The Memory Hierarchy Caches, and VM CS 740

10/2/2013

Topics

- The memory hierarchy
- Cache design
- Virtual Memory

Ideal Memory

Zero access time (latency)

Infinite capacity

Zero cost

Infinite bandwidth (to support multiple accesses in parallel)

The Problem

Ideal memory's requirements oppose each other

Bigger is slower

 \cdot Bigger \rightarrow Takes longer to determine the location

Faster is more expensive

Memory technology: SRAM vs. DRAM

Higher bandwidth is more expensive

 Need more banks, more ports, higher frequency, or faster technology

Computer System



- 4 -

CS 740

The Tradeoff



Why is bigger slower?

- Physics slows us down
- Racing the speed of light: (3.0x10⁸m/s)
 - clock = 500MHz
 - how far can I go in a clock cycle?
 - (3.0x10^8 m/s) / (500x10^6 cycles/s) = 0.6m/cycle
 - For comparison: 21264 is about 17mm across

• Capacitance:

- long wires have more capacitance
- either more powerful (bigger) transistors required, or slower
- signal propagation speed proportional to capacitance
- going "off chip" has an order of magnitude more capacitance

Alpha 21164 Chip Photo

Microprocessor Report 9/12/94

Caches:

- L1 data
- L1 instruction
- L2 unified
- + L3 off-chip



Alpha 21164 Chip Caches



Memory in a Modern System



Locality of Reference

Principle of Locality:

- Programs tend to reuse data and instructions near those they have used recently.
- <u>Temporal locality:</u> recently referenced items are likely to be referenced in the near future.
- <u>Spatial locality:</u> items with nearby addresses tend to be referenced close together in time.

Locality in Example:

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
*v = sum;
```

- Data
 - Reference array elements in succession (spatial)
- Instructions
 - Reference instructions in sequence (spatial)
 - Cycle through loop repeatedly (temporal)

- 10 -

Caching: The Basic Idea

Main Memory

Stores words

A-Z in example

Cache

Stores subset of the words

4 in example

- Organized in lines
 - Multiple words
 - To exploit spatial locality

Access

 Word must be in cache for processor to access

= - 11



How important are caches?



- •21264 Floorplan
- •Register files in middle of execution units
- ·64k instr cache
- •64k data cache
- •Caches take up a large fraction of the die

(Figure from Jim Keller, Compaq Corp.)

Accessing Data in Memory Hierarchy

- Between any two levels, memory is divided into *lines* (aka "*blocks*")
- Data moves between levels on demand, in line-sized chunks
- Invisible to application programmer
 - Hardware responsible for cache operation
- Upper-level lines a subset of lower-level lines





Design Issues for Caches

Key Questions:

- \cdot Where should a line be placed in the cache? (line placement)
- \cdot How is a line found in the cache? (line identification)
- Which line should be replaced on a miss? (line replacement)
- What happens on a write? (write strategy)

Constraints:

- Design must be very simple
 - Hardware realization
 - All decision making within nanosecond time scale
- \cdot Want to optimize performance for "typical" programs
 - Do extensive benchmarking and simulations
 - Many subtle engineering tradeoffs

Direct-Mapped Caches

Simplest Design

• Each memory line has a unique cache location

Parameters

- Line (aka block) size B = 2^b
 - Number of bytes in each line
 - Typically 2X-8X word size
- Number of Sets S = 2^s
 - Number of lines cache can hold
- Total Cache Size = B*S = 2^{b+s}

Physical Address

- Address used to reference main memory
- *n* bits to reference $N = 2^n$ total bytes
- Partition into fields
 - Offset: Lower b bits indicate which byte within line
 - Set: Next s bits indicate how to locate line within cache
 - Tag: Identifies this line when in cache

n-bit Physical Address

S

set index

b

offset

t

tag

Indexing into Direct-Mapped Cache



Direct-Mapped Cache Tag Matching

Identifying Line



Properties of Direct Mapped Caches

Strength

- Minimal control hardware overhead
- Simple design
- \cdot (Relatively) easy to make fast

Weakness

- Vulnerable to thrashing
- \cdot Two heavily used lines have same cache index
- Repeatedly evict one to make room for other



Vector Product Example

```
float dot_prod(float x[1024], y[1024])
{
   float sum = 0.0;
   int i;
   for (i = 0; i < 1024; i++)
      sum += x[i]*y[i];
   return sum;
}</pre>
```

Machine

- DECStation 5000
- MIPS Processor with 64KB direct-mapped cache, 16 B line size

Performance

- Good case: 24 cycles / element
- Bad case: 66 cycles / element



Thrashing Example: Good Case



Access Sequence

- Read ×[0]
 - x[0], x[1], x[2], x[3] loaded
- Read y[0]
 - y[0], y[1], y[2], y[3] loaded
- Read x[1]
 - Hit
- Read y[1]
 - Hit
- • •
- 2 misses / 8 reads

- 22 - =

Analysis

- x[i] and y[i] map to different cache lines
- Miss rate = 25%
 - Two memory accesses / iteration
 - On every 4th iteration have two misses

Timing

- 10 cycle loop time
- 28 cycles / cache miss
- Average time / iteration =
 10 + 0.25 * 2 * 28

Thrashing Example: Bad Case





- Read ×[0]
 - x[0], x[1], x[2], x[3] loaded
- Read y[0]
 - y[0], y[1], y[2], y[3] loaded
- Read x[1]
 - x[0], x[1], x[2], x[3] loaded
- Read y[1]
 y[0], y[1], y[2], y[3] loaded
- • •
- 8 misses / 8 reads

Analysis

Cache

Line

- · x[i] and y[i] map to same cache lines
- Miss rate = 100%
 - Two memory accesses / iteration
 - On every iteration have two misses

Timing

- 10 cycle loop time
- 28 cycles / cache miss
- Average time / iteration =
 10 + 1.0 * 2 * 28

Miss Types	
Compulsory Misses – required to warm up the cache	
Capacity Misses –	occur when the cache is full
Conflict Misses –	Block placement may cause these in direct or non-fully associative caches
Coherence Misses -	occur because of invalidations caused between threads
_ 24 -	CS 740

Set Associative Cache

Mapping of Memory Lines

- Each set can hold E lines (usually E=2-8)
- $\boldsymbol{\cdot}$ Given memory line can map to any entry within its given set

Eviction Policy

- $\boldsymbol{\cdot}$ Which line gets kicked out when bring new line in
- Commonly either "Least Recently Used" (LRU) or pseudo-random
 - LRU: least-recently accessed (read or written) line gets evicted





Associative Cache Tag Matching

Identifying Line





- Set index selects a set from the cache
- The two tags in the set are compared in parallel
- Data is selected based on the tag result



Fully Associative Cache

Mapping of Memory Lines

- Cache consists of single set holding E lines
- Given memory line can map to any line in set
- Only practical for small caches



Entire Cache

29

Fully Associative Cache Tag Matching



Replacement Algorithms

• When a block is fetched, which block in the target set should be replaced?

Optimal algorithm:

- replace the block that will not be used for the longest period of time
- must know the future

Usage based algorithms:

- Least recently used (LRU)
 - replace the block that has been referenced least recently
 - hard to implement (unless low associativity)
- Not most recently used
- Victim/next-victim

Non-usage based algorithms:

- First-in First-out (FIFO)
 - treat the set as a circular queue, replace block at head of queue.
 - Essentially replace oldest
- Random (RAND)
 - replace a random block in the set
 - even easier to implement

CS 740

Implementing RAND and FIFO

FIFO:

- maintain a modulo E counter for each set.
- counter in each set points to next block for replacement.
- increment counter with each replacement.

RAND:

- maintain a single modulo E counter.
- counter points to next block for replacement in any set.
- increment counter according to some schedule:
 - each clock cycle,
 - each memory reference, or
 - each replacement anywhere in the cache.

LRU

- Need state machine for each set
- $\boldsymbol{\cdot}$ Encodes usage ordering of each element in set
- E! possibilities ==> ~ E log E bits of state

Write Policy

- What happens when processor writes to the cache?
- Should memory be updated as well?

Write Through:

- Store by processor updates cache *and* memory
- Memory always consistent with cache
- Never need to store from cache to memory
- ~2X more loads than stores



Write Policy (Cont.)

Write Back:

- Store by processor only updates cache line
- $\boldsymbol{\cdot}$ Modified line written to memory only when it is evicted
 - Requires "dirty bit" for each line
 - » Set when line in cache is modified
 - » Indicates that line in memory is stale
- Memory not always consistent with cache



Write Buffering

Write Buffer

- Common optimization for all caches
- Overlaps memory updates with processor execution
- \cdot Read operation must check write buffer for matching address





How does this affect self modifying code?
Bandwidth Matching

Challenge

- CPU works with short cycle times
- DRAM (relatively) long cycle times
- Short • How can we provide enough bandwidth between processor Latency & memory?

Effect of Caching

- Caching greatly reduces amount of traffic to main memory
- But, sometimes need to move large amounts of data from memory into cache

Trends

- Need for high bandwidth much greater for multimedia applications
 - Repeated operations on image data
- Recent generation machines (e.g., Pentium II) greatly improve on predecessors



CPU



Cache Performance Metrics

Miss Rate

- fraction of memory references not found in cache (misses/references)
- Typical numbers:

3-10% for L1

can be quite small (e.g., < 1%) for L2, depending on size, etc.

Hit Time

- time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)
- Typical numbers:
 - 1-3 clock cycles for L1
 - 3-12 clock cycles for L2

Miss Penalty

- $\boldsymbol{\cdot}$ additional time required because of a miss
 - Typically 25-100 cycles for main memory

Hierarchical Latency Analysis

For a given memory hierarchy level i it has a technology-intrinsic access time of t_i. The perceived access time T_i is longer than t_i

Except for the outer-most hierarchy, when looking for a given address there is

- a chance (hit-rate h_i) you "hit" and access time is t_i
- a chance (miss-rate m_i) you "miss" and access time t_i +T_{i+1}
- $h_i + m_i = 1$

Thus

 $T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})$ $T_i = t_i + m_i \cdot T_{i+1}$

keep in mind, h_i and m_i are defined to be the hit-rate and miss-rate of just the references that missed at L_{i-1}

Hierarchy Design Considerations

Recursive latency equation

 $\mathbf{T}_{i} = \mathbf{t}_{i} + \mathbf{m}_{i} \cdot \mathbf{T}_{i+1}$

The goal: achieve desired T₁ within allowed cost

 $T_i \approx t_i$ is desirable

Keep m_i low

- increasing capacity C_i lowers m_i, but beware of increasing t_i
- lower m_i by smarter management (replacement::anticipate what you don't need, prefetching::anticipate what you will need)

Keep T_{i+1} low

- faster lower hierarchies, but beware of increasing cost
- introduce intermediate hierarchies as a compromise

Intel Pentium 4 Example

90nm P4, 3.6 GHz

L1 D-cache

- C₁ = 16K
- t₁ = 4 cyc int / 9 cycle fp

L2 D-cache

- C₂ =1024 KB
- t₂ = 18 cyc int / 18 cyc fp

Main memory

• t₃ = ~ 50ns or 180 cyc

Notice

- best case latency is not 1
- worst case access latencies are into 500+ cycles

- if m₁=0.1, m₂=0.1 T₁=7.6, T₂=36
- if m_1 =0.01, m_2 =0.01 T₁=4.2, T₂=19.8
 - if $m_1 = 0.05$, $m_2 = 0.01$ $T_1 = 5.00$, $T_2 = 19.8$

if
$$m_1 = 0.01$$
, $m_2 = 0.50$
 $T_1 = 5.08$, $T_2 = 108$

Impact of Cache and Block Size

Cache Size

- Effect on miss rate?
- Effect on hit time?

Block Size

- Effect on miss rate?
- Effect on miss penalty?
- Effect on hit time?

CS 740

Impact of Associativity

• Direct-mapped, set associative, or fully associative? Total Cache Size (tags+data)?

Miss rate?

Hit time?

Miss Penalty?

Impact of Replacement Strategy

= CS 740

RAND, FIFO, or LRU?
 Total Cache Size (tags+data)?

Miss Rate?

Miss Penalty?

Impact of Write Strategy

• Write-through or write-back? Advantages of Write Through?

Advantages of Write Back?

Allocation Strategies

• On a write miss, is the block loaded from memory into the cache?

Write Allocate:

- Block is loaded into cache on a write miss.
- Usually used with write back
- Otherwise, write-back requires read-modify-write to replace word within block

			read			modify			write					
write buffer block		17			17				17				17	
temporary buffer			5	7	11	13	5	7	17	13	5	7	17	13
memory block	5 7	11 13	5	7	11	13	5	7	11	13	5	7	17	13
• But if it in c	you've <u>c</u> ache?	gone to th	e tro	bubl	e of	[;] readi	ing t	he e	entir	re blo	ck, v	vhy	not l	oad

Allocation Strategies (Cont.)

• On a write miss, is the block loaded from memory into the cache?

No-Write Allocate (Write Around):

- $\boldsymbol{\cdot}$ Block is not loaded into cache on a write miss
- Usually used with write through
 - Memory system directly handles word-level writes

Qualitative Cache Performance Model

Miss Types

- Compulsory ("Cold Start") Misses
 - First access to line not in cache
- Capacity Misses
 - Active portion of memory exceeds cache size
- Conflict Misses
 - Active portion of address space fits in cache, but too many lines map to same cache entry
 - Direct mapped and set associative placement only
- · Coherence Misses
 - Block invalidated by multiprocessor cache coherence mechanism

Hit Types

- \cdot Reuse hit
 - Accessing same word that previously accessed
- \cdot Line hit
 - Accessing word spatially near previously accessed word

Interactions Between Program & Cache

/* ijk */

Major Cache Effects to Consider

- Total cache size
 - Try to keep heavily used data in highest level cache
- Block size (sometimes referred to "line size")
 - Exploit spatial locality

Example Application

- Multiply n X n matrices
- O(n³) total operations
- Accesses
 - n reads per source element
 - n values summed per destination
 - » But may be able to hold in register

_ _

Variable sum

held in register

Matmult Performance (Alpha 21164)



Block Matrix Multiplication

Example n=8, B = 4:

A11 A12		B11 B12		C11 C12
A21 A22	Х	B21 B22	=	C21 C22

Key idea: Sub-blocks (i.e., A_{ii}) can be treated just like scalars.

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

- 52

Blocked Matrix Multiply (bijk)

```
for (jj=0; jj<n; jj+=bsize) {</pre>
 for (i=0; i<n; i++)</pre>
   for (j=jj; j < min(jj+bsize,n); j++)</pre>
     c[i][j] = 0.0;
-for (kk=0; kk<n; kk+=bsize) {</pre>
   for (i=0; i<n; i++) {</pre>
     for (j=jj; j < min(jj+bsize,n); j++) {</pre>
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {</pre>
          sum += a[i][k] * b[k][j];
        c[i][j] += sum;
```

53 -

Blocked Matrix Multiply Analysis

- Innermost loop pair multiplies 1 X bsize sliver of A times bsize X bsize block of B and accumulates into 1 X bsize sliver of C
- Loop over i steps through n row slivers of A & C, using same B



Blocked matmult perf (Alpha 21164)



Why VM?: 1) DRAM a "Cache" for Disk

The full address space is quite large:

- 32-bit addresses: ~4,000,000,000 (4 billion) bytes
- 64-bit addresses: ~16,000,000,000,000,000 (16 quintillion) bytes

Disk storage is ~30X cheaper than DRAM storage

- 8 GB of DRAM: ~ \$12,000
- 8 GB of disk: ~ \$200

To access large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk



Levels in Memory Hierarchy



DRAM vs. SRAM as a "Cache"

DRAM vs. disk is more extreme than SRAM vs. DRAM

- access latencies:
 - DRAM is ~10X slower than SRAM
 - disk is ~100,000X slower than DRAM
- importance of exploiting spatial locality:
 - first byte is ~100,000X slower than successive bytes on disk
 » vs. ~4X improvement for page-mode vs. regular accesses to DRAM
- "cache" size:
 - main memory is ~100X larger than an SRAM cache
- addressing for disk is based on sector address, not memory address



Impact of These Properties on Design

If DRAM was to be organized similar to an SRAM cache, how would we set the following design parameters?

- Line size?
- Associativity?
- Replacement policy (if associative)?
- Write through or write back?

What would the impact of these choices be on:

- miss rate
- \cdot hit time
- miss latency
- tag overhead

- 59 -

Locating an Object in a "Cache"



A System with Physical Memory Only

Examples:

• most Cray machines, early PCs, nearly all embedded systems, etc.



CPU's load or store addresses used directly to access memory.

A System with Virtual Memory

Examples:

= - 62 -



<u>Address Translation</u>: the hardware converts *virtual addresses* into *physical addresses* via an OS-managed lookup table (*page table*)

Page Faults (Similar to "Cache Misses")

What if an object is on disk rather than in memory?

- Page table entry indicates that the virtual address is not in memory
- An OS trap handler is invoked, moving data from disk into memory

Memory

- current process suspends, others can resume
- OS has full control over placement, etc.







Soln: Separate Virtual Addr. Spaces

- Virtual and physical address spaces divided into equal-sized blocks
 - "Pages" (both virtual and physical)
- Each process has its own virtual address space
 - operating system controls how virtual pages as assigned to physical memory





VM Address Translation

V = {0, 1, ..., N-1} virtual address space N > M P = {0, 1, ..., M-1} physical address space

MAP: $V \rightarrow P \cup \{\emptyset\}$ address mapping function

MAP(a) = a' if data at virtual address <u>a</u> is present at physical address <u>a'</u> in P = Ø if data at virtual address a is not present in P



VM Address Translation

Parameters

- P = 2^p = page size (bytes). Typically 4KB-64KB
- N = 2ⁿ = Virtual address limit
- M = 2^m = Physical address limit



Notice that the page offset bits don't change as a result of translation





Page Table Operation

Translation

- Separate (set of) page table(s) per process
- VPN forms index into page table

Computing Physical Address

- Page Table Entry (PTE) provides information about page
 - if (Valid bit = 1) then page in memory.
 - » Use physical page number (PPN) to construct address
 - if (Valid bit = 0) then page in secondary memory

»Page fault

» Must load into main memory before continuing

Checking Protection

- Access rights field indicate allowable access
 - e.g., read-only, read-write, execute-only
 - typically support multiple protection modes (e.g., kernel vs. user)
- Protection violation fault if don't have necessary permission

- 72 -
Integrating VM and Cache



Most Caches "Physically Addressed"

- $\boldsymbol{\cdot}$ Accessed by physical addresses
- $\boldsymbol{\cdot}$ Allows multiple processes to have blocks in cache at same time
- Allows multiple processes to share pages
- Cache doesn't need to be concerned with protection issues
 Access rights checked as part of address translation

Perform Address Translation Before Cache Lookup

- $\boldsymbol{\cdot}$ But this could involve a memory access itself
- Of course, page table entries can also become cached

Speeding up Translation with a TLB

"Translation Lookaside Buffer" (TLB)

- Small, usually fully associative cache
- Maps virtual page numbers to physical page numbers
- Contains complete page table entries for small number of pages





Alpha AXP 21064 TLB



TLB-Process Interactions

TLB Translates Virtual Addresses

 \cdot But virtual address space changes each time have context switch

Could Flush TLB

- Every time perform context switch
- \cdot Refill for new process by series of TLB misses
- ~100 clock cycles each

Could Include Process ID Tag with TLB Entry

- Identifies which address space being accessed
- OK even when sharing physical pages

Virtually-Indexed Cache



Cache Index Determined from Virtual Address

 $\boldsymbol{\cdot}$ Can begin cache and TLB index at same time

Cache Physically Addressed

- Cache tag indicates physical address
- Compare with TLB result to see if match - Only then is it considered a hit

What extra info needs to be included in cache? When Can Cache be virtually tagged?

Generating Index from Virtual Address



Size cache so that index is determined by page offset

Index

- Can increase associativity to allow larger cache
- E.g., early PowerPC's had 32KB cache
 - 8-way associative, 4KB page size

Page Coloring

- Make sure lower k bits of VPN match those of PPN
- Page replacement becomes set associative
- Number of sets = 2^k

(Single Level) Page Tables



- 4kb page size
- 512MB physical memory

```
Page table entries: 2<sup>64</sup>/2<sup>12</sup> entries!
Entry size: 2<sup>29</sup>/2<sup>12</sup> -> 17bits per entry + other
Page table size: 2<sup>52</sup>*2<sup>2</sup> = 2<sup>54</sup>!!
```

Alpha Virtual Addresses

Page Size

8KB (and some multiples)

Page Tables

- Each table fits in single page
- Page Table Entry 8 bytes
 - 4 bytes: physical page number
 - Other bytes: for valid bit, access information, etc.
- 8K page can have 1024 PTEs

Alpha Virtual Address

Based on 3-level paging structure

	level 1	level 2	level 3	page offset		
	10	10	10	13		

- Each level indexes into page table
- Allows 43-bit virtual address when have 8KB page size





Virtual Address Ranges

Binary Address Segment Purpose

- 1...1 11 xxxx...xxx seg1 Kernel accessible virtual addresses
 - Information maintained by OS but not to be accessed by user
- 1...1 10 xxxx...xxx kseg Kernel accessible physical addresses
 - No address translation performed
 - Used by OS to indicate physical addresses
- 0...0 0x xxxx...xxx seg0 User accessible virtual addresses
 - Only part accessible by user program

Address Patterns

- \cdot Must have high order bits all 0's or all 1's
 - Currently 64-43 = 21 wasted bits in each virtual address
- $\boldsymbol{\cdot}$ Prevents programmers from sticking in extra information
 - Could lead to problems when want to expand virtual address space in future

Alpha Seg0 Memory Layout



Regions

• Data

- Static space for global variables
 Allocation determined at compile time
 - » Access via \$gp
- Dynamic space for runtime allocation
 - » E.g., using malloc
- Text
 - Stores machine code for program
- Stack
 - Implements runtime stack
 - Access via \$sp
- Reserved
 - Used by operating system <u>» shared libraries</u> process info, etc.

Alpha Seg0 Memory Allocation

Address Range

- User code can access memory locations in range 0x000000000010000 to 0x000003FFFFFFFFF
- Nearly $2^{42} \approx 4.3980465 \ X10^{12}$ byte range
- In practice, programs access far fewer

Dynamic Memory Allocation

- Virtual memory system only allocates blocks of memory as needed
- As stack reaches lower addresses, add to lower allocation
- As break moves toward higher addresses, add to upper allocation

= - 86 - =

- Due to calls to malloc, calloc, etc. "



Minimal Page Table Configuration



Partitioning Addresses							
Address 0x001 2000 0000 00000000000000000000000000							
000000000 100100000 00000000 0000000000							
• Level 1: 0 Level 2: 576 Level 3: 0							
Address 0x001 4000 0000 0000 0000 0001 0100 0000 00							
000000000 10100000 00000000 00000000000							
• Level 1: 0 Level 2: 640 Level 3: 0							
Address 0x3FF 8000 0000 0011 1111 1111 1000 0000 0000							
011111111 11000000 00000000 00000000000							
• Level 1: 511 Level 2: 768 Level 3: 0							

______ - 88 - ______ CS 740 _____



Increasing Heap Allocation



Expanding Alpha Address Space

Increase Page Size

- Increasing page size 2X increases virtual address space 16X
 - 1 bit page offset, 1 bit for each level index

level 1	level 2	level 3	page offset
10+k	10+k	10+k	13+k

Physical Memory Limits

- Cannot be larger than kseg
 VA bits -2 ≥ PA bits
- Cannot be larger than 32 + page offset bits
 - Since PTE only has 32 bits for PPN

Configurations

•	Page Size	8K	16K	32K	64K
•	VA Size	43	47	51	55
•	PA Size	41	45	47	48



VM Theme

Programmer's View

- Large "flat" address space
 - Can allocate large blocks of contiguous addresses
- Process "owns" machine
 - Has private address space
 - Unaffected by behavior of other processes

System View

- \cdot User virtual address space created by mapping to set of pages
 - Need not be contiguous
 - Allocated dynamically
 - Enforce protection during address translation
- OS manages many processes simultaneously
 - Continually switching among processes
 - Especially when one must wait for resource
 - » E.g., disk I/O to handle page fault

The Memory Hierarchy

- Exploits locality to give appearance of a large, fast flat address space owned by a process
 - cache(s)
 - VM
 - TLB
- Hierarchical to exploit qualities of each technology while reducing cost of overall system
- Interaction between cache design and VM design
- What should be automatic and what "manual"?
 - Registers? Cache placement? TLB management?