other blocks.
- BusWB: Updates memory

Misc.

- Flush: Cache writes block on bus. Memory and requesting cache both pickup the block.



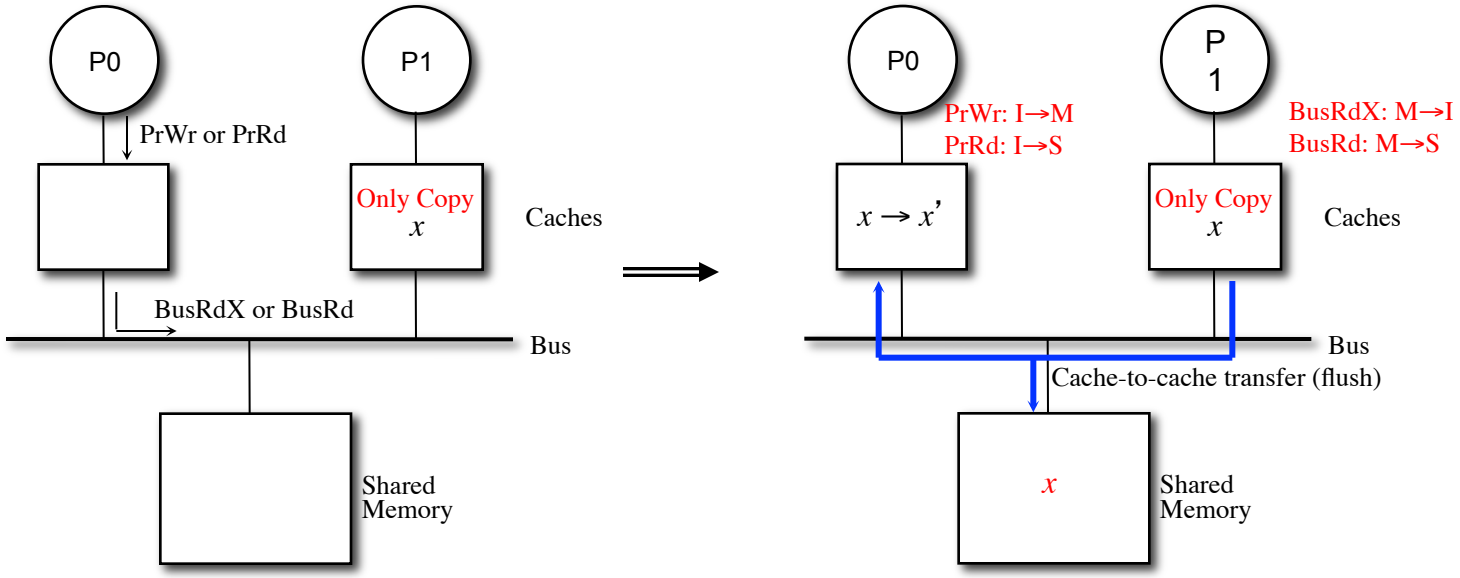
CC for Write-Back: MSI Protocol

Some Issues:

- Flush requires *cache-to-cache transfer*, i.e., both memory and the requesting cache pick up the block.
- Writing into a shared block (S → M): Uses `BusRdX` to acquire exclusive ownership. But, the read block from MM (or another cache) can be ignored since the block is already in the cache (S state). To reduce traffic on the bus, a new transaction, called *bus upgrade* or `BusUpgr`, can be used which simply invalidates other copies but does not cause MM (or another cache) to respond with the data for the block.
- When a processor reads in and modifies a data item (read followed by a write), two bus transactions are generated. First is `BusRd` (I → S) and then followed by `BusRdX` or `BusUpgr` (S → M). By adding a state that indicates that the block is the only copy but not modified, we can eliminate invalidation (`BusRdX` or `BusUpgr`), thus reducing the bus traffic. This new state is called *exclusive-clean*.

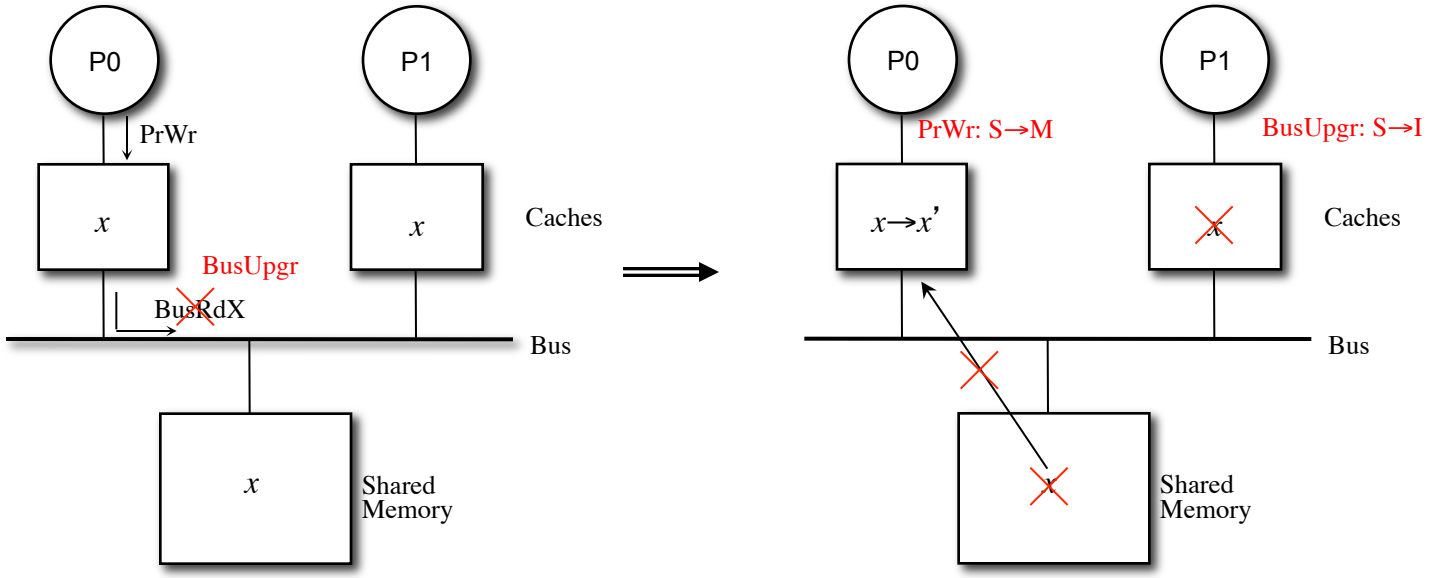
Cache-to-Cache Transfer

Flush requires *cache-to-cache transfer*, i.e., both memory and the requesting cache pick up the block.



Writing into Shared Block

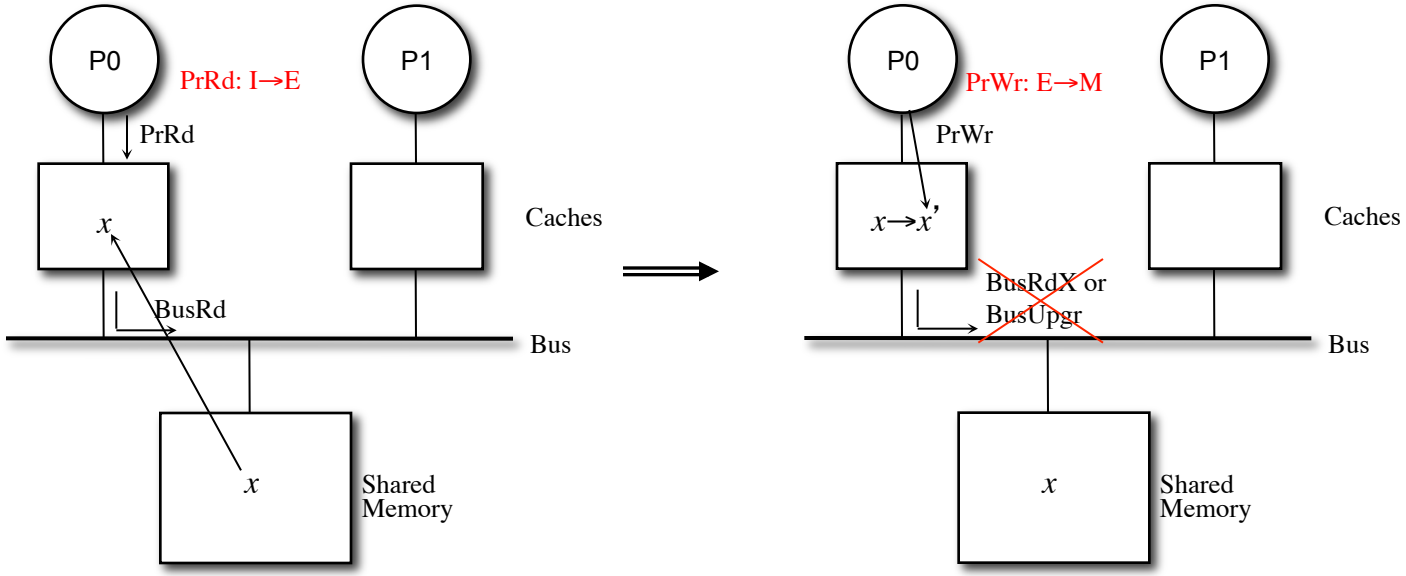
Since block already in cache, uses *bus upgrade* or `BusUpgr` instead `BusRdX` to acquire exclusive ownership. Reduces traffic on the bus.



If it was BusRdX, memory would have had to first respond by return x and the invalidate itself!

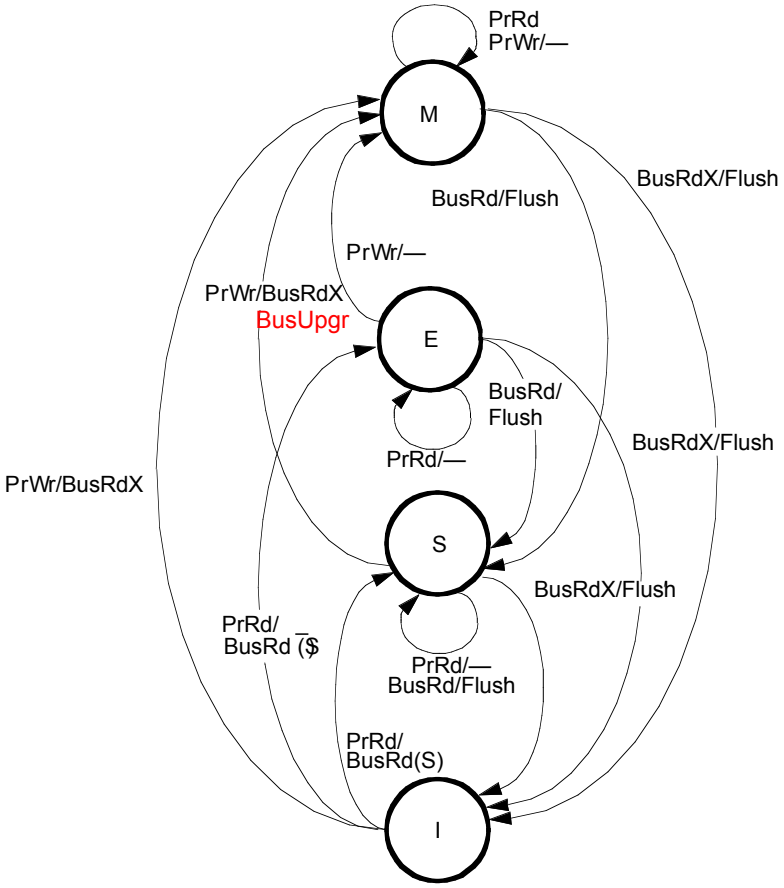
Exclusive-Clean State

When a processor reads in (`BusRd` (I → S)) and modifies a data item (`BusRdX` or `BusUpgr` (S → M)), two bus transactions are generated. By adding a state that indicates that the block is the only copy but not modified, we can eliminate invalidation (`BusRdX` or `BusUpgr`), thus reducing the bus traffic.



MESI Protocol

- MESI eliminates the need to invalidate when writing in the exclusive state. Also refer to as the Illinois protocol.
  - *Modified* - Main memory does not have an up-to-date copy of this cache line. No other cache has a copy of this sector.
  - *Exclusive* - Main memory has an up-to-date copy of this cache line. No other caches hold the line.
  - *Shared* - Main memory has an up-to-date copy of this cache line. Other caches may also have an up-to-date copy.
  - *Invalid* - This cache does not have a valid copy of the sector.

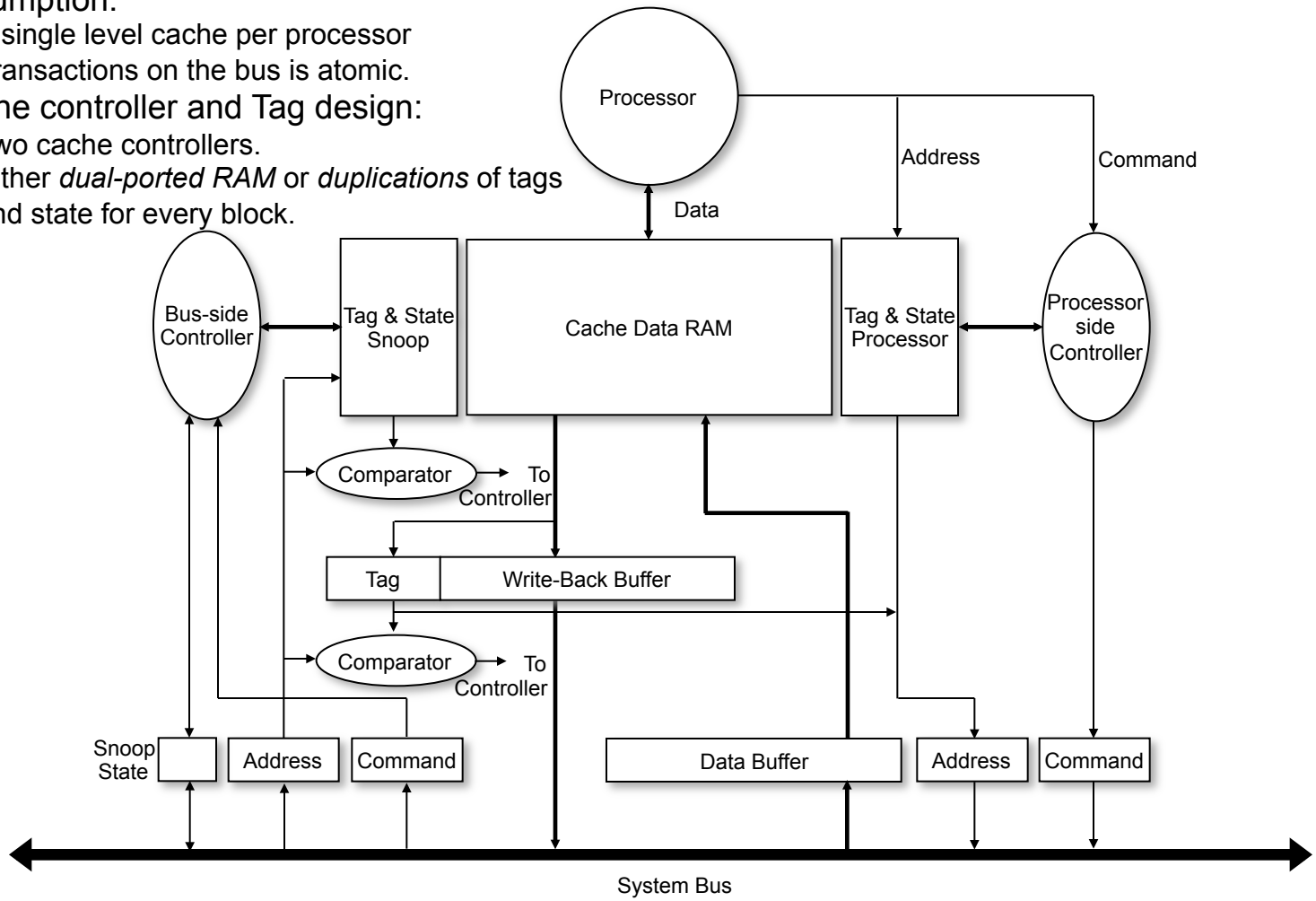


Some Issues

- Transition I → S or I → E depends on whether any other caches have this block. This is detected by a shared signal `S`. `BusRd(S')` means `S` was unasserted, while `BusRd(S)` means `S` was asserted.
- From state E, either `BusRd` (E → S) or `BusRdX` (E → I) causes the block to be flushed onto the bus if cache-to-cache transfer is used.
- From state S, either `BusRd` or `BusRdX` causes one of the processors to flushed the block onto the bus. Since many caches can have the block, a priority scheme is used. `Flush'` simply indicates this operation is only performed by the cache responsible for providing the block. No other caches are involved in the process.

Snoop-Based CC Design

- Assumption:
  - A single level cache per processor
  - Transactions on the bus is atomic.
- Cache controller and Tag design:
  - Two cache controllers.
  - Either *dual-ported RAM* or *duplications* of tags and state for every block.



Reporting Snoop Results

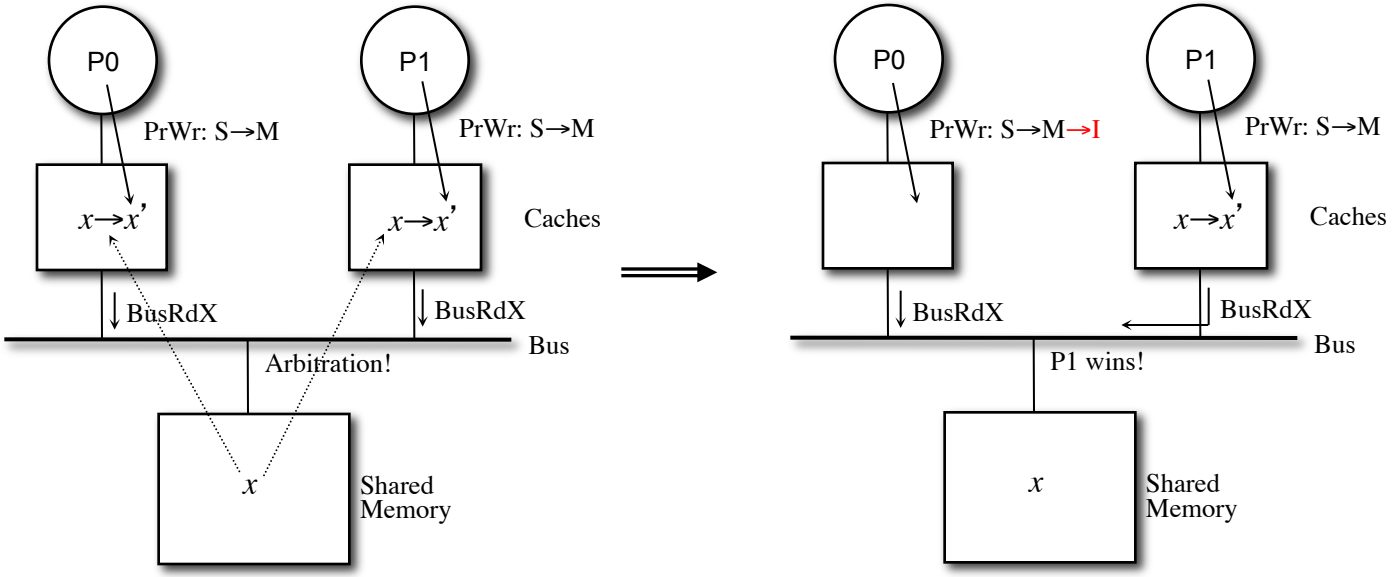
- Bus-side controller for each cache checks the address against its tags, and the collective result of the snoop from all caches must be reported on the bus.
- Requesting cache controller needs to know where the requested block is in other processors' cache so that it can decide to load the block in exclusive (E) state or shared (S) state; and
- Memory system needs to know whether any cache has the block in modified (M) state, in which case the memory need not respond.

Dealing with Write-backs

- Write-backs require two bus transactions (one for incoming and the other for outgoing (modified or dirty) block that is being replaced).
- To speed up the write-back process, a *write-back buffer* is used to temporarily store the block being replaced.
- Before write-back is complete, it's possible to see a bus transaction containing the address of the block being written back. Thus, the controller must supply the data from the write buffer and cancel its earlier pending request to the bus for a write back. This will require an address comparator to be added to snoop on the write-back buffer.

Non-Atomic State Transitions

- Assumption thus far was state transition was atomic!
  - Bus has to arbitrate among multiple requests.
  - Request and response is done using split-transaction



Transient States for Bus Acquisition