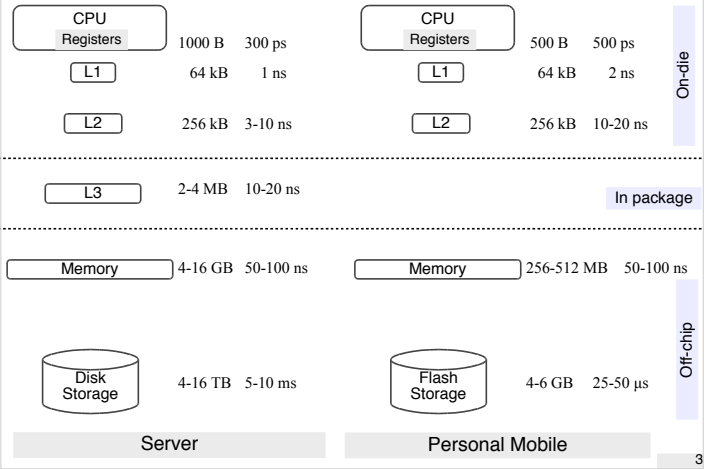
Chapter 2

Memory Hierarchy Design

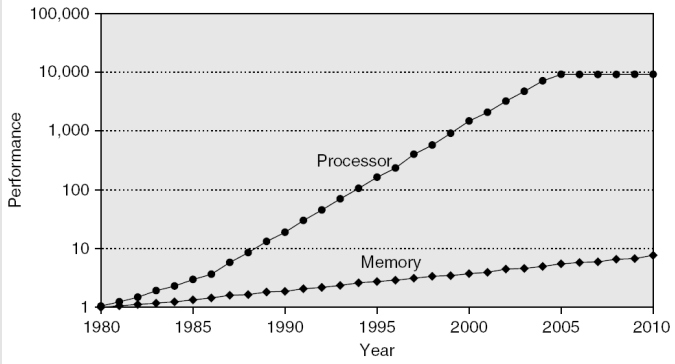
Introduction

- Goal: unlimited amount of memory with low latency
- Fast memory technology is more expensive per bit than slower memory
  - Use principle of locality (spatial and temporal)
- Solution: organize memory system into a hierarchy
  - Entire addressable memory space available in largest, slowest memory
  - Incrementally smaller and faster memories, each containing a subset of the memory below it, proceed in steps up toward the processor
- Temporal and spatial locality insures that nearly all references can be found in smaller memories
  - Gives the illusion of a large, fast memory being presented to the processor

Memory Hierarchies



CPU vs. Memory: Performance vs Latency



Memory Hierarchy Design Considerations

- Memory hierarchy design becomes more crucial with recent multi-core processors:
  - Aggregate peak bandwidth grows with # cores:
    - Intel Core i7 can generate two references per core per clock
    - Four cores and 3.2 GHz clock
      - 25.6 billion 64-bit data references/second +
      - 12.8 billion 128-bit instruction references
      - = 409.6 GB/s!
    - DRAM bandwidth is only 6% of this (25 GB/s)
    - Requires:
      - Multi-port, pipelined caches
      - Two levels of cache per core
      - Shared third-level cache on chip

Performance and Power for Caches

- High-end microprocessors have >10 MB on-chip cache
  - Consumes large amount of area and power budget
  - Both static (idle) and dynamic power is an issue
- Personal/mobile devices have
  - 20-50x smaller power budget
  - 25%-50% is consumed by memory

Handling of Cache Misses

- When a word is not found in the cache, a *miss* occurs:
  - Fetch word from lower level in hierarchy, requiring a higher latency reference
  - Lower level may be another cache or the main memory
  - Also fetch the other words contained within the block
    - Takes advantage of spatial locality
  - Place block into cache in any location within its set, determined by address
    - block address MODULO number of sets

Cache Associativity and Writing Strategies

- n sets → n-way set associative
  - Direct-mapped cache → one block per set
  - Fully associative → one set
- Writing to cache: two strategies
  - Write-through
    - Immediately update lower levels of hierarchy
  - Write-back
    - Only update lower levels of hierarchy when an updated block is replaced
  - Both strategies use write buffer to make writes asynchronous

Miss Rate and Types of Cache Misses

- Miss rate is...
  - Fraction of cache access that result in a miss
- Reasons for cache misses and their names
  - Compulsory
    - Cache block is referenced for the first time
      - Solution: hardware and software prefetchers
  - Capacity
    - Cache block was discarded and later retrieved
      - Solution: build bigger cache or reorganize the code
  - Conflict
    - Program makes repeated references to multiple addresses from different blocks that map to the same location in the cache
      - Solution: add padding or stride to code; change size of data
  - Coherency

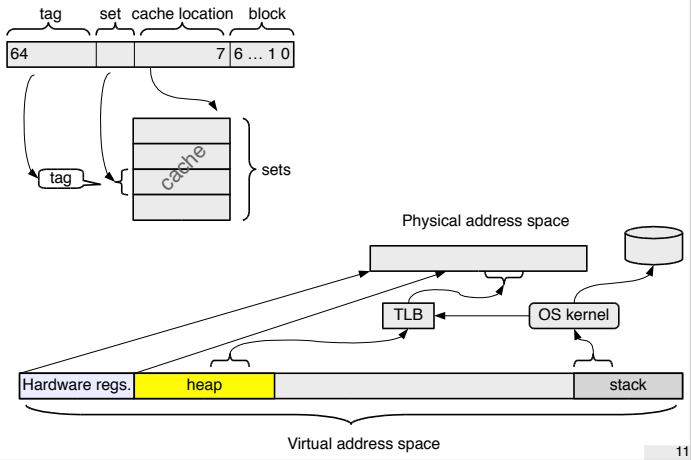
Calculating Miss Rate

$$\frac{\text{Misses}}{\text{Instruction}} = \frac{\text{Miss rate} \times \text{Memory accesses}}{\text{Instruction count}} = \frac{\text{Miss rate} \times \text{Memory accesses}}{\text{Instruction}}$$

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

- Note that speculative and multithreaded processors may execute other instructions during a miss
  - This reduces performance impact of misses
  - But complicates analysis and introduces runtime dependence

Cache and TLB Mapping Illustrated



Cache Design Considerations

- Larger block size
  - Reduced compulsory misses
  - Slightly reduced static power (smaller tag)
  - Sometimes increased capacity and conflict misses
- Bigger cache
  - Reduced miss rate
  - Increased hit time
  - Increased static & dynamic power
- Higher associativity
  - Reduced conflict miss rate
  - Increased hit time
  - Increased power
- More levels of cache
  - Reduced miss penalty

$$\text{Access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

- Prioritize read misses over write misses
  - Reduced miss penalty
  - Need for write buffer and hazard resolution
- Avoid address translation during indexing of cache
  - Reduced hit time
  - Limited size and structure of cache

Categories of Metrics for Cache Optimization

- Reduce hit time
  - Small and simple first-level caches
  - "way-prediction"
  - Side effect: reduction in power consumption
- Increase cache bandwidth
  - Pipelined caches
  - Multibanked caches
  - Nonblocking caches
- Reduce miss penalty
  - "Critical word first"
  - Merging write buffers
- Reduce miss rate
  - Compiler optimization
  - Side effect: reduced power
- Reduce miss penalty and/or rate via parallelism
  - Hardware prefetching
  - Software and compiler prefetching
  - Side effect: increased power due to unused data

Optimization 1: Small/Simple 1st-level Caches

- 1st level cache should match the clock cycle of CPU
- Cache addressing is a three step process
  - Address tag memory with index portion of address
  - Compare the found tag with address tag
  - Choose cache set
- High associativity helps with...
  - Address aliasing
  - Dealing with TLB and multiprocessing conflicts
- Low associativity...
  - Is faster
    - Overlap tag check with data transmission
    - 10% or more for each doubling of set count
  - Consumes less power

Optimization 2: "way prediction"

- Idea: predict "the way"
  - which block within set will be accessed next
- Index multiplexor (mux) starts working early
- Implemented as extra bits kept in cache for each block
- Prediction accuracy (simulated)
  - more effective of instruction caches
  - > 90% for two-way associative
  - > 80% for four-way associative
- On misprediction
  - Try the other block
  - Change the prediction bits
  - Incur penalty (commonly 1 cycle for slow CPUs)
- Examples: 1st use MIPS R10000 in 1990s, ARM Cortex-A8

Optimization 3: Pipelined Cache Access

- Improves bandwidth
- History
  - Pentium (1993) 1 cycle
  - Pentium Pro (1995) 2 cycles
  - Pentium III (1999) 2 cycles
  - Pentium 4 (2000) 4 cycles
  - Intel Core i7 (2010) 4 cycles
- Interaction with branch prediction
  - Increased penalty for mispredicted branches
- Load instructions are longer
  - Waiting for cache pipeline to finish
- Pipeline cache cycles make high degrees of associativity easier to implement

Optimization 4: Nonblocking Caches

- If one instruction stalls on a cache miss, should the following instruction stall if its data is in cache?
  - NO
    - But you have design a nonblocking cache (lockup-free cache)
  - Call it "hit under miss"
- Why stop at two instructions?
  - Make it "hit under multiple miss"
- What about two outstanding misses?
  - "miss under miss"
  - Next level cache has to be able to handle multiple misses
    - Rarely the case in contemporary CPUs
- How long before our caches become
  - Out-of-order, superscalar, …
  - Moving the CPU innovation into the memory hierarchy?

Optimization 5: Multibanked Caches

- Main memory has been organized in banks for increased bandwidth
- Caches can do this too
- Each cache block is evenly spread across banks
  - Sequential interleaving
    - Bank 0: blk[0] Bank 1: blk[1] Bank 2: blk[2] Bank 3: blk[3]
    - Bank 0: blk[4] Bank 1: blk[5] Bank 2: blk[6] Bank 3: blk[7]
- Modern use
  - ARM Cortex-A8
    - 1-4 banks in L2 cache
  - Intel Core i7
    - 4 banks in L1 (2 memory accesses/cycle)
    - 8 banks in L2
- Reduced power usage

Optimization 6: Critical Word 1st ,Early Restart

- Forget cache blocks (lines) deal with words
  - Start executing instruction when its word arrives, not the entire block where the word resides
- Critical word first
  - What if the instruction needs the last word in the cache block?
    - Go ahead and request the last word from memory before requesting others
      - Out-of-order loading of cache block words
    - As soon as the word arrives pass it on to the CPU
    - Continue fetching the remaining words
- Early restart
  - Don't change the order of words, but supply the missing word as soon as it arrives
    - it won't help if the last word in block is needed
- Useful for large cache blocks, depends on data-stream

Optimization 7: Merging Write Buffer (Intro)

- Write buffer basics
  - Write buffer sits between cache and memory
  - Write buffer stores both: data and its address
  - Write buffer allows for the store instruction to finish immediately
    - Unless the write buffer is full
  - Especially useful for write-through caches
    - Write-back caches will benefit for when block is replaced

Optimization 7: Merging Write Buffer

- Merging write buffer: a buffer that merges write requests
- When storing to a block that is already pending in the write buffer, only update the write buffer
- Another way to look at it: the write buffer with merging is equivalent to a larger buffer but without merging
- Merging buffer reduces stalls due to the buffer being full
- Should not be used for I/O addresses (special memory locations)

Addr	valid	Data	valid		valid		valid	
100	1	M[100]	0		0		0	
108	1	M[108]	0		0		0	
116	1	M[116]	0		0		0	
124	1	M[124]	0		0		0	

Addr	valid	Data	valid		valid		valid	
100	1	M[100]	1	M[108]	1	M[116]	1	M[124]

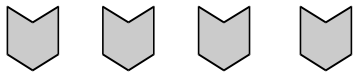
Optimization 8: Compiler Optimizations

- No hardware changes required
- Two main techniques
  - Loop interchange
    - Requires 2 or more loop nests
    - The order of loops is changed to walk through memory in a more cache-friendly manner
  - Loop blocking
    - Additional loop nests are introduced to deal with small portion of an array (called a block but also a tile)

Optimization 8: Loop Interchange

```
/* Before: memory stride = 100 */

for (j = 0; j < 100; ++j)
  for (i = 0; i < 5000; ++i)
    x[i][j] = 2 * x[i][j];
```

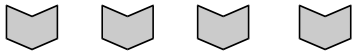


```
/* After: memory stride = 1
 * Uses all words in a single cache block */

for (i = 0; i < 5000; ++i)
  for (j = 0; j < 100; ++j)
    x[i][j] = 2 * x[i][j];
```


Optimization 8: Blocking

```
/* Before: memory stride: A → 1, B → N */
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j) { x = 0;
    for (k = 0; k < N; ++k)  // dot product
      x += A[i][k] * B[k][j];
    C[i][j] = x;
  }
```



```
/* After: blocking factor = B */
for (jj = 0; jj < N; jj += B)
for (kk = 0; kk < N; kk += B)
for (i = 0; i < N; ++i)
  for (j = jj; j < min(jj+B,N); ++j) { x=0;
    for (k = kk; k < min(kk+B,N); ++k)
      x += A[i][k] * B[k][j];
    C[i][j] += x;
}
```

Optimization 9: Hardware Prefetching

- Prefetch asks for data before it's needed
  - Reduction of latency (miss penalty)
- Overlap the fetch from memory with
  - Execution of instructions
- Principal of locality
  - On a miss: request not just one (cache) block but also the next block
- The prefetched data might not end up in cache
  - It might go to a special prefetch buffer that is cheaper to access than memory
- Intel Core i7 can prefetch both L1 and L2
  - Pairwise prefetching (get the current and the next sequential block)
  - May be turned off in BIOS

Optimization 9: Hardware Prefetching contd.

- Hardware prefetching may waste bandwidth
  - When the prediction accuracy is poor and most of the prefetched data goes unused
- Aggressive and sophisticated prefetchers
  - Use extra silicon area
  - Consume power
  - Increase chip complexity
- Compilers may reorganize the code to increase hardware prefetch accuracy

Optimization 10: Compiler Prefetching

- Compiler may insert (special) instructions that fetch data from memory but do not influence the current code
- Two flavors of prefetching instructions
  - Register prefetches
    - Data loaded into a register
  - Cache prefetches
    - Data loaded into cache
- Interaction with the virtual memory:
  - Faulting prefetches cause virtual address faults
  - Nonfaulting prefetches do not cause such faults
    - Because the hardware ignores the prefetch instruction right before the fault
- The most common prefetch is into cache and nonfaulting
  - It is called nonbinding

Optimization 10: Compile Prefetch Example

```
/* Before */
for (i = 0; i < 5000; ++i)
  x[i] = 2 * x[i];

/* After */
for (i = 0; i < 5000; ++i) {
  // get ready for next iteration
  prefetch(x[i+1]); // better use cache block
  x[i] = 2 * x[i];
}


/* Even better */
for (i = 0; i < 5000; ++i) {
  // better prefetch the next cache block
  prefetch(x[i+8]);
  x[i] = 2 * x[i];
}
```


Memory Technology Optimization

- Memory sits between caches and I/O devices
  - Bridges the gap of few tens of cycles cache latency and millions of cycles latency to disk
- Memory performance is measured with
  - Latency
    - Cache designers are concerned with that because this is the main factor in miss penalty
  - Bandwidth
    - Ever faster I/O devices (think solid disk storage) and increasing core counts stress the memory bandwidth
- Faster and larger cache cannot eliminate the need for main memory
  - Even with principle of locality in mind there are codes that wait for memory
  - Use Amdahl's law to see how important is faster memory

Memory Technology Overview

- Memory responsiveness is measured with
  - Access time
    - The time between read request and data arrival
    - This could also be called memory latency
  - Cycle time
    - Minimum time between unrelated memory requests
    - Comes from the way data is stored in a single memory cell
- Two types of memory
  - SRAM
    - Used caches (since 1975)
  - DRAM
    - Used for main memory (since 1975)

Memory Technology: SRAM

- SRAM = Static RAM
- No need for refresh (unlike DRAM)
  - Hence, access time = cycle time
- Typically, 6 transistors to store a single bit
  - Prevents loss of information after read
- Minimum power required to retain the charge in standby mode
- Most SRAM memories (caches) is not integrated into the chip

Memory Technology: DRAM

- DRAM = Dynamic RAM
- Only one transistor used to store a bit
  - After a read, the information is gone
  - So a read must be followed by a write to restore the state!
  - Hence cycle time is longer than access time
  - Using multiple banks helps hiding the rewrite of data
  - Also, over time (8ms) data is lost: periodic refresh required
    - Row refreshed at once (usually part of the mem. controller)
- Number of pins required to address a single item is an issue
  - Address lines are multiplexed
  - Row Access Strobe (RAS) is the first half of the address
  - Column Access Strobe (CAS) - the second half of the address
  - Internally, DRAM is a 2D matrix and each half of the address works in one of the dimensions

Memory Technology: DRAM (contd.)

- Amdahl suggested linear growth of capacity with CPU speed
  - 1000 MB for every 1000 MIPS
  - But Moore's law made that impossible: CPU speed grew faster
  - Capacity grew only about half as fast in recent years
    - The expected rate (from the CPU designer perspective) is 55% a year
      - Or fourfold improvement in capacity every three years
- Memory latency (measured as row access time) improves only 5% per year
- Data transfer time (related to CAS) improves at over 10% a year
- DRAM packaging unit: DIMM
  - Dual Inline Memory Module
  - 8 bytes wide + ECC (Error-Correcting Code)

Improving Memory Performance

- The main pressure comes from the Moore's law
  - Faster single core CPUs need more data faster
  - More cores per chip need more data faster
- Multiple accesses to the same row
  - Utilizes the row buffer that holds the data from the row when the column portion of the address arrives
  - Multiple column addresses can nowadays utilize the row buffer
- Synchronous DRAM
  - Memory controller and the SDRAM use a clock for synchronization (faster than asynchronous operation)
  - Also, SDRAM has burst mode
    - In this mode, SDRAM delivers multiple data items without new address requests
    - Possible due to an extra length register in SDRAM
  - Supports "Critical Word First" cache optimization

Improving Memory Performance (contd.)

- DRAM is getting wider
  - Required due to every increasing density (Get more data out at once)
    - Old DDR: 4-bit bus
    - DDR 2, DDR 3: 16-bit bus (2010)
- DDR = Double Data Rate
  - Single data rate: data transferred on one edge of the clock signal
  - Double data rate: data transferred on both edges of the clock signal
  - Effectively doubles the bandwidth with the same clock
- Multiple banks
  - Helps due to interleaving (one word spread across banks)
  - Smaller power consumption
  - Adds delay because bank has to get opened before access

DDR Standards and Technologies

- DDR is now a standard
  - This helps interoperability, competition and results in lower prices
- DDR (2000)
  - 2.5 V; 133, 150, 200 MHz; >100 nm
- DDR2 (2004)
  - 1.8 V; 266, 333, 400 MHz; 80-90 nm
- DDR3 (2010)
  - 1.5 V; 533, 666, 800 MHz; 32-40 nm
- DDR4 (expected late 2013?)
  - 1-1.2 V; 1066-1600 MHz; 2x nm

GPU Memory

- GPU memory
  - GDRAM Graphics = DRAM
  - GSDRAM = Graphics Synchronous DRAM
- GDDR
  - GDDR5 based on DDR3, earlier GDDRs based DDR2
- GPUs demand more bandwidth because of higher performance due to greater parallelism
- Greater bandwidth achieved with:
  - Wider interface: 32-bits (vs. 4,8,16 for CPUs)
  - Higher clock rate allowed by soldering into GPU board rather snap in sockets for CPU DIMMs
- In practice, GDDR is 2-5x faster than DDR

SDRAM Power Optimizations

- Higher clock rate means greater (static and dynamic) power draw of SDRAM
- Optimizing power becomes important as SDRAMs grow
- Common techniques include
  - Reduce operating voltage (from 1.5 to 1.35)
  - Introduce multiple banks
    - one bank opens at a time to deliver subsequent word
  - Recent SDRAMs enter power-down mode
    - In this mode the memory modules ignore the clock
    - However, need to keep refreshing the clock
      - Usually implemented as an internal refresh circuitry

Flash Memory Introduction

- Flash memory is a type of EEPROM
  - Electronically Erasable Programmable Read-Only Memory
  - Read-only under normal operation
  - Erasable when special voltage applied
    - The memory module is flashed = erased
- Suitable for long term storage
  - Used in mobile form factors: phones, tables, laptops
    - May be used as another level of memory hierarchy due the small size of RAM in these devices

Flash Memory Properties

- Flash memory must be erased before overwritten
  - NAND Flash (higher density Flash) erasing is done by blocks
    - This becomes a software (OS kernel) problem to assemble data in blocks and clean up old blocks
- Flash memory is static
  - No continuous refreshes, almost no power draw when inactive
- Flash has limited number of times (100k) each block can be written
- Cheaper than SDRAM, more expensive than disk
  - Flash: $2 / GB, SDRAM: $30 / GB, Magnetic disk: $0.09 / GB
- Much slower than SDRAM, much faster than disk
  - For reads: 4 times slower than SDRAM, 1000x faster than disk
  - Writes to flash are 2x-20x slower than reads

Memory Dependability

- Errors in memory systems
  - Permanent errors during fabrication
  - Dynamic errors during operation = soft errors
    - Cosmic rays (high energy particles)
- Manufacturing defects are accommodated with spare rows
  - After fabrication memory modules are configured
    - Defective rows are disabled and spare rows replace them
- Dynamic errors are detected with parity bits and corrected with ECC (Error-Correcting Codes)
  - Instruction caches are read-only so the parity bits suffice
  - Parity bit does not detect multi-bit errors: 1 parity + 8 data bits
  - ECC detects 2 and corrects 1 error for 8-bit ECC + 64-bit data

Memory Dependability: Chipkill

- High end servers and warehouse clusters need Chipkill
  - Chipkill is like RAID for Disks
  - Parity bits and ECC are not kept together with data but are distributed
  - Complete failure of a single memory module can be handled
- Developed by IBM, Intel calls it SDDC
- According to IBM analysis, 10,000 processor server with 4GB/CPU has unrecoverable failures in 3 years:
  - 90000 when only parity is used
  - 3500 with ECC only
  - 6 with Chipkill

Virtual Memory Basics

- Virtual memory is
  - Protection mechanism to keep process memory data private
  - Context switch changes the process which the CPU is executing
    - After switch, old process' instructions and data are not visible to the new process
  - Cooperation between hardware (TLB=translation look-aside buffer, segmented memory, segmentation fault mechanism) and software (OS kernel)
  - The same address space for each running process
- Virtual memory system is a security measure
  - It's harder to break the encrypted message than to snoop it from cache right after the context switch
  - Context switch reloads TLBs and registers but not cache
    - Flushing all caches takes too many cycles for each switch
    - This becomes the starting point for side-channel crypto-attacks

Hardware Requirements for Virtual Memory

- Provide user and kernel modes of execution
  - Only the kernel is allowed certain operations
- Make some CPU state read-only
  - Read-only: Am I in kernel mode? vs. Change to kernel mode.
- Provide programmable way of changing modes
  - Entering the kernel occurs usually with a system call and/or a special assembly instruction
  - There is an instruction to return to the previous mode by restoring the register file saved before the system call
- Check memory protection for every memory access
  - Limit access to memory of other processes
  - Execute kernel code if segmentation violation occurs
- TLB is the main component of any Virtual Memory system

Translation Look-aside Buffer (TLB)

- Without TLB, each memory access would be
  - A load to check the translation table
  - The actual memory access (if it doesn't violate the protection)
- TLB operates on the principle of locality
  - If memory access have locality then the address translations must have locality
  - If the address accessed is not in TLB then the translation must involve main memory
- TLB is like cache
  - TLB tags store a portion of the virtual address
  - TLB data is the physical address, protection bit, valid bit, use bit, dirty bit

Virtual Memory Caveats

- Virtual Memory system works if both hardware and software are flawless
  - In practice it's never the case because of bugs
    - Complexity of hardware increases through Moore's law
    - Complexity of OS software (millions of lines of code) increases with hardware complexity and new features
- Finding a security whole is a matter of time and effort
- If complexity is the problem than a simpler system has a better chance of being secure
  - Virtual Machines have smaller “code base” in terms of hardware and software
  - OS kernel no longer has to be trusted
    - In fact, it could be a malicious OS and the VM system will keep it isolated

Rationale Behind Virtual Machines

- Security becomes important issue in modern systems connected to the Internet
- Security failures of standard OS kernels
- Users of modern systems are unrelated and little in common
  - In cloud computing, users might request different OS's
- Performance gains in CPU speed made the VM overhead acceptable
- The first implementations appeared in the 1960s
  Wide acceptance in 2000s

Virtual Machine Types

- Every interpreted language could be considered a VM
  - Java VM, Dalvik (Android)
  - Python (Cpython, Jython, IronPython), Ruby (Rubinus), Perl (Parrot)
  - .Net is a virtualization with the assumption that the byte code will be translated into assembly upon execution and never interpreted
  - x86 code inside Chrome browser can run inside Native Client (NaCl)
  - Emscripten translates C/C++ into obscure Javascript that runs very fast (asm.js project)
- Here, we are interested in VMs at binary ISA and hardware levels
  - We will only consider VMs that export the same ISA as the underlying hardware (no on-the-fly instruction translation)

Virtual Machine Terminology

- Virtual Machine Monitor (or Hypervisor)
  - Software that supports VMs
  - Much smaller than the tradition OS
    - Tens of thousands of lines of code for VMM vs. many millions for the OS
- Host
  - Underlying hardware platform
- Guest (VM)
  - Software (OS with applications) that run on the VM

Virtual Machine Overheads and Benefits

- VM comes with overhead
  - CPU-bound codes experience very little slow-down
  - I/O-intensive workloads end up calling the OS kernel frequently
    - This results in system calls and privileged instruction execution
    - Virtualization later has to emulate or protect these code sections
    - Overhead depends on the speed of emulation/protection and is often noticeable (cost I/O instruction surges compared with the rest of instructions)
  - I/O-bound applications spend most of the time in I/O
    - Overhead is large by small compared to speed of disks
- Benefits of VMs
  - Software management: multiple OS's and versions simultaneously
  - Hardware management: multiplexing of server resources, migration on failure, load balancing overloaded servers

Requirements for Virtual Machines

- Guest software should behave on a VM as it behaves on the hardware (bare metal)
  - Except for overhead and limitation on resources
- Guest software should not be able to change resource allocation directly
  - Security and isolation
- Hardware has to provide at least to modes: system (for the host) and user (for the guest Vms)
  - This is similar to virtual memory system privileges
- Some instructions should only be available in system mode
  - This is similar to virtual memory system calls and TLB instructions

Impact of Virtual Machines on Virtual Memory

- Every guest OS maintains its own set of pages
  - Virtualization introduces real memory
  - Guest OS maps virtual address to real memory addresses
  - VMM maps real memory addresses to physical address
  - The mapping is done via shadow page table
    - No actual table exists, no additional level of indirection
    - Instead, changes to guest OS table occur through special instructions that trap and transfer control to VMM
  - TLBs in some RISC processors include Process ID
    - This eliminates the need for TLB flush upon guest VM switch

Impact of Virtual Machines on Virtual Memory

- I/O subsystem has to be virtualized to allow sharing between VM guests
- Diversity of I/O devices poses a challenge
  - Supporting many device drivers introduces complexity into VMM
  - Generic drivers are often used to provide a common interface
- Physical disks are partitioned by the VMM and each partition becomes a disk visible to a guest VM
- Network interface are share in time slices
  - VMM keeps track of virtual network addresses and delivers packets to the corresponding guest VM