

An Introduction to Parallel architectures

Ing. Andrea Marongiu

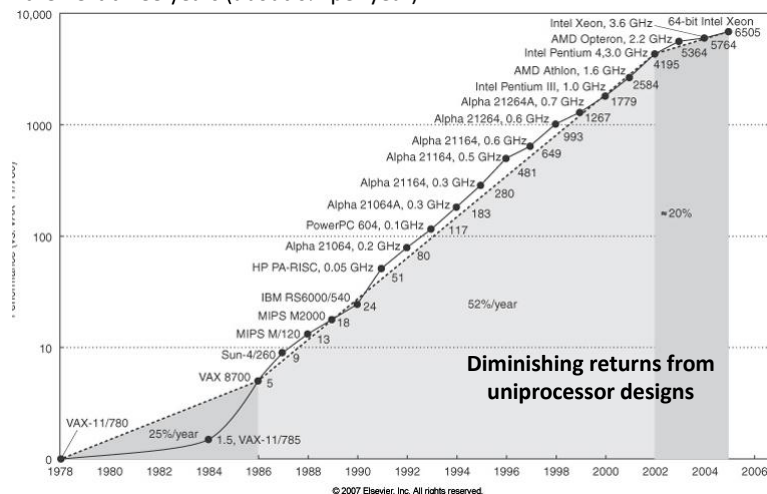
Ing. Andrea Bartolini

{a.marongiu, a.bartolini}@unibo.it

Why Multicores?

What is the Problem?

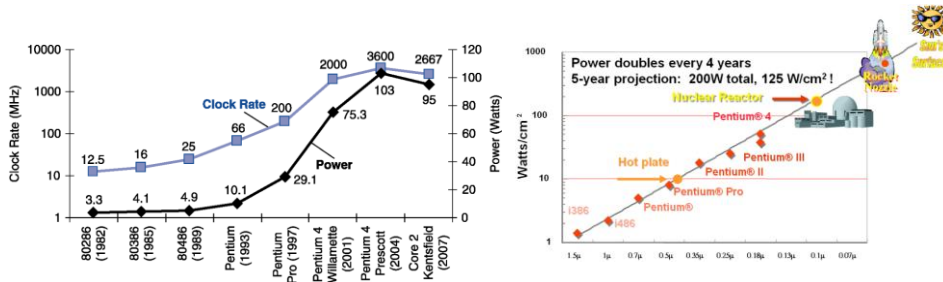
The problem is illustrated in this figure, taken from Patterson & Hennessy. The SPECint performance of the hottest chip grew by 52% per year from 1986 to 2002, and then grew only 20% in the next three years (about 6% per year).



Power Wall

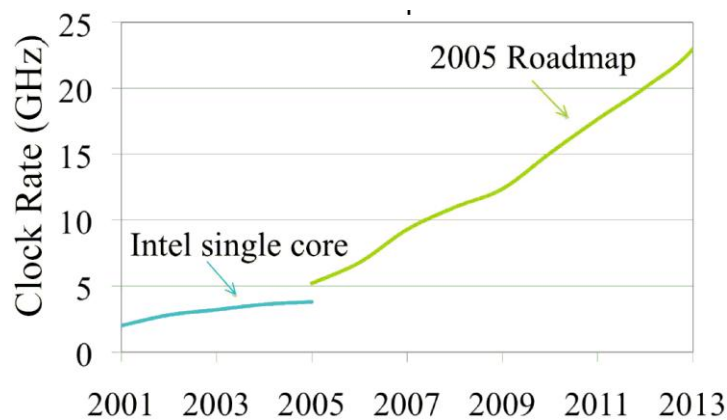
Here is a Clue to the Problem

The problem is now called “the Power Wall”. It is illustrated in this figure, taken from Patterson & Hennessy.



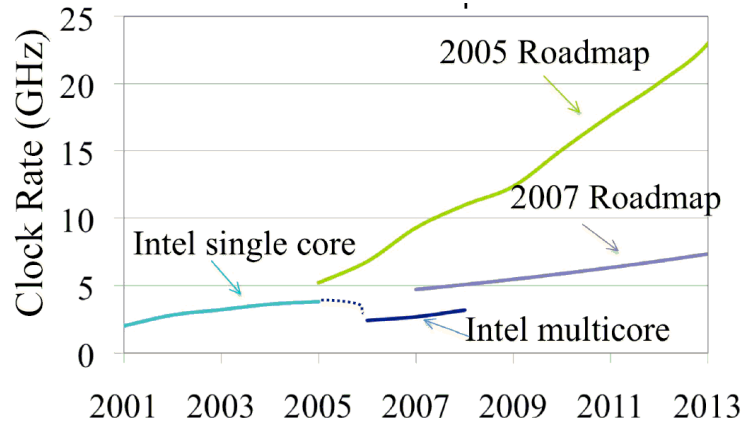
- The design goal for the late 1990's and early 2000's was to drive the clock rate up. This was done by adding more transistors to a smaller chip.
- Unfortunately, this increased the power dissipation of the CPU chip beyond the capacity of inexpensive cooling techniques

Roadmap for CPU Clock Speed: Circa 2005



Here is the result of the best thought in 2005. By 2015, the clock speed of the top “hot chip” would be in the 12 – 15 GHz range.

The CPU Clock Speed Roadmap (A Few Revisions Later)

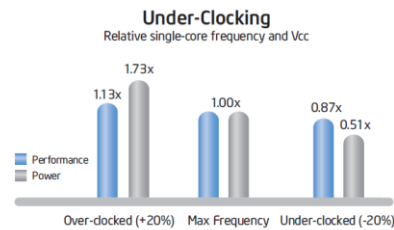


This reflects the practical experience gained with dense chips that were literally “hot”; they radiated considerable thermal power and were difficult to cool.
Law of Physics: All electrical power consumed is eventually radiated as heat.

The MultiCore Approach

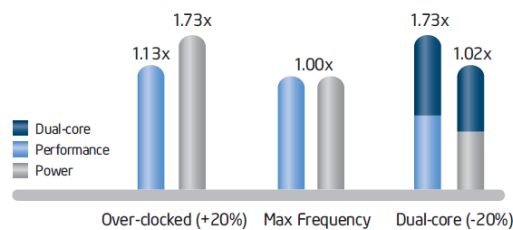
Multiple cores on the same chip

- Simpler
- Slower
- Less power demanding

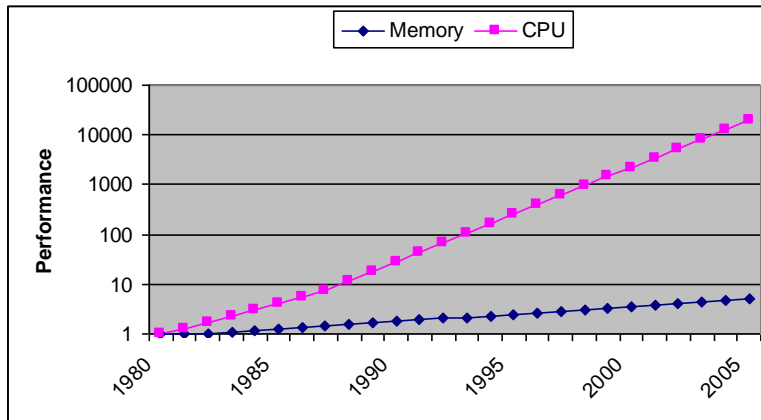


Multi-Core Energy-Efficient Performance

Relative single-core frequency and Vcc



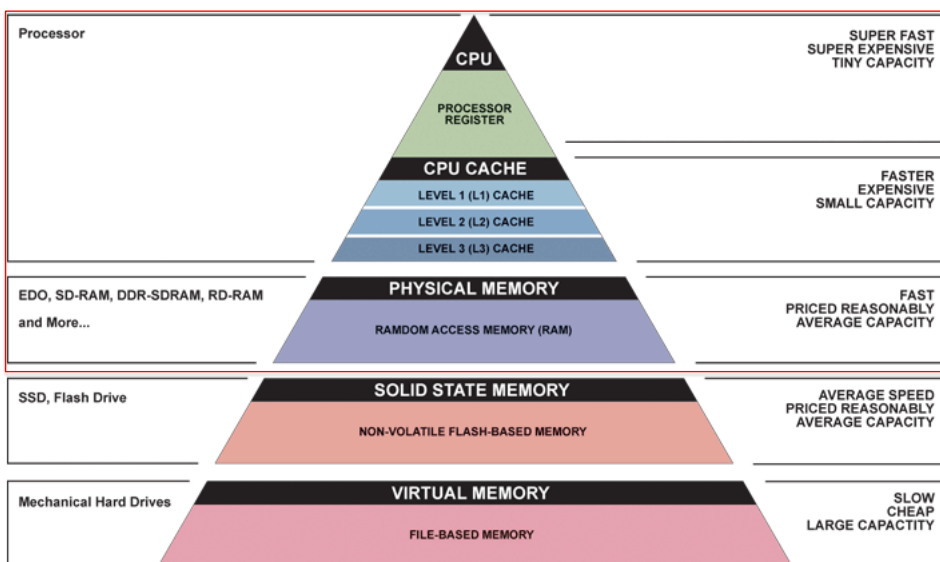
The Memory Gap



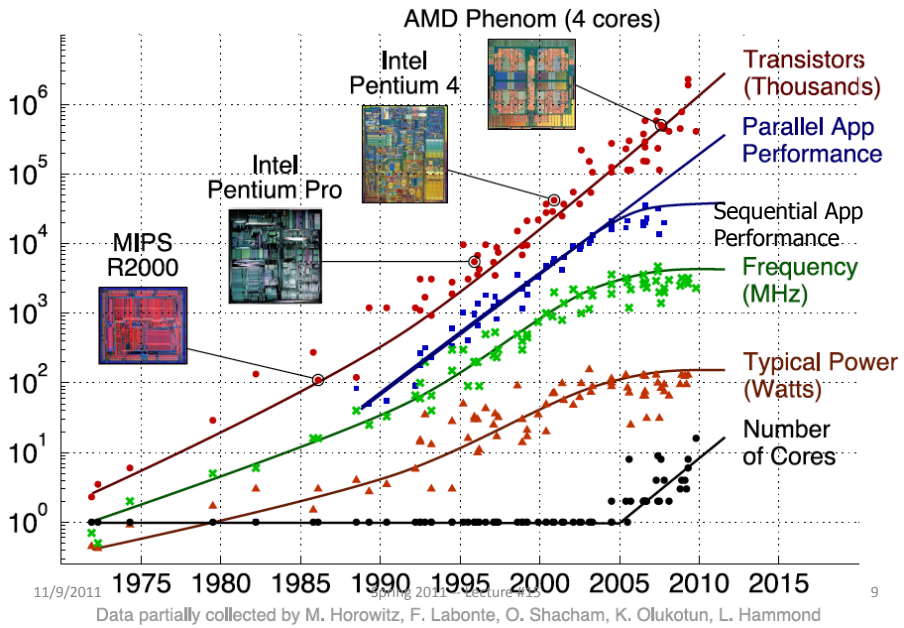
H&P
Fig. 5.2

- **Bottom-line: memory access is increasingly expensive and CA must devise new ways of hiding this cost**

Memory Hierarchy



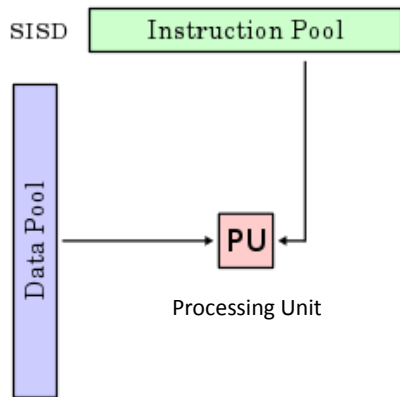
Transition to Multicore



Flynn Taxonomy of parallel computers

		Data streams	
		Single	Parallel
Instruction Streams	Single	SISD	SIMD
	Multiple	MISD	MIMD

Alternative Kinds of Parallelism: Single Instruction/Single Data Stream



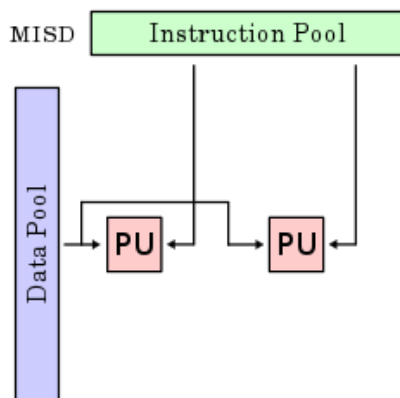
- Single Instruction, Single Data stream (SISD)
 - Sequential computer that exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are traditional uniprocessor machines

11/9/2011

Spring 2011 -- Lecture #13

11

Alternative Kinds of Parallelism: Multiple Instruction/Single Data Stream



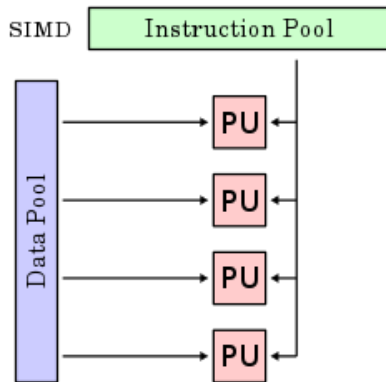
- Multiple Instruction, Single Data streams (MISD)
 - Computer that exploits multiple instruction streams against a single data stream for data operations that can be naturally parallelized. For example, certain kinds of array processors.
 - No longer commonly encountered, mainly of historical interest only

11/9/2011

Spring 2011 -- Lecture #13

12

Alternative Kinds of Parallelism: Single Instruction/Multiple Data Stream



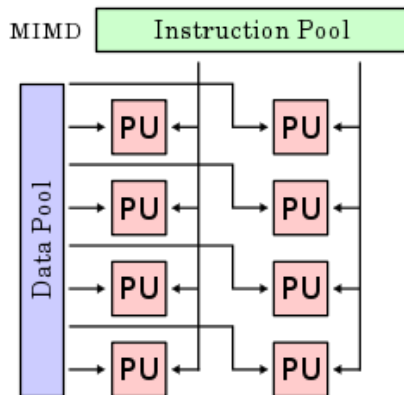
- Single Instruction, Multiple Data streams (SIMD)
 - Computer that exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., SIMD instruction extensions or Graphics Processing Unit (GPU)

11/9/2011

Spring 2011 -- Lecture #13

13

Alternative Kinds of Parallelism: Multiple Instruction/Multiple Data Streams



- Multiple Instruction, Multiple Data streams (MIMD)
 - Multiple autonomous processors simultaneously executing different instructions on different data.
 - MIMD architectures include multicore and Warehouse Scale Computers (datacenters)

11/9/2011

Spring 2011 -- Lecture #13

14

Flynn Taxonomy of parallel computers

		Data streams	
		Single	Parallel
Instruction Streams	Single	SISD : Intel Pentium 4	SIMD : SSE instructions of x86 CELL BE SPEs
	Multiple	MISD : No examples today	MIMD : Intel Nehalem

- In 2011, **SIMD** and **MIMD** most common parallel computers
- Most common parallel processing programming style: **Single Program Multiple Data (“SPMD”)**
 - Single program that runs on all processors of an MIMD
 - Cross-processor execution coordination through conditional expressions (thread parallelism)
- **SIMD (aka hw-level data parallelism): specialized function units, for handling lock-step calculations involving arrays**
 - Scientific computing, signal processing, multimedia (audio/video processing)

11/9/2011

Spring 2011 -- Lecture #13

15

SIMD Architectures

- *Data parallelism*: executing one operation on multiple data streams
- Example to provide context:
 - Multiplying a coefficient vector by a data vector (e.g., in filtering)
$$y[i] := c[i] \times x[i], \quad 0 \leq i < n$$
- Sources of performance improvement:
 - One instruction is fetched & decoded for entire operation
 - Multiplications are known to be independent
 - Pipelining/concurrency in memory access as well

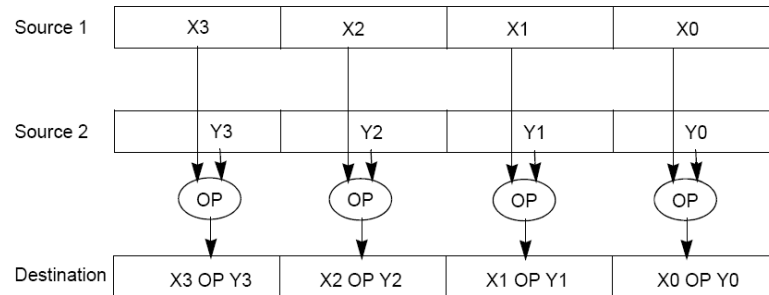
11/9/2011

Spring 2011 -- Lecture #13

Slide 16

“Advanced Digital Media Boost”

- To improve performance, SIMD instructions
 - Fetch one instruction, do the work of multiple instructions



11/9/2011

Spring 2011 -- Lecture #13

17

Example: SIMD Array Processing

```
for each f in array
  f = sqrt(f)
```

```
for each f in array
{
  load f to the floating-point register
  calculate the square root
  write the result from the register to memory
}
```

```
for each 4 members in array
{
  load 4 members to the SIMD register
  calculate 4 square roots in one operation
  write the result from the register to memory
}
```

11/9/2011

Spring 2011 -- Lecture #13

18

Data Level Parallelism and SIMD

- SIMD wants adjacent values in memory that can be operated in parallel
- Usually specified in programs as loops

```
for(i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

- How can reveal more data level parallelism than available in a single iteration of a loop?
- *Unroll loop* and adjust iteration rate

11/9/2011

Spring 2011 -- Lecture #14

19

Loop Unrolling (MIPS)

Assumptions:

- R1 is initially the address of the element in the array with the highest address
- F2 contains the scalar value s
- 8(R2) is the address of the last element to operate on.

```
for(i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

Loop:

```
1. l.d      F0, 0(R1)    ; F0=array element
2. add.d    F4, F0, F2   ; add s to F0
3. s.d      F4, 0(R1)   ; store result
4. addui    R1, R1, #-8  ; decrement pointer 8 byte
5. bne      R1, R2, Loop ; repeat loop if R1 != R2
```

Loop Unrolled

```

Loop: l.d    F0,0(R1)
      add.d  F4,F0,F2
      s.d    F4,0(R1)
      l.d    F6,-8(R1)
      add.d  F8,F6,F2
      s.d    F8,-8(R1)
      l.d    F10,-16(R1)
      add.d  F12,F10,F2
      s.d    F12,-16(R1)
      l.d    F14,-24(R1)
      add.d  F16,F14,F2
      s.d    F16,-24(R1)
      addui  R1,R1,#-32
      bne   R1,R2,Loop
  
```

NOTE:

1. Different Registers eliminate stalls
2. Only 1 Loop Overhead every 4 iterations
3. This unrolling works if $\text{loop_limit} \pmod{4} = 0$

Loop Unrolled Scheduled

```

Loop: l.d    F0,0(R1)
      l.d    F6,-8(R1)
      l.d    F10,-16(R1)
      l.d    F14,-24(R1)
      -----
      add.d  F4,F0,F2
      add.d  F8,F6,F2
      add.d  F12,F10,F2
      add.d  F16,F14,F2
      -----
      s.d    F4,0(R1)
      s.d    F8,-8(R1)
      s.d    F12,-16(R1)
      s.d    F16,-24(R1)
      -----
      addui  R1,R1,#-32
      bne   R1,R2,Loop
  
```

4 Loads side-by-side: Could replace with 4 wide SIMD Load

4 Adds side-by-side: Could replace with 4 wide SIMD Add

4 Stores side-by-side: Could replace with 4 wide SIMD Store

Loop Unrolling

Loop Unrolling can be implemented in the compiler to transform the original C code

```
for(i=1000; i>0; i=i-1)
  x[i] = x[i] + s;
```

into

```
for(i=1000; i>0; i=i-4)
{
  x[ i ] = x[ i ] + s;
  x[i-1] = x[i-1] + s;
  x[i-2] = x[i-2] + s;
  x[i-3] = x[i-3] + s;
}
```

11/9/2011

Spring 2011 -- Lecture #14

23

Generalizing Loop Unrolling

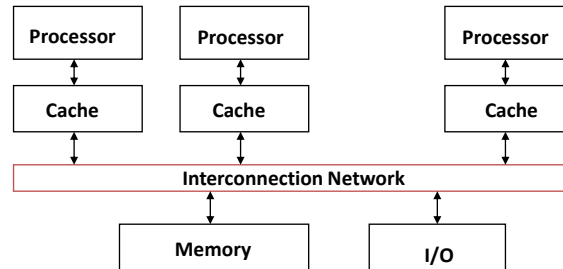
- A loop of **n iterations**
- **k copies** of the body of the loop

Then we will run the loop with 1 copy of the body **$n \bmod k$** times and

with k copies of the body **$\text{floor}(n/k)$** times

Parallel Processing: Multiprocessor Systems (MIMD)

- **Multiprocessor (MIMD)**: a computer system with at least 2 processors



1. Deliver high throughput for independent jobs via job-level parallelism
2. **Improve the run time of a single program that has been specially crafted to run on a multiprocessor - a parallel processing program**

11/9/2011

Spring 2011 -- Lecture #15

25

Multiprocessors

- Only path to performance is parallelism
 - Clock rates flat or declining
 - SIMD: 2X width every 3-4 years
 - 128b wide now, 256b 2011, 512b in 2014?, 1024b in 2018?
 - MIMD: Add 2 cores every 2 years: 2, 4, 6, 8, 10, ...
- A key challenge is to craft parallel programs that have high performance on multiprocessors as the number of processors increase – i.e., that scale
 - Scheduling, load balancing, time for synchronization, overhead for communication (*topic of upcoming lecture*)

11/9/2011

Spring 2011 -- Lecture #15

26

Potential Parallel Performance (Assuming SW can use it!)

Year	Cores	SIMD bits /Core	Core * SIMD bits	Peak DP FLOPs
2003	MIMD 2	SIMD 128	256	MIMD 4
2005	+2/ 4	2X/ 128	512	*SIMD 8
2007	2yrs 6	4yrs 128	768	12
2009	8	128	1024	16
2011	10	256	2560	40
2013	12	256	3072	48
2015	2.5X 14	8X 512	7168	20X 112
2017	16	512	8192	128
2019	18	1024	18432	288
2021	20	1024	20480	320

11/9/2011

Spring 2011 -- Lecture #15

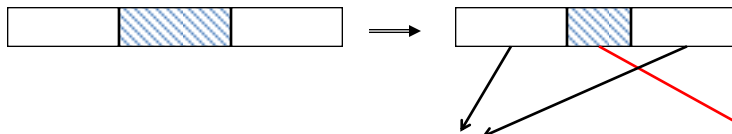
27

Big Idea: Amdahl's (Heartbreaking) Law

- Speedup due to enhancement E is

$$\text{Speedup w/ E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/ E}}$$

- Suppose that enhancement E accelerates a fraction F (F < 1) of the task by a factor S (S > 1) and the remainder of the task is unaffected



$$\text{Execution Time w/ E} = \text{Execution Time w/o E} \times [(1-F) + F/S]$$

$$\text{Speedup w/ E} = 1 / [(1-F) + F/S]$$

11/9/2011

Fall 2010 -- Lecture #17

28

Big Idea: Amdahl's Law

Speedup =

Example: the execution time of half of the program can be accelerated by a factor of 2. What is the program speed-up overall?

11/9/2011

Fall 2010 -- Lecture #17

29

Big Idea: Amdahl's Law

$$\text{Speedup} = \frac{1}{(1 - F) + \frac{F}{S}}$$

Non-speed-up part
Speed-up part

Example: the execution time of half of the program can be accelerated by a factor of 2. What is the program speed-up overall?

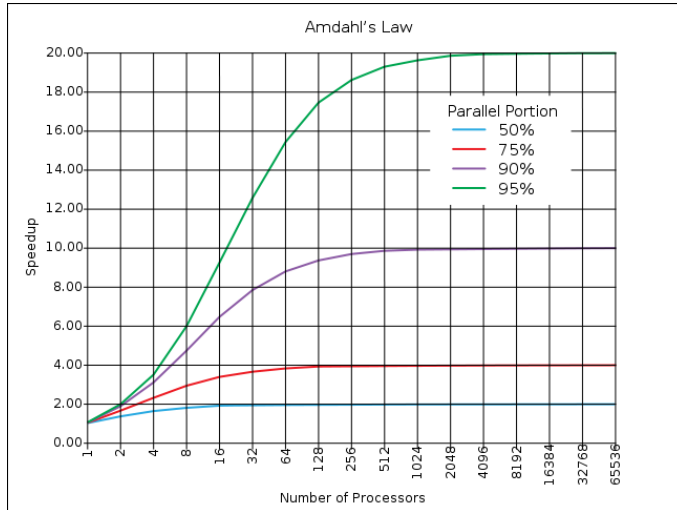
$$\frac{1}{\frac{0.5 + 0.5}{2}} = \frac{1}{0.5 + 0.25} = 1.33$$

11/9/2011

Fall 2010 -- Lecture #17

30

Big Idea: Amdahl's Law



If the portion of the program that can be parallelized is small, then the speedup is limited

The non-parallel portion limits the performance

11/9/2011

Fall 2010 -- Lecture #17

31

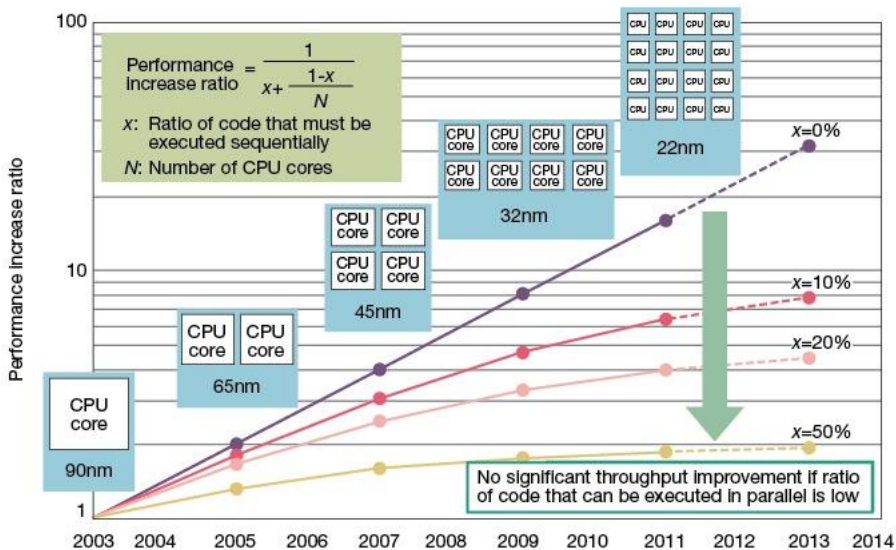


Fig 3 Amdahl's Law an Obstacle to Improved Performance Performance will not rise in the same proportion as the increase in CPU cores. Performance gains are limited by the ratio of software processing that must be executed sequentially. Amdahl's Law is a major obstacle in boosting multicore microprocessor performance. Diagram assumes no overhead in parallel processing. Years shown for design rules based on Intel planned and actual technology. Core count assumed to double for each rule generation.

Strong and Weak Scaling

- To get good speedup on a multiprocessor while keeping the problem size fixed is harder than getting good speedup by increasing the size of the problem.
 - *Strong scaling*: when speedup can be achieved on a parallel processor without increasing the size of the problem
 - *Weak scaling*: when speedup is achieved on a parallel processor by increasing the size of the problem proportionally to the increase in the number of processors

Needed to amortize sources of **OVERHEAD** (*additional code, not present in the original sequential program, needed to execute the program in parallel*)

11/9/2011

Fall 2010 -- Lecture #17

33

Example: Sum Reduction

- Sum 100,000 numbers on 100 processor SMP
 - Each processor has ID: $0 \leq P_n \leq 99$
 - Partition 1000 numbers per processor
 - Initial summation on each processor

```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i = i + 1)
    sum[Pn] = sum[Pn] + A[i];
```

- Now need to add these partial sums
 - Reduction: divide and conquer
 - Half the processors add pairs, then quarter, ...
 - Need to synchronize between reduction steps

11/9/2011

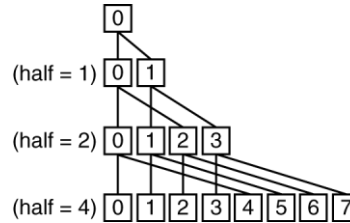
Spring 2011 -- Lecture #15

34

Example: Sum Reduction

```
half = 100;
repeat
```

```
  synch();
  if (half%2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets missing element */
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```



11/9/2011

Spring 2011 -- Lecture #15

35

An Example with 10 Processors

sum[P0] sum[P1] sum[P2] sum[P3] sum[P4] sum[P5] sum[P6] sum[P7] sum[P8] sum[P9]

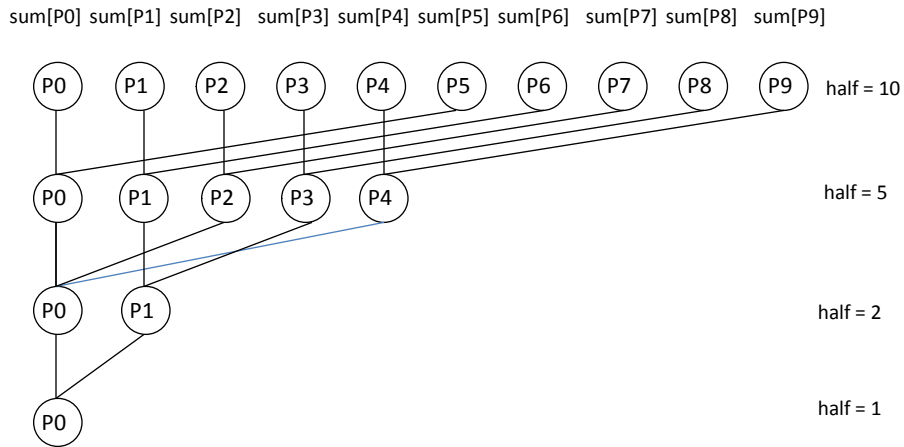


11/9/2011

Spring 2011 -- Lecture #15

36

An Example with 10 Processors



11/9/2011

Spring 2011 -- Lecture #15

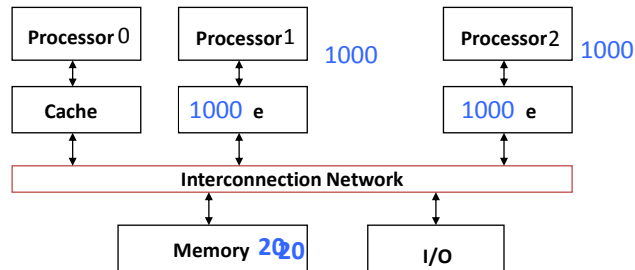
37

The memory wall

- How do multicores attach the memory wall?
- By building on-chip cache hierarchies
- Area
- Coherence
- Scalability

Shared Memory and Caches

- What if?
 - Processors 1 and 2 read Memory[1000] (value 20)



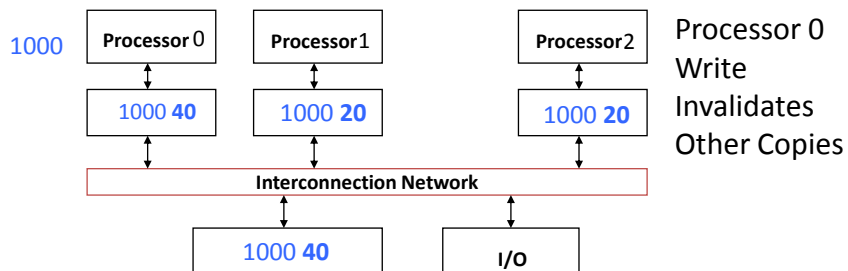
11/9/2011

Spring 2011 -- Lecture #15

39

Shared Memory and Caches

- What if?
 - Processors 1 and 2 read Memory[1000]
 - Processor 0 writes Memory[1000] with 40



11/9/2011

Spring 2011 -- Lecture #15

40

Keeping Multiple Caches Coherent

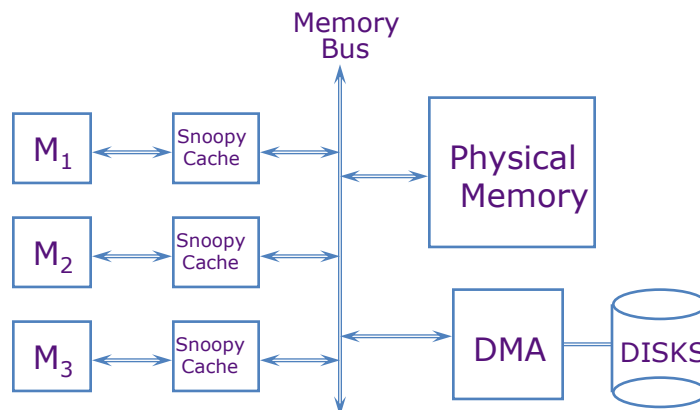
- **Architect's job:** shared memory => keep cache values coherent
- **Idea:** When any processor has cache miss or writes, notify other processors via interconnection network
 - If only reading, many processors can have copies
 - If a processor writes, invalidate all other copies
- Shared written result can “ping-pong” between caches

11/9/2011

Spring 2011 -- Lecture #15

41

Shared Memory Multiprocessor



Use snoopy mechanism to keep all processors' view of memory coherent

42

Snoopy Cache Coherence Protocols

write miss:

the address is *invalidated* in all other caches *before* the write is performed

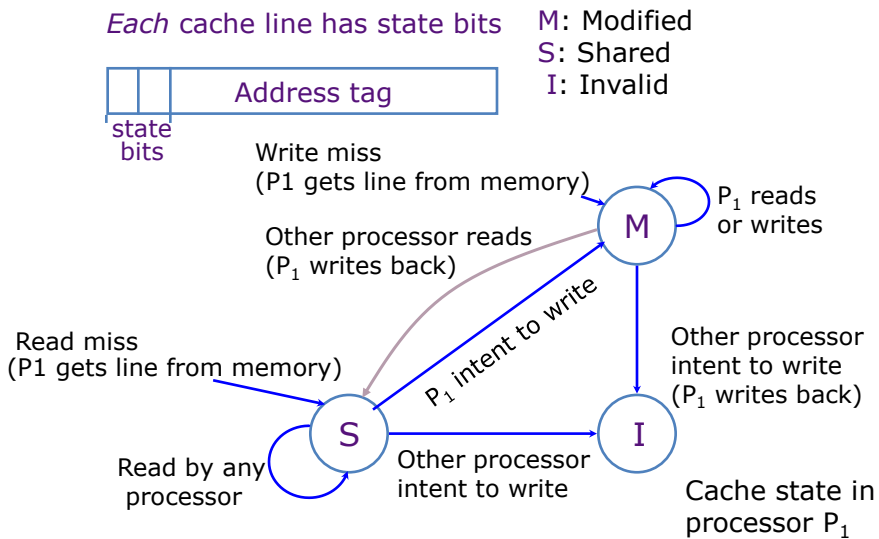
read miss:

if a dirty copy is found in some cache, a write-back is performed before the memory is read

43

Cache State Transition Diagram

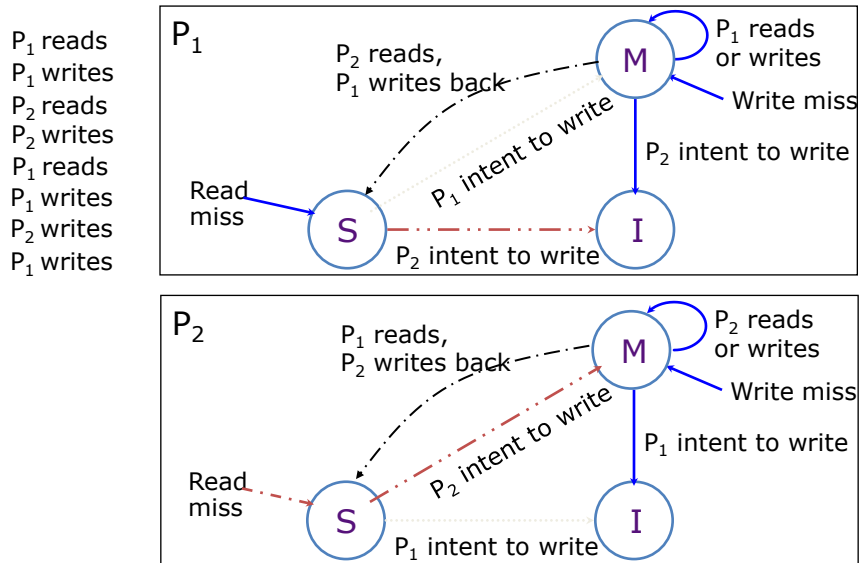
The MSI protocol



44

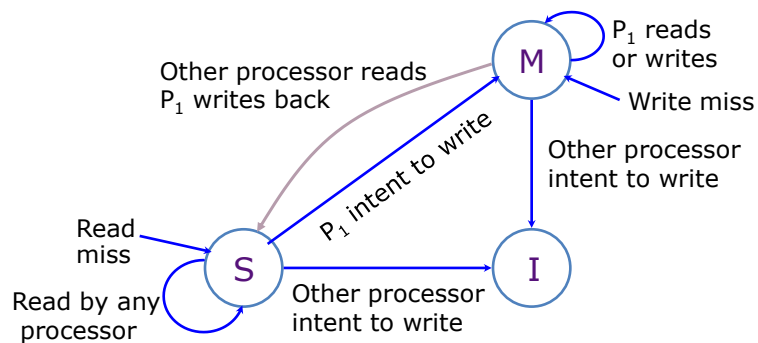
Two Processor Example

(Reading and writing the same cache line)



45

Observation



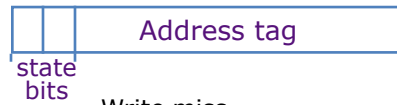
- If a line is in the M state then no other cache can have a copy of the line!
 - Memory stays coherent, multiple differing copies cannot exist

46

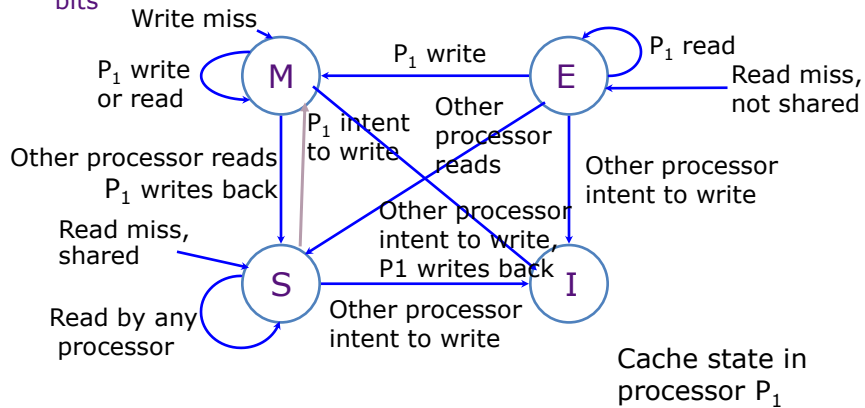
MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag



- M: Modified Exclusive
- E: Exclusive but unmodified
- S: Shared
- I: Invalid

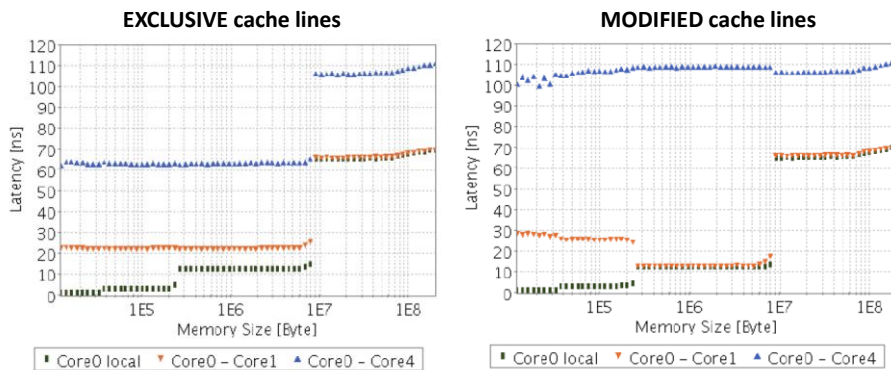


Cache state in processor P₁

47

Coherency scalability

- Cache coherency overhead in manycore can be seen directly in a 4 core general purpose multicore.
- Increasing the number of cores accessing the same cache reduces the performance of it.



Performance of Symmetric Shared-Memory Multiprocessors

Cache performance is combination of:

1. Uniprocessor cache miss traffic
2. Traffic caused by communication
 - Results in invalidations and subsequent cache misses
- Adds 4th C: *coherence miss*
 - Joins Compulsory, Capacity, Conflict
 - (Sometimes called a *Communication miss*)

49

Coherency Misses

1. **True sharing misses** arise from the communication of data through the cache coherence mechanism
 - Invalidates due to 1st write to shared block
 - Reads by another CPU of modified block in different cache
 - Miss would still occur if block size were 1 word
2. **False sharing misses** when a block is invalidated because some word in the block, other than the one being read, is written into
 - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
 - Block is shared, but no word in block is actually shared
 - ⇒ miss would not occur if block size were 1 word

50

False Sharing

state	blk addr	data0	data1	...	dataN
-------	----------	-------	-------	-----	-------

A cache block contains more than one word

Cache-coherence is done at the block-level and not word-level

Suppose M_1 writes $word_i$ and M_2 writes $word_k$ and both words have the same block address.

What can happen?

51

Example: True v. False Sharing v. Hit?

- Assume x1 and x2 in same cache block.
P1 and P2 both read x1 and x2 before.

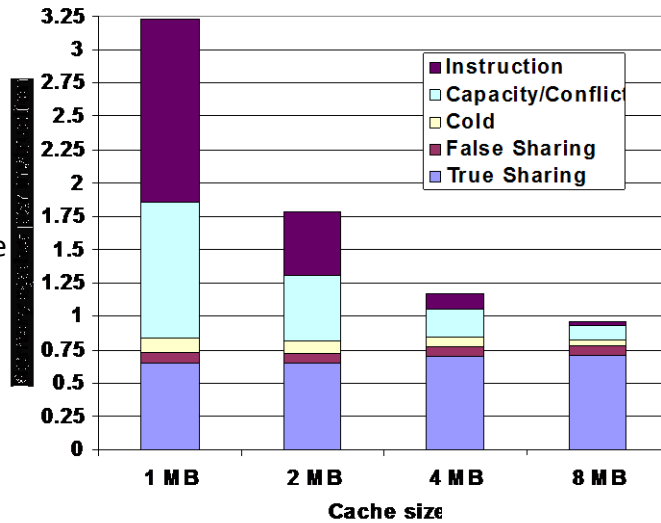
Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	False miss; x1 irrelevant to P2
5	Read x2		True miss; invalidate x2 in P1

52

MP Performance 4 Processor Commercial Workload: OLTP, Decision Support (Database), Search Engine

- True sharing and false sharing unchanged going from 1 MB to 8 MB (L3 cache)

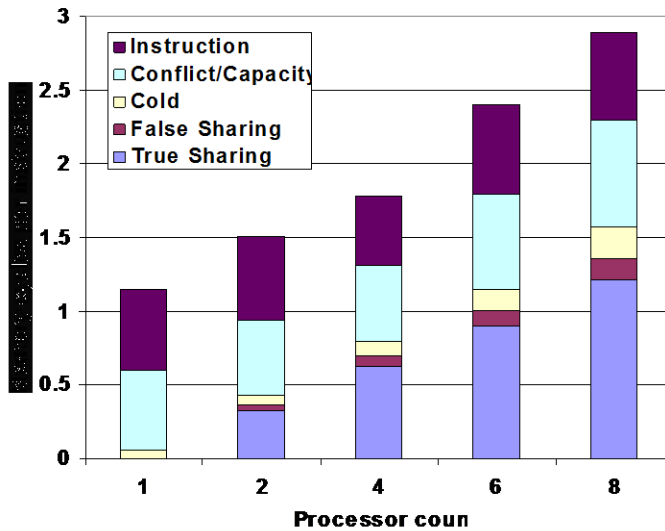
- Uniprocessor cache misses improve with cache size increase (Instruction, Capacity/Conflict, Compulsory)



53

MP Performance 2MB Cache Commercial Workload: OLTP, Decision Support (Database), Search Engine

- True sharing, false sharing increase going from 1 to 8 CPUs



54

A Cache Coherent System Must:

- Provide set of states, state transition diagram, and actions
- Manage coherence protocol
 - (0) Determine when to invoke coherence protocol
 - (a) Find info about state of address in other caches to determine action
 - whether need to communicate with other cached copies
 - (b) Locate the other copies
 - (c) Communicate with those copies (invalidate/update)
- (0) is done the same way on all systems
 - state of the line is maintained in the cache
 - protocol is invoked if an “access fault” occurs on the line
- Different approaches distinguished by (a) to (c)

55

Bus-based Coherence

- All of (a), (b), (c) done through broadcast on bus
 - faulting processor sends out a “search”
 - others respond to the search probe and take necessary action
- Could do it in scalable network too
 - broadcast to all processors, and let them respond
- Conceptually simple, but broadcast doesn’t scale with number of processors, P
 - on bus, bus bandwidth doesn’t scale
 - on scalable network, every fault leads to at least P network transactions
- Scalable coherence:
 - can have same cache states and state transition diagram
 - different mechanisms to manage protocol

56

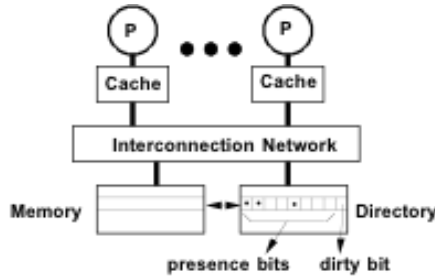
Need for a more scalable protocol

- Snoopy schemes do not scale because they rely on broadcast
- Hierarchical snoopy schemes have the root as a bottleneck
- **Directory** based schemes allow scaling
 - They avoid broadcasts by keeping track of all CPUs caching a memory block, and then using point-to-point messages to maintain coherence
 - They allow the flexibility to use any scalable point-to-point network

Scalable Approach: Directories

- Every memory block has associated directory information
 - keeps track of copies of cached blocks and their states
 - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
 - in scalable networks, communication with directory and copies is through network transactions
- Many alternatives for organizing directory information

Basic Operation of Directory

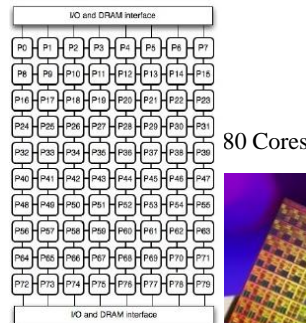
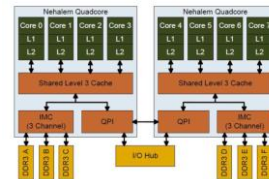


- k processors.
- With each cache-block in memory: k presence-bits, 1 dirty-bit
- With each cache-block in cache: 1 valid bit, and 1 dirty (owner) bit

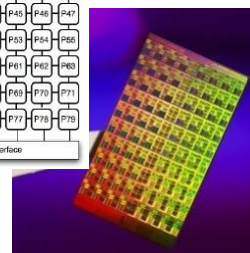
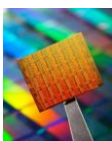
- Read from main memory by processor i:
 - If dirty-bit OFF then { read from main memory; turn p[i] ON; }
 - If dirty-bit ON then { recall line from dirty proc (downgrade cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to i; }
- Write to main memory by processor i:
 - If dirty-bit OFF then {send invalidations to all caches that have the block; turn dirty-bit ON; supply data to i; turn p[i] ON; ... }

Off-chip bandwidth

- As number of integrated cores increases, more off-chip bandwidth is required.
 - Many core => Many Caches => Off chip bandwidth
- Nehalem solution:
 - 1 MC on Chip
- Polaris (80cores)
 - 2 MC on chip
- SCC (48 core)
 - 4MC on chip



80 Cores



The path towards ManyCores

- How to overcome the scalability bottlenecks?
- Approaches to scale to 100s (1000s) of cores
 1. Symmetric architectures
 - On-chip clusters → **Intel Single-Chip Cloud Computer (SCC)**
 2. Asymmetric architectures
 - Control core(s) + accelerators → **ST Microelectronics p2012**
 3. Data parallel, bandwidth-oriented architectures
 - GPGPUs → **NVIDIA Tesla, Fermi**