

Multiprocessors and Parallel Computing

- Preliminary questions:
 - What do we use computers for?
 - What is computing all about?
- Here's another question:
 - Why would one want to have more than one processor working at the same problem at the same time?
- Hypothesis:
 - The idea is that if it takes time T to finish a task using one processor, it will take time T/N to accomplish the same task using N processors. Right?
- Note: Kind of, but not really. Recall *Amdahl's Law* (the law of diminishing returns):

*Manipulating INFORMATION
or processing DATA*

*Execution time after improvement =
(Execution time affected by improvement / Amount of improvement) +
Execution time unaffected*

Multiprocessors and Parallel Computing

Conclusion: In order to see *speedup* equal to the number of processors, the application must be *fully parallelizable*. In other words, the computation must have *no sequential component* at all.

Multiprocessors and Parallel Computing

Obviously, parallel computing went on despite the performance bounds predicted in *Amdahl's Law*.

The bigger the problem, the bigger the gains...

Keyword in multiprocessing: **SCALABILITY**

Multiprocessors and Parallel Computing

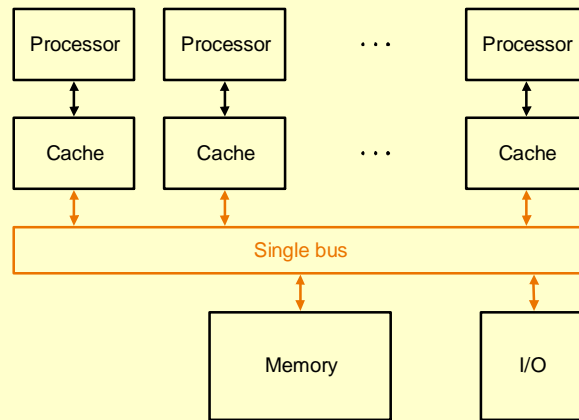
- **Idea:** create powerful computers by connecting many smaller ones
 - good news:** works for timesharing (better than supercomputer)
vector processing may be coming back
 - bad news:** its really hard to write good concurrent programs
many commercial failures
- **How do parallel processors share data?**
 - single address space (SMP vs. NUMA)
 - message passing
- **How do parallel processors coordinate?**
 - synchronization (locks, semaphores)
 - built into send / receive primitives
 - operating system protocols
- **How are they implemented?**
 - connected by a single bus
 - connected by a network



(See Figure 9.2)

Shared-Memory Architecture: Configuration

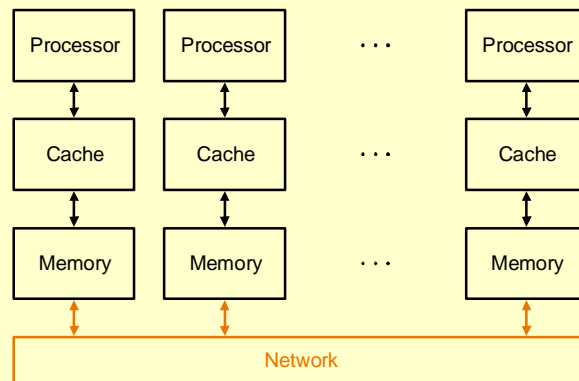
- What possible problems need to be resolved in this picture?



(See Figure 9.8)

Message-Passing Architecture : Configuration

- What possible problems need to be resolved in this picture?



Shared-Memory Architecture: Classification

- **SMP = Symmetric Multiprocessor**
 - all memory is equally close to all processors
 - typical interconnection network is a *shared bus*
 - easier to program, but doesn't scale to many processors
 - **Example:** Quad-processor Intel Pentium Pro (circa 1997)
 - SMP, bus interconnection
 - 4 x 200 MHz Intel Pentium Pro processors
 - 8 + 8 Kb L1 cache per processor
 - 512 Kb L2 cache per processor
 - Snoopy cache coherence
 - Compaq, HP, IBM, NetPower
 - Windows NT, Solaris, Linux, etc.

Shared-Memory Architecture: Classification



UMA:
Uniform
Memory
Access

Same time, no matter which processor, no matter what address is accessed (SMP).



NUMA:
Non-Uniform
Memory
Access

Time depends on which processor is asking for the data and where the data is in memory.

- **NUMA = Non-Uniform Memory Access**
 - each memory is closer to some processors than others
 - *a.k.a.* “Distributed Shared Memory”
 - typically interconnection is *grid* or *hypercube*
 - harder to program, but scales to more processors

Shared-Memory Architecture: Classification

- **NUMA**, *continued ...*
 - **Example: SGI Origin 2000 (circa 1997)**
 - NUMA, hypercube interconnection
 - Up to 128 (64 x 2) MIPS R10000 processors
 - 32 + 32 Kb L1 cache per processor
 - 4 Mb L2 cache per processor
 - Distributed directory-based cache coherence
 - Automatic page migration/replication
 - SGI IRIX with Pthreads

Message-Passing vs. Shared-Memory Architecture

- Shared-memory *programming model* is *easier* because data transfer is handled automatically.
 - **Proof:** message passing can be *efficiently* implemented on a shared memory system, but not vice versa.
- How much of shared-memory programming model should be implemented in hardware?
- How efficient is shared-memory programming model?
- How well does shared-memory scale?
- Does *scalability* really matter?

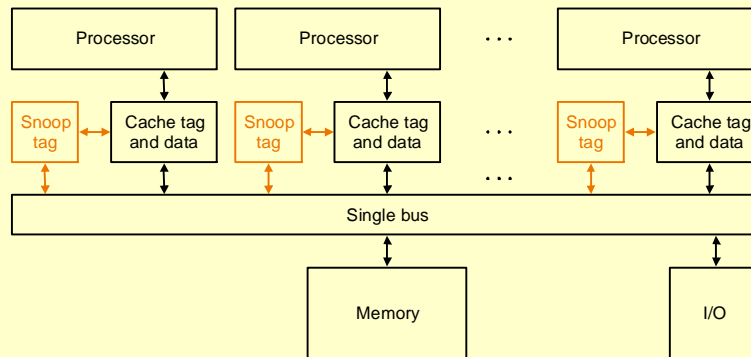
Shared-Memory Architecture: Cache Coherence

- Effective caching reduces memory contention.
- Processors must see single consistent memory.
- Many different consistency models.
- Weak consistency is sufficient.
- *Snoopy cache coherence* for bus-based SMPs.
- *Distributed directories* for NUMA.
- Many implementation issues: multiple-levels, I-D separation, cache line size, update policy, etc. etc.
- Usually don't need to know all the details.

Some Interesting Problems

(See Figure 9.4)

- **Cache Coherency**



- **Synchronization**
— provide special atomic instructions (test-and-set, swap, etc.)
- **Network Topology**

Multiprocessor Cache Coherency

- **Snooping** - the most popular protocol to maintain *cache coherency*
 - distributes responsibility of maintaining cache coherence among all of the cache controllers in a multiprocessor
 - on any miss, all caches check to see if they have a copy of the requested block ...
 - **reads**: processors need most recent copy of shared data
 - **writes**: requires exclusive access, then
 - invalidate all other copies; or
 - update shared copies with the value being written
- **Snooping protocols**:
 - *write-invalidate*
 - allows multiple readers but only a single writer)
 - *write-update*
 - a.k.a. write-broadcast; allows multiple readers and multiple writers

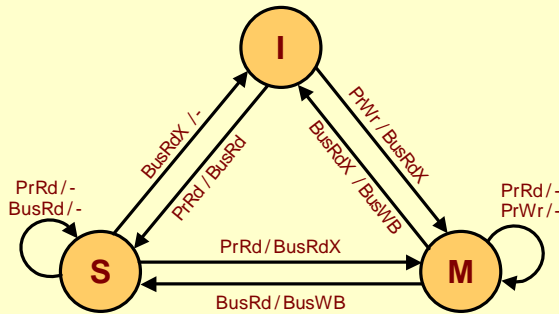
Multiprocessor Cache Coherency

- **Write-invalidate vs. write-update ...**
 - comparable under all circumstances
 - performance dependent on
 - number of local caches
 - pattern of memory reads and writes
- **Standard: write-back cache with write-invalidate protocol**
 - write-back reduces bus traffic and thereby allows more processors on a single bus
 - write-invalidate uses the bus only on the *first* write to invalidate other copies; hence, subsequent writes do not result in bus activity

Multiprocessor Cache Coherency

(From B. Falsafi, 1999)

- **Example:** The *MSI* write-invalidate protocol
 - used by SGI 4D
 - defines the following *cache block states* (using two *status bits* in the cache tag):
 - **Modified:** Read/Write; dirty (written) and may *not* be shared
 - **Shared:** Read Only, clean (not written) and may be shared
 - **Invalid:** does not have valid data



Legend:

PrRd: processor read (load)
 PrWr: processor write (store)
 BusRd: read only copy due to a PrRd
 BusRdX: writable copy due to a PrWr
 BusWB: writing back a block
 BusInv: invalidate other copies
 BusCache: cache-to-cache block transfer
 BusUpdate: one/two word update

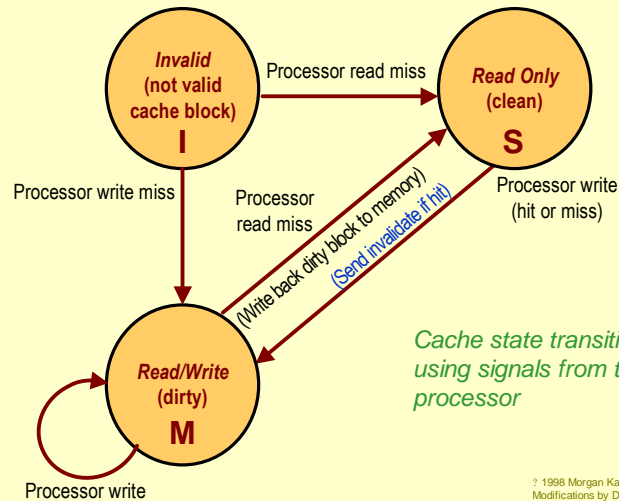
Chapter 9

© 1998 Morgan Kaufmann Publishers
 Modifications by Dr. J. ? 2001 Cal State Univ, Chico 15

Multiprocessor Cache Coherency

(See Figure 9.5)

- **Example:** The *MSI* write-invalidate protocol, *continued ...*
 - Here's *Part 1* of the protocol presented in our textbook:



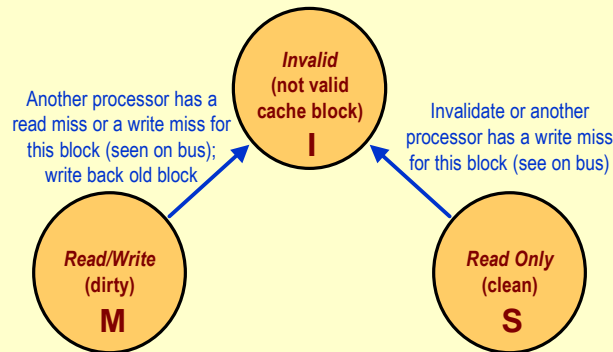
Chapter 9

© 1998 Morgan Kaufmann Publishers
 Modifications by Dr. J. ? 2001 Cal State Univ, Chico 16

Multiprocessor Cache Coherency

(See Figure 9.5)

- **Example:** The *MSI* write-invalidate protocol, *continued ...*
 - Here's *Part 2* of the protocol presented in our textbook:



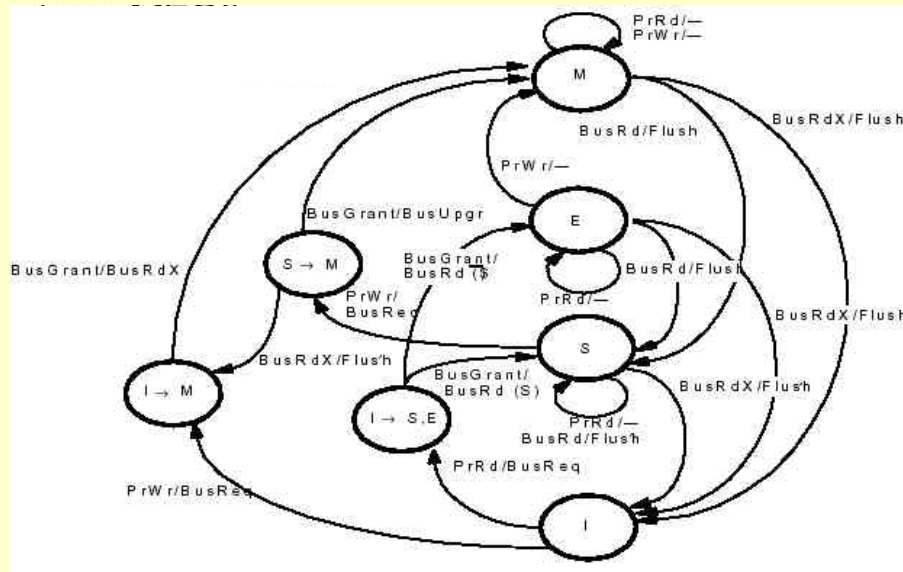
- **Note:** More complex models, requiring additional cache block states, are also available.

Multiprocessor Cache Coherency

(From B. Falsafi, 1999)

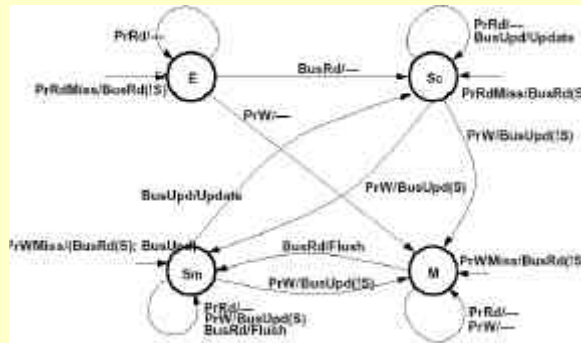
- **Example:** The *MESI* write-invalidate protocol
 - used by Pentium and PowerPC
 - defines the following *cache block states*:
 - **Modified:** cache block modified (different from memory) and is available only in this cache
 - **Exclusive:** cache block same as main memory and is not present in any other cache
 - **Shared:** cache block same as main memory and may be present in another cache
 - **Invalid:** cache block does not contain valid data
 - has **non-atomic state transitions**
 - requires two types of states:
 - stable
 - transient or intermediate
 - increases complexity
 - operations involve multiple hardware transitions

(From B. Falsafi, 1999)



Multiprocessor Cache Coherency (From A. Krishnamurthy, 2000)

- **Example:** The *Dragon* write-back update protocol
 - defines the following *cache block states*:
 - **Exclusive-clean (E):** MyProc and Memory have it
 - **Shared-clean:** MyProc, other procs, and memory have it
 - **Shared-modified:** MyProc and other procs have it, memory does not
 - **Modified:** MyProc has it, no one else



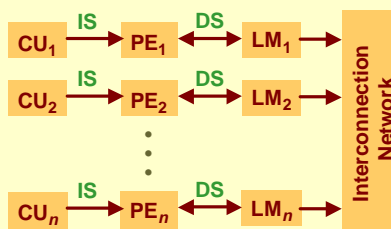
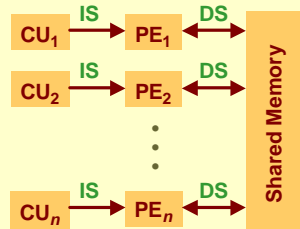
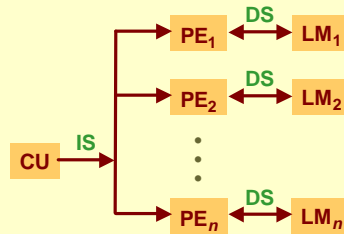
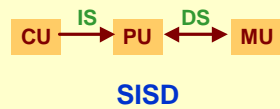
Multiprocessors and Parallel Computing

- What type of “parallelism” is meant in *parallel computing*?
 - low-level parallelism (single processor)
 - instruction pipelining
 - multiple processor functional units
 - separate specialized processors
 - high-level parallelism (multiple processors)
 - taxonomy introduced by M. Flynn (1972):
 - single instruction single data (SISD) stream
 - single instruction multiple data (SIMD) stream
 - e.g. vector and array processors
 - multiple instruction single data (MISD) stream
 - Note: this structure has never been implemented ...
 - multiple instruction multiple data (MIMD) stream
 - shared memory (multiprocessor)
 - distributed memory (multicomputer)

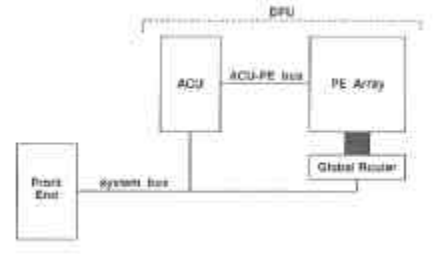
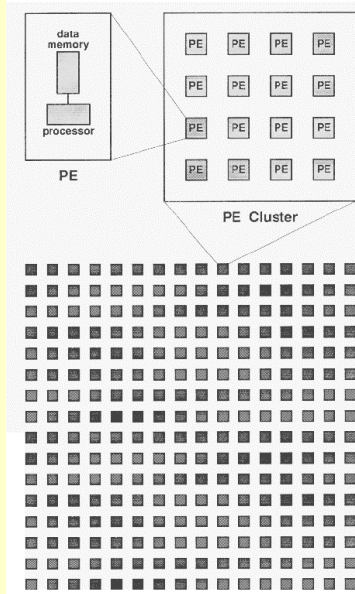
parallel processors

Multiprocessors and Parallel Computing

- Flynn's (1972) taxonomy:



Example SIMD Computer: The MASP



ACU: array control unit; issues instructions to all the PEs (RISC)
PEs: clusters of 32-bit ALUs; 64KB memory, 64 32-bit registers
Topology: grid connection
Scalability: 1024, 2048, 4096, 8192 or 16384 processors
Target: great for data parallel applications

Example SIMD: The Connection Machine CM-2



A 5 feet tall cube formed of smaller cubes, representing a 12-dimensional hypercube structure of the network that connected the processors together.

“This hard geometric object, black, the non-color of sheer, static mass, was transparent, filled with a soft, constantly changing cloud of lights from the processor chips, red, the color of life and energy. It was the archetype of an electronic brain, a living, thinking machine.”

Example MIMD: The SGI Origin 2000



Expandable and flexible rack design: add processors as needs grow. Uses cc-NUMA building blocks to scale the single shared-memory system from 2 to 16 processors in a single rack.

Each module supports two to eight MIPS® processors and up to 16GB of memory and provides I/O bandwidth of 6.24GB per second.

“Capable of connecting with multiple racks to scale to 64 processors in a single-system image utilizing the revolutionary NUMalink™ interconnect, a high-speed, scalable interconnect fabric that provides incremental bandwidth while maintaining the shared-memory model of an SMP server.”

Example MIMD: The Sun Enterprise 6500



For high-end, mission-critical enterprise applications, this 30-processor system combines excellent performance with unmatched availability features and award-winning system management tools. Its modular design makes it easy to expand capacity online. Ideal for demanding Internet and data center applications.

Key Specifications: Up to 30 CPUs, maximum memory of 60 GB (SMP style shared memory), RAID disks.

Key Benefits:

A highly expandable system that offers mission-critical performance and availability.

Additional Notes ...

- Affordable, cost-effective MIMD computers ...
 - **Beowulf clusters:** parallel computer built from off-the-shelf hardware; use of freely available operating systems such as *Linux*, message passing software such as *PVM* and *MPI*, and other software often contributed by Beowulf users.
- **Question:** Does multiprocessing really solve the *performance problem*?
- **Answer:** It has been decades since research on *parallel processing* started, and programming a multiprocessor is still a hard task (remember Amdahl's law!)
 - **Problem areas:**
 - Communication (it takes time to transfer data around)
 - Synchronization (do we have to agree on time?)
 - At the root of it all: DATA DEPENDENCIES

Characteristics of a Network

- **Topology** (how things are connected)
 - Crossbar, ring, 2-D and 3-D torus, hypercube, omega network
- **Routing algorithm:**
 - Example: all east-west then all north-south (avoids deadlock)
- **Switching strategy:**
 - Circuit switching: full path reserved for entire message, like the telephone
 - Packet switching: message broken into separately-routed packets
- **Flow control** (what if there is congestion):
 - Stall, store data temporarily in buffers, re-route data to other nodes, tell source node to temporarily halt, discard, etc.

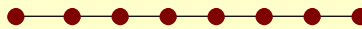
Properties of a Network

- **Diameter**: the maximum (over all pairs of nodes) of the shortest path between a given pair of nodes
- A network is **partitioned** into two or more disjoint sub-graphs if some nodes cannot reach others
- The **bandwidth** of a link = $w * 1/t$
 - w is the number of wires
 - t is the time per bit
- **Effective bandwidth** is usually lower due to packet overhead
- **Bisection bandwidth**: sum of the bandwidths of the minimum number of channels which, if removed, would partition the network into two sub-graphs

Linear and Ring Topologies

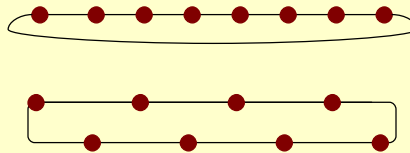
(From A. Krishnamurthy, 2000)

- **Linear array**



- Diameter = $n-1$; average distance $\sim n/3$
- Bisection bandwidth = 1

- **Torus or Ring**

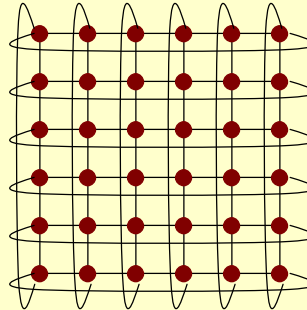
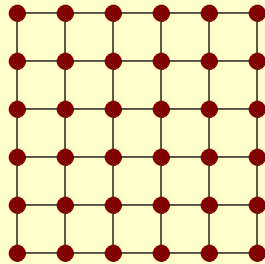


- Diameter = $n/2$; average distance $\sim n/4$
- Bisection bandwidth = 2
- Natural for algorithms that work with 1D arrays

Meshes

(From A. Krishnamurthy, 2000)

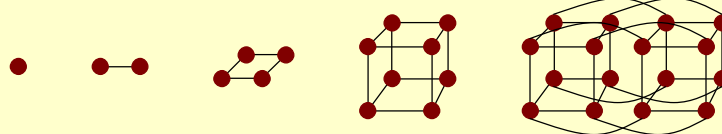
- Diameter = $2 * \sqrt{n}$
- Bisection bandwidth = \sqrt{n}
- Often used as networks in machines (Cray T3D used 3D Torus)



Hypercubes or n -Cube

(From A. Krishnamurthy, 2000)

- Number of nodes $n = 2^d$ for dimension d
 - Diameter = d
 - Bisection bandwidth = $n/2$

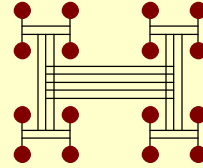
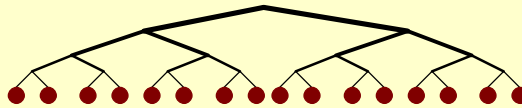
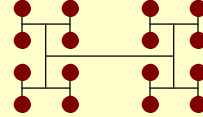


- Popular in early machines (Intel iPSC, NCUBE)
 - Lots of clever algorithms
- Grey-code addressing:
 - Each node connected to d others with 1 bit different

Trees

(From A. Krishnamurthy, 2000)

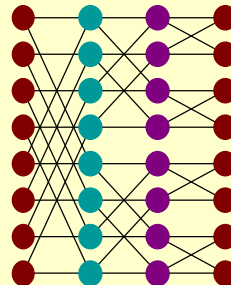
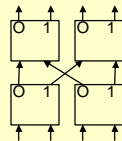
- Diameter = $\log_2 n$
- Bisection bandwidth = 1
- Easy layout as *planar graph*
- Many tree algorithms (e.g., summation)
- Fat trees avoid bisection bandwidth problem:
 - More (or wider) links near top
 - Example: Thinking Machines CM-5



Butterflies

(From A. Krishnamurthy, 2000)

- Diameter = $\log_2 n$
- Bisection bandwidth = 1
- Cost: lots of wires
- Used in BBN Butterfly
- Natural for FFT

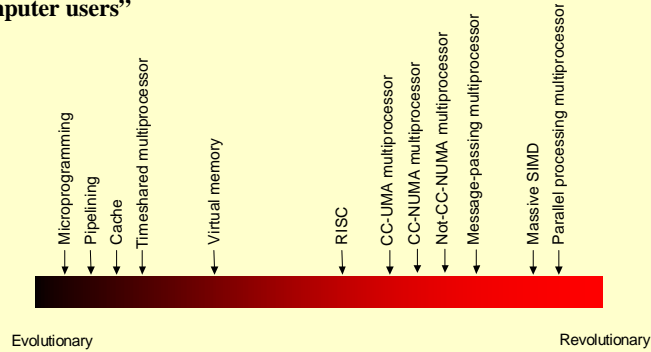


Concluding Remarks

(See Figure 9.18)

- **Evolution vs. Revolution**

“More often the expense of innovation comes from being too disruptive to computer users”



“Acceptance of hardware ideas requires acceptance by software people; therefore hardware people should learn about software. And if software people want good machines, they must learn more about hardware to be able to communicate with and thereby influence hardware engineers.”