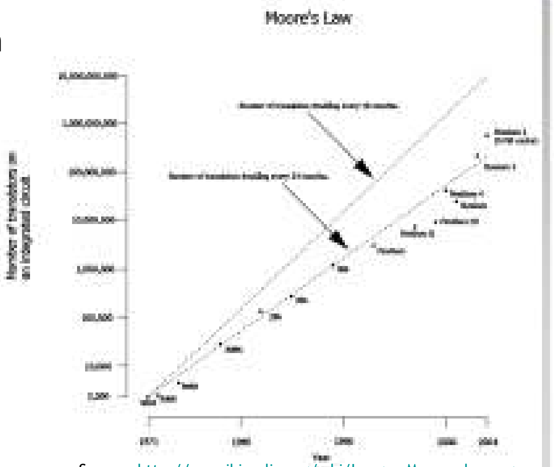
COSC 6385 Computer Architecture - Thread Level Parallelism (I)

Edgar Gabriel

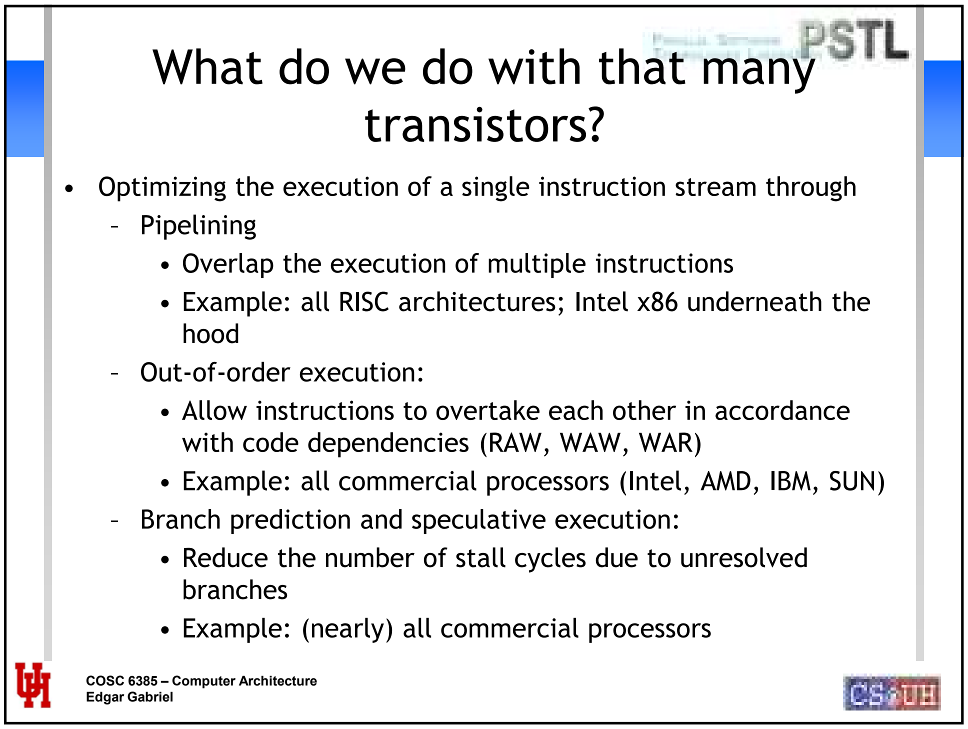
Spring 2013

Moore's Law

- Long-term trend on the number of transistor per integrated circuit
- Number of transistors double every ~18 month

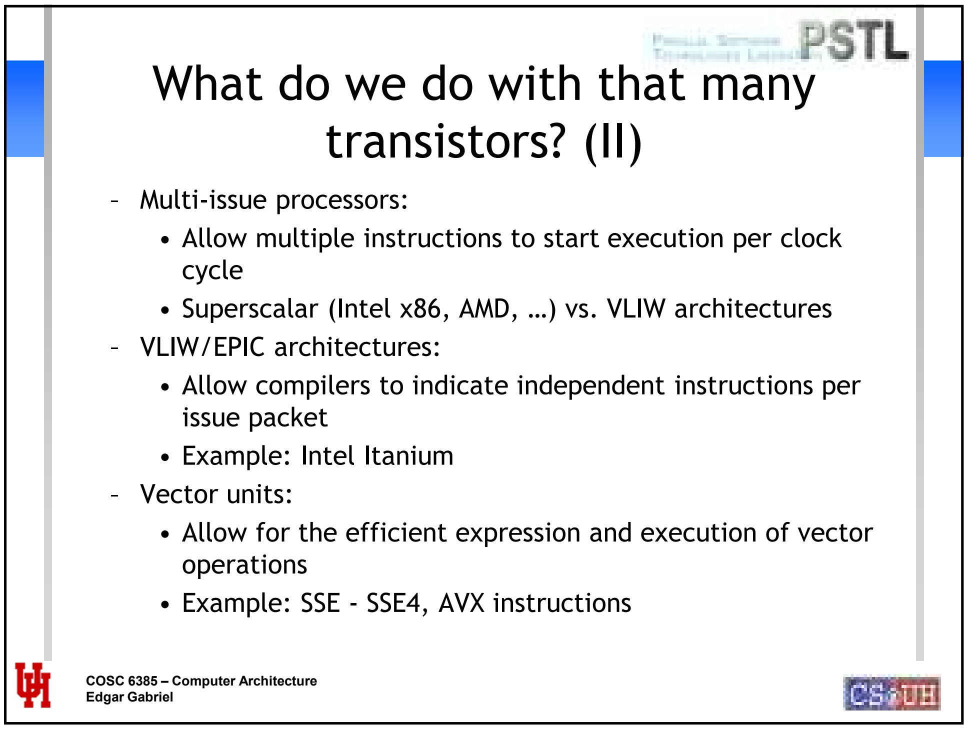


Source: http://en.wikipedia.org/wki/Images:Moores_law.svg



What do we do with that many transistors?

- Optimizing the execution of a single instruction stream through
  - Pipelining
    - Overlap the execution of multiple instructions
    - Example: all RISC architectures; Intel x86 underneath the hood
  - Out-of-order execution:
    - Allow instructions to overtake each other in accordance with code dependencies (RAW, WAW, WAR)
    - Example: all commercial processors (Intel, AMD, IBM, SUN)
  - Branch prediction and speculative execution:
    - Reduce the number of stall cycles due to unresolved branches
    - Example: (nearly) all commercial processors



What do we do with that many transistors? (II)

- Multi-issue processors:
  - Allow multiple instructions to start execution per clock cycle
  - Superscalar (Intel x86, AMD, …) vs. VLIW architectures
- VLIW/EPIC architectures:
  - Allow compilers to indicate independent instructions per issue packet
  - Example: Intel Itanium
- Vector units:
  - Allow for the efficient expression and execution of vector operations
  - Example: SSE - SSE4, AVX instructions

Limitations of optimizing a single instruction stream (II)

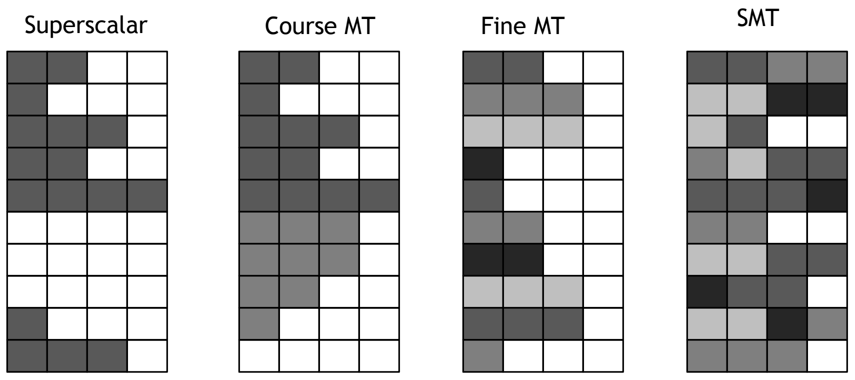
- Problem: within a single instruction stream we do not find enough independent instructions to execute simultaneously due to
  - data dependencies
  - limitations of speculative execution across multiple branches
  - difficulties to detect memory dependencies among instruction (alias analysis)
- Consequence: significant number of functional units are idling at any given time
- Question: Can we maybe execute instructions from another instructions stream
  - Another thread?
  - Another process?

Thread-level parallelism

- Problems for executing instructions from multiple threads at the same time
  - The instructions in each thread might use the same register names
  - Each thread has its own program counter
- Virtual memory management allows for the execution of multiple threads and sharing of the main memory
- When to switch between different threads:
  - Fine grain multithreading: switches between every instruction
  - Course grain multithreading: switches only on costly stalls (e.g. level 2 cache misses)

Simultaneous Multi-Threading (SMT)

- Convert Thread-level parallelism to instruction-level parallelism



Simultaneous multi-threading (II)

- Dynamically scheduled processors already have most hardware mechanisms in place to support SMT (e.g. register renaming)
- Required additional hardware:
  - Registerfile per thread
  - Program counter per thread
- Operating system view:
  - If a CPU supports *n* simultaneous threads, the Operating System views them as *n* processors
  - OS distributes most time consuming threads ‘fairly’ across the *n* processors that it sees.

Example for SMT architectures (I)

- Intel Hyperthreading:
  - First released for Intel Xeon processor family in 2002
  - Supports two architectural sets per CPU,
  - Each architectural set has its own
    - General purpose registers
    - Control registers
    - Interrupt control registers
    - Machine state registers
  - Adds less than 5% to the relative chip size

    Reference: D.T. Marr et. al. “Hyper-Threading Technology Architecture and Microarchitecture”, Intel Technology Journal, 6(1), 2002, pp.4-15. ftp://download.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf

Example for SMT architectures (II)

- IBM Power 5
  - Same pipeline as IBM Power 4 processor but with SMT support
  - Further improvements:
    - Increase associativity of the L1 instruction cache
    - Increase the size of the L2 and L3 caches
    - Add separate instruction prefetch and buffering units for each SMT
    - Increase the size of issue queues
    - Increase the number of virtual registers used internally by the processor.

Simultaneous Multi-Threading

- Works well if
  - Number of compute intensive threads does not exceed the number of threads supported in SMT
  - Threads have highly different characteristics (e.g. one thread doing mostly integer operations, another mainly doing floating point operations)
- Does not work well if
  - Threads try to utilize the same function units
  - Assignment problems:
    - e.g. a dual processor system, each processor supporting 2 threads simultaneously (OS thinks there are 4 processors)
    - 2 compute intensive application processes might end up on the same processor instead of different processors (OS does not see the difference between SMT and real processors!)

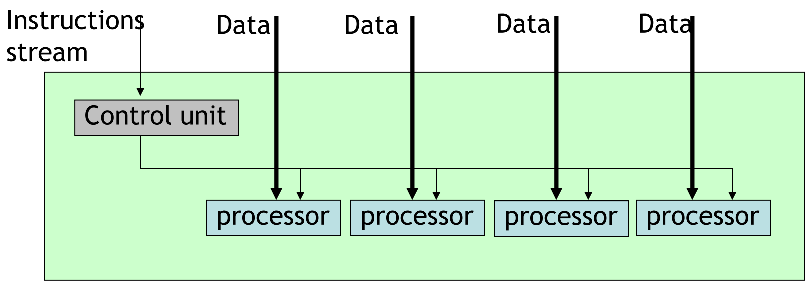
Classification of Parallel Architectures

Flynn's Taxonomy

- SISD: Single instruction single data
  - Classical von Neumann architecture
- SIMD: Single instruction multiple data
- MISD: Multiple instructions single data
  - Non existent, just listed for completeness
- MIMD: Multiple instructions multiple data
  - Most common and general parallel machine

Single Instruction Multiple Data

- Also known as Array-processors
- A single instruction stream is broadcasted to multiple processors, each having its own data stream
  - Still used in some graphics cards today



Multiple Instructions Multiple Data (I)

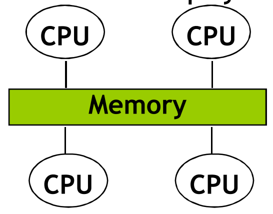
- Each processor has its own instruction stream and input data
- Very general case
  - every other scenario can be mapped to MIMD
- Further breakdown of MIMD usually based on the memory organization
  - Shared memory systems
  - Distributed memory systems

Shared memory systems (I)

- All processes have access to the same address space
  - E.g. PC with more than one processor
- Data exchange between processes by writing/reading shared variables
  - Shared memory systems are easy to program
  - Current standard in scientific programming: OpenMP
- Two versions of shared memory systems available today
  - Centralized Shared Memory Architectures
  - Distributed Shared Memory architectures

Centralized Shared Memory Architecture

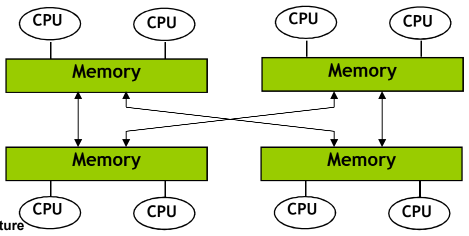
- Also referred to as Symmetric Multi-Processors (SMP)
- All processors share the same physical main memory



- Memory bandwidth per processor is limiting factor for this type of architecture
- Typical size: 2-32 processors

Distributed Shared Memory Architectures

- Also referred to as Non-Uniform Memory Architectures (NUMA)
- Some memory is closer to a certain processor than other memory
  - The whole memory is still addressable from all processors
  - Depending on what data item a processor retrieves, the access time might vary strongly

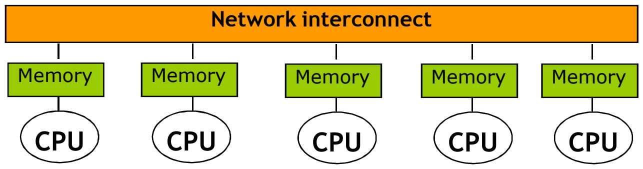


NUMA architectures (II)

- Reduces the memory bottleneck compared to SMPs
- More difficult to program efficiently
  - E.g. first touch policy: data item will be located in the memory of the processor which uses a data item first
- To reduce effects of non-uniform memory access, caches are often used
  - ccNUMA: cache-coherent non-uniform memory access architectures
- Largest example as of today: SGI Origin with 512 processors

Distributed memory machines (I)

- Each processor has its own address space
- Communication between processes by explicit data exchange
  - Sockets
  - Message passing
  - Remote procedure call / remote method invocation



Performance Metrics (I)

- **Speedup:** how much faster does a problem run on *p* processors compared to 1 processor?

$$S(p) = \frac{T_{total}(1)}{T_{total}(p)}$$

  - Optimal: *S(p)* = *p* (linear speedup)
- **Parallel Efficiency:** Speedup normalized by the number of processors

$$E(p) = \frac{S(p)}{p}$$

  - Optimal: *E(p)* = *1.0*

Amdahl's Law (I)

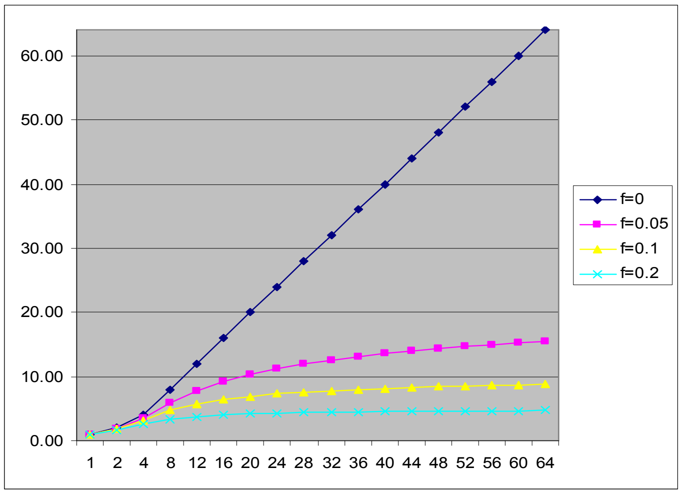
- Most applications have a (small) sequential fraction, which limits the speedup

$$T_{total} = T_{sequential} + T_{parallel} = fT_{Total} + (1-f)T_{Total}$$

f: fraction of the code which can only be executed sequentially

$$S(p) = \frac{T_{total}(1)}{(f + \frac{1-f}{p})T_{total}(1)} = \frac{1}{f + \frac{1-f}{p}}$$

Example for Amdahl's Law

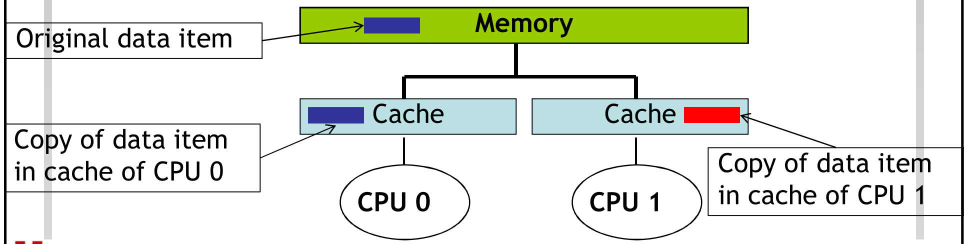


Amdahl's Law (II)

- Amdahl's Law assumes, that the problem size is constant
- In most applications, the sequential part is independent of the problem size, while the part which can be executed in parallel is not.

Cache Coherence

- Real-world shared memory systems have caches between memory and CPU
- Copies of a single data item can exist in multiple caches
- Modification of a shared data item by one CPU leads to outdated copies in the cache of another CPU

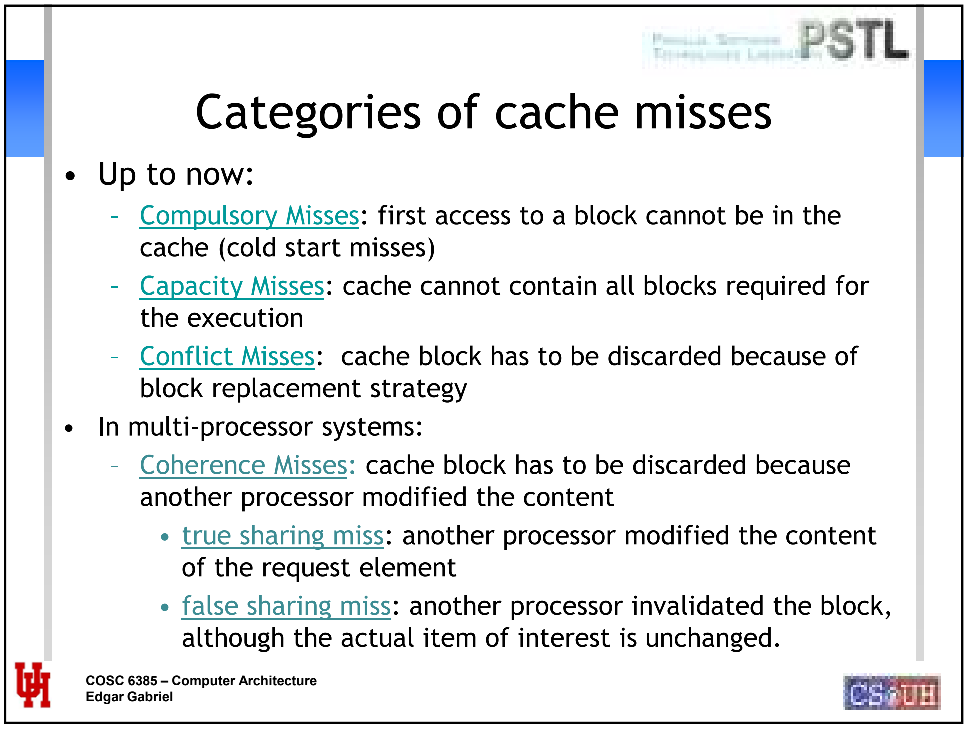

Original data item
Memory
Cache
Cache
Copy of data item in cache of CPU 0
CPU 0
CPU 1
Copy of data item in cache of CPU 1

Cache coherence (II)

- Typical solution:
  - Caches keep track on whether a data item is shared between multiple processes
  - Upon modification of a shared data item, ‘notification’ of other caches has to occur
  - Other caches will have to reload the shared data item on the next access into their cache
- Cache coherence only an issue in case multiple tasks access the same item
  - Multiple threads
  - Multiple processes have a joint shared memory segment
  - Process is being migrated from one CPU to another

Cache Coherence Protocols

- Snooping Protocols
  - Send all requests for data to all processors
  - Processors snoop a bus to see if they have a copy and respond accordingly
  - Requires broadcast, since caching information is at processors
  - Works well with bus (natural broadcast medium)
  - Dominates for centralized shared memory machines
- Directory-Based Protocols
  - Keep track of what is being shared in centralized location
  - Distributed memory => distributed directory for scalability (avoids bottlenecks)
  - Send point-to-point requests to processors via network
  - Scales better than Snooping
  - Commonly used for distributed shared memory machines



Categories of cache misses

- Up to now:
  - Compulsory Misses: first access to a block cannot be in the cache (cold start misses)
  - Capacity Misses: cache cannot contain all blocks required for the execution
  - Conflict Misses: cache block has to be discarded because of block replacement strategy
- In multi-processor systems:
  - Coherence Misses: cache block has to be discarded because another processor modified the content
    - true sharing miss: another processor modified the content of the request element
    - false sharing miss: another processor invalidated the block, although the actual item of interest is unchanged.

COSC 6385 – Computer Architecture
Edgar Gabriel