Program Optimization

- Code motion
- Reduction in strength → Shift, add instead of multiply or divide
- procedure calls → optimization blocker#1 compiler won't do code motion to prevent side effects. (strlen example)
- Memory matters

```
b[i] = b[i] + a[i*n+j]
```

(load, add, store) (as opposed to adding in the register in place)

compiler cannot optimize because of memory aliasing
↳ 2 different memory references specify single location)
↳ Introduce local variables that accumulates within loops
This is a way to tell compiler not to check for aliasing.

- Translation

Jump.

Instruction control unit determines the destination of Jump
predicts whether will be taken
starts fetching instruction at predicted destination
Execution unit checks whether prediction was ok.
if not → signals instruction control
instruction control invalidates any operations generated from misfetched instructions.
begins fetching & decoding instructions at correct target.

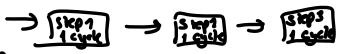
Unrolling & Accumulating:

- can unroll to any degree L
- can accumulate K results in parallel
- L must be multiple of K.

SIMD operations: (Single Inst Multiple Data)
single precision → 8 additions
double " → 4 "

Latency vs Throughput

	latency	cycles/issue
Integer multiply	3	1



3 independent multiplications will be pipelined.
Sequentially dependent won't be that efficient.

T = CPE * n + Overhead
↳ is a slope of line

cycles/issue: 1
latency: 3

p1 = a * b
p2 = a * c
p3 = p1 * p2

Pipelined Functional Units

	1	2	3	4	5	6	7
Stage 1	a*b	a*c			p1*p2		
Stage 2		a*b	a*c			p1*p2	
Stage 3			a*b	a*c			p1*p2

Branches → IU must work well ahead of Execution Unit to generate enough operations to keep EU busy.
When encounters conditional branch, cannot reliably determine where to continue fetching.

The Memory Hierarchy

RAM: → SRAM: retains value indefinitely, insensitive to electrical noise, faster & expensive (6-transistor circuit)
→ DRAM: value must be refreshed, sensitive " " ", slower & cheaper (capacitor & transistor)
↳ reading the content of a memory cell destructs its content.

basic storage unit is a cell
multiple RAM chips form a memory

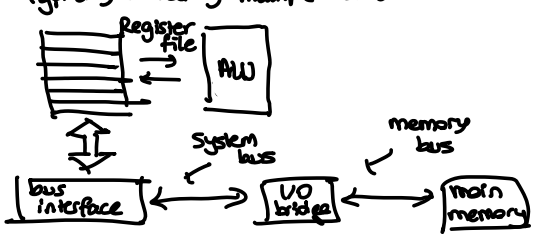
DRAM & SRAM are volatile memories

Nonvolatile memories → ROM → PROM
→ EPROM
→ EEPROM
→ Flash Memory

Firmware → Program stored in ROM (boot time code, BIOS, graphics card, disk controllers)

Traditional Bus Structure

Bus is a collection of parallel wires that carry address, data & control signals.
Typically shared by multiple devices.



Memory Read Transaction:

① CPU places address A on the memory bus
② Main memory reads from the memory bus, retrieves word x and places it on the bus.
③ CPU read word x from the bus and copies it into register %rax

Memory Write Transaction:

① CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.
② CPU places data word y on the bus
③ Main memory reads data word y from the bus and stores

it at address A.

Locality

Principle of Locality: Programs tend to use data & instructions with addresses near or equal to those they have used recently.

Temporal Locality: Recently referenced items are likely to be referenced again in the future.

Spatial Locality: Items with nearby addresses tend to be referenced close together in time.

Locality Example:

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

Data References:
reference array elements in succession
reference variable sum each iteration.

Instruction References
reference instructions in sequence
cycle through loop repeatedly

L0: Registers
L1: L1 Cache (SRAM)
L2: L2 Cache (SRAM)
L3: Main Memory (DRAM)
L4: Local Secondary Storage (Local disks)
L5: Remote Secondary Storage (tapes, distributed file systems, web servers)

↑ smaller, faster, costlier per byte
↓ larger, slower, cheaper per byte.

Caches:

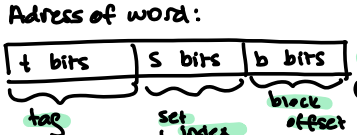
fundamental idea of memory hierarchy → For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1

Because of locality, programs tend to access the data at level k more often than they access the data at level k+1. Thus the storage at level k+1 can be slower, and thus larger and cheaper per bit.

▽ The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

Read:
- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset.

Address of word:



General Cache Organization (S, E, B)

Direct Mapped Cache (E=1) → lines per set

S = 2^s sets

① One line per set

Assume: cache block size is 8 bytes

address of int:

t bits	0...01	100

find set

valid? match: assume yes = hit block offset

Address trace (reads, one byte per read)

0 [0000$_2$] miss → set index

1 [0001$_2$] hit → tag

7 [0111$_2$] miss → block offset

8 [1000$_2$] miss

0 [0000$_2$] miss

address: 0111 → 0110 / 0111 > page that we are looking for in memory

t=1 s=2 b=1

x	xx	x

M = 16 byte addresses, B = 2 bytes/block

S = 4 sets, E = 1 Blocks/set

	v	Tag	Block	
(00)	Set 0	1	0	M[0] M[1]
(01)	Set 1	0		
(10)	Set 2	0		
(11)	Set 3	1 → 0	0	M[6] M[7]

memory: M[0] M[1] ... M[6] M[7] ... 15

0 → bring from memory

7

8 → will change set 0 as:

v	Tag	Block
1	1	M[8] M[9]

E-way Set Associative Cache

E = 2: Two lines per set

Assume: cache block size: 8

Address of short int: 1 bits | 0...01 | 100 (tag bits)

3 bits as block size is 8 bytes

find set

t=2	s=1	b=1
xx	x	x

tag, set index, block offset

	v		
Set 0	0		0
Set 1	0		0

0 → Set 0: 1 | 00 | M[0] | M[1]

Set 1: 0

7 → Set 0: 1 | 00 | M[0] | M[1]

Set 1: 1 | 01 | M[6] | M[7]

8 → Set 0: 1 | 00 | M[0] | M[1]

Set 1: 1 | 01 | M[6] | M[7]

Address Trace

0 [0000$_2$] → miss

1 [0001$_2$] → hit

7 [0111$_2$] → miss

8 [1000$_2$] → miss

0 [0000$_2$] → hit

Write hit:

- Write through: write immediately to memory
- Write back: defer write to memory until replacement of line (need a dirty bit)

Write Miss:

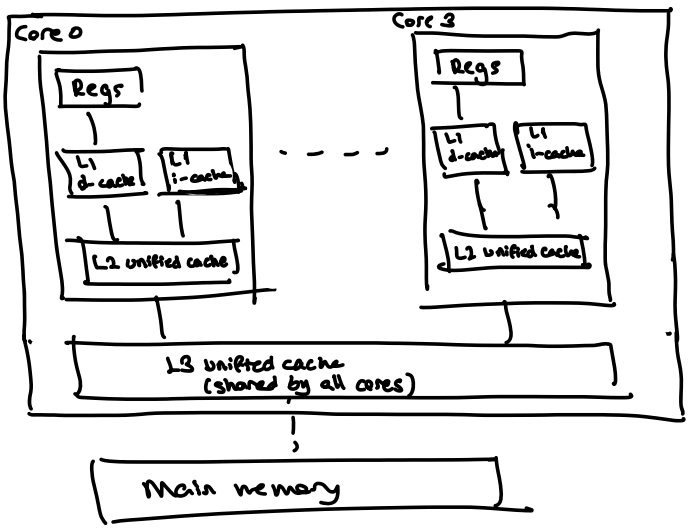
- Write allocate: load into cache, update line in cache
- No-write allocate: writes immediately to memory

Typical:

Write through + No write-allocate

Write back + Write allocate

Intel core i7 Cache Hierarchy



L1: 8 way

L2: 8 way

L3: 16 way

Block size: 64 bytes for all caches

Direct mapped → easy (comparison)
2-way → too many tag comparisons but less collisions

Types of cache misses

Cold (compulsory) miss:
occur because cache is empty

conflict miss:
occurs when the level k cache is large enough, but multiple data objects all map to the same level k block.

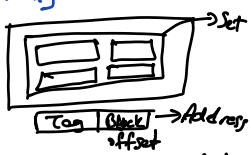
capacity miss:
occurs when the set of active cache blocks (working set) is larger than the cache.

Cache Summary

parameters: $C = S \times E \times B$ → data block size
- S → # of sets
- E → # of lines per set

only the total # of data bytes are stored

Fully Associated



Maximum # of tag comparisons needed

There are several cache types, but Aybüke didn't let me write them.

Layout of C arrays in Memory

- allocated in row-major order (each row in contiguous memory locations)
- stepping through columns in 1 row

```
for (i=0; i<N; i++)
    sum+= a[0][i]
```

→ accesses successive elements
$a_{00}, a_{01}, a_{02}, a_{03} \dots$
if block size > 4, exploit spatial locality
compulsory miss rate = 4 bytes / B

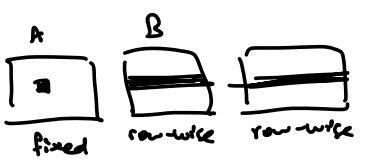
- stepping through rows in 1 column

```
for(i=0; i<N, i++)
    sum+= a[i][0]
```

→ accesses distant elements
no spatial locality
compulsory miss rate = 1 (100%)
$a_{00}, a_{10}, a_{20} \dots a_{n0} \dots$

Matrix Multiplication

		misses per inner loop
ijk loop →	a[i][k] →	0.25
	b[k][j] →	1.0
	c[i][j] →	0.0
jik →	~	0.25
		1.0
		0.0
kij →	a[i][k] →	0.0
	b[k][j] →	0.25
	c[i][j] →	0.25
jki →	c[i][j] →	1.0
	a[i][k] →	1.0
	b[k][j] →	0.0



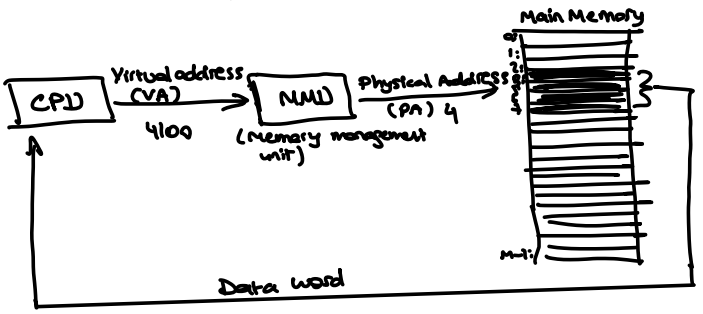
Summary

ijk (& jik): 2 loads, 0 stores, 1.25 misses

kij (ikj): 2 loads, 1 store, 0.5 misses

jki (kji): 2 loads, 1 store, 2.0 misses

Virtual Memory



Address Spaces:

Linear address space: ordered set of contiguous non-negative integer addresses.

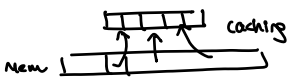
Virtual address space: Set of $N = 2^n$ virtual addresses $\{0, 1, 2, 3, \ldots, N-1\}$

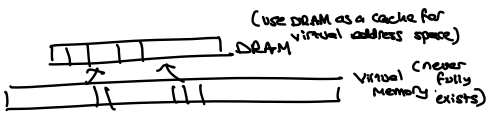
Physical address space: Set of $M = 2^m$ physical addresses. $\{0, 1, 2, 3 \ldots, M-1\}$

Every byte in main memory has one physical address and 0 or more virtual addresses.

Why Virtual Memory?

1. Uses main memory efficiently
   - Use DRAM as a cache for parts of a virtual address space.
2. Simplifies memory management
   - Each process gets the same uniform linear address space
3. Isolates address spaces
   - One process can't interfere with another's memory.
   - User programs cannot access privileged kernel information and code.

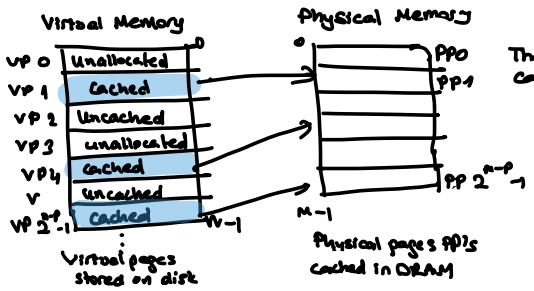




VM as a Tool for Caching

Virtual memory is an array of N contiguous bytes stored on disk.

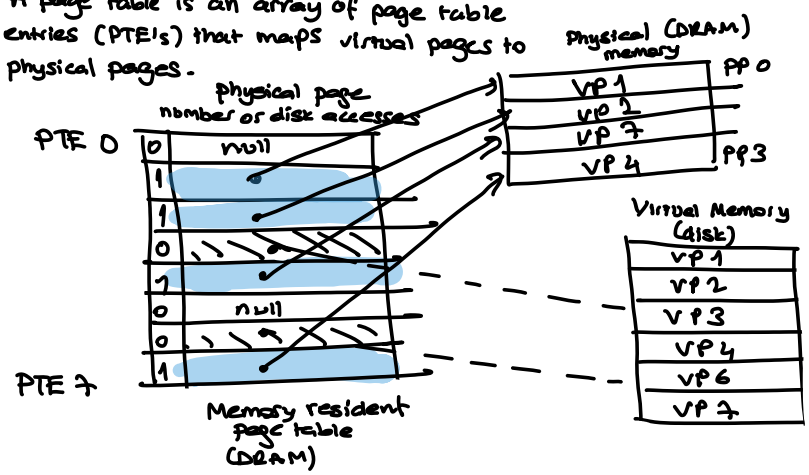
The contents of the array on disk are cached in physical memory (DRAM cache)



These cache blocks are called pages (size = $P = 2^p$)

Enabling Data Structure: Page Table

A page table is an array of page table entries (PTE's) that maps virtual pages to physical pages.



Page Hit: reference to VM word that is in physical memory (DRAM cache hit)

Page Fault: " " " " that is not in physical memory.
(when it is mapped to a valid bit 0.
page miss causes page fault.
page fault handler selects a victim to be evicted (here VP4) and updates it with the new one.
Offending instruction is restarted → page hit

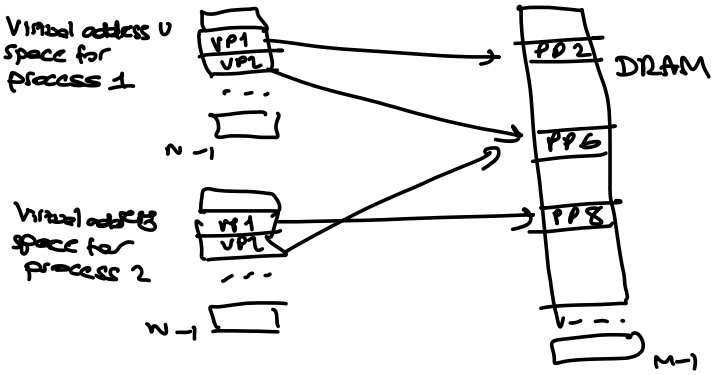
Set of active virtual pages → working set

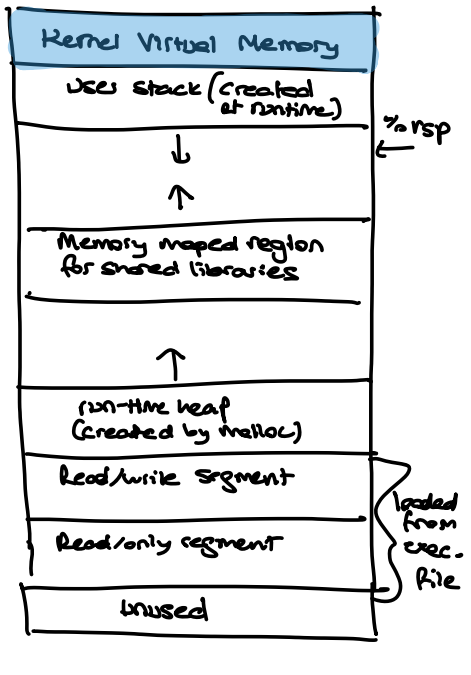
if working set size < main memory size:
good performance after compulsory misses

if sum(working set size) > main memory size
thrashing: performance meltdown where pages are swapped in & out continuously

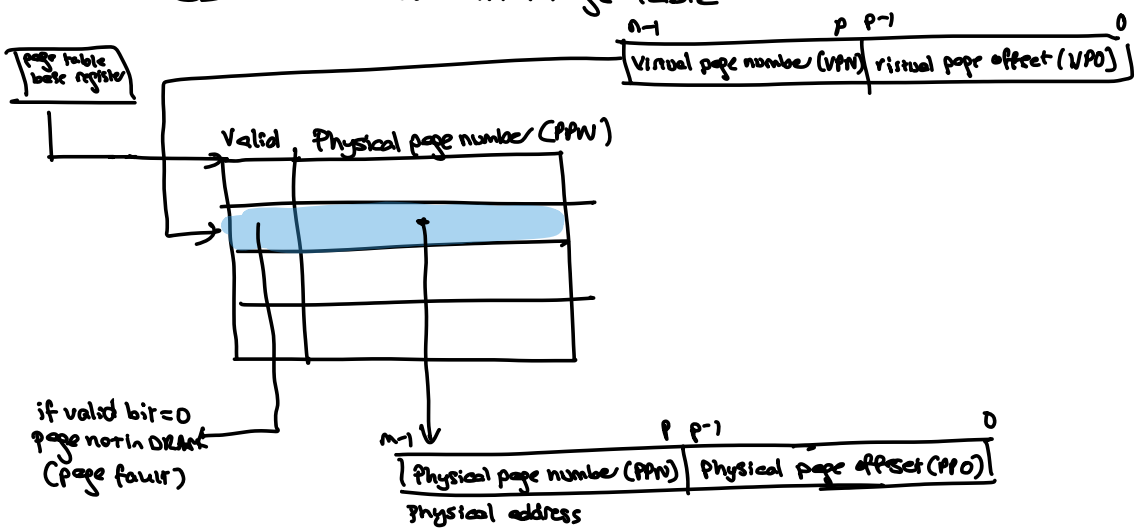
VM as a Tool for Memory Management:

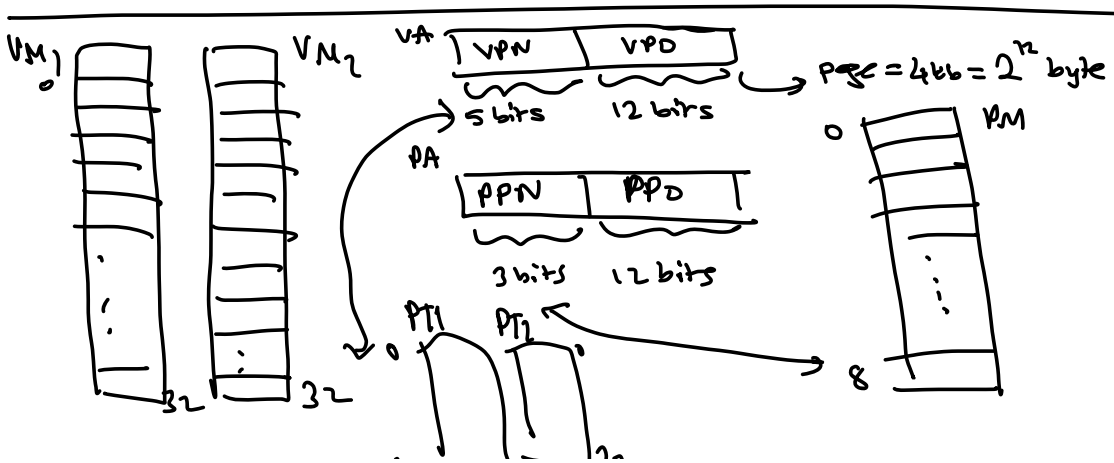
Key idea: each process has its own virtual address space.





Address Translation with a Page Table





VM as a Tool for Memory Protection

- Extend PTE's with permission bits
- Page Fault Handler checks these before remapping.

	SUP	READ	WRITE	EXEC	Address
VP0	No	Yes	No	Yes	PP6
VP1	No	Yes	Yes	Yes	PP4
VP2	Yes	Yes	Yes	No	PP2

Speeding up Translation with a TLB → cache for Page Table

PTE's are cached in L1 like any other memory word.

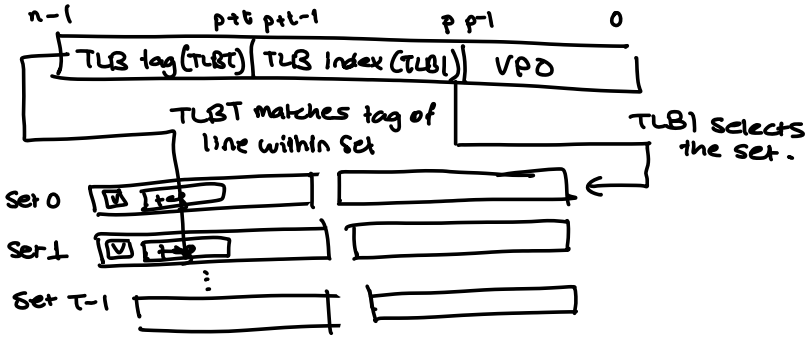
PTE's may be evicted by other data references

PTE hit still requires a small L1 delay

Solution: Translation Lookaside Buffer (TLB)

- Small set-associative hardware cache in MMU
- maps virtual page numbers to physical page numbers
- contains complete page table entries for small number of pages.

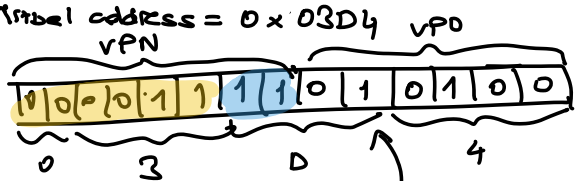
MMU uses the VPN portion of the virtual address to access the TLB.



Address Translation Example:

VA → VPN → TLB → PPN → PA

Virtual address = 0x03D4



VPN = 0x0F

TLBI = 0x3

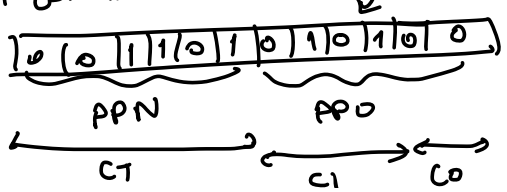
TLBT = 0x03

PPN = 0x0D

TLB Hit: Y

Page Fault: N

Physical Address



CO: 0

CI = 0x5

CT: 0x0D

Hit: Y

Byte: 0x36

Address Translation (Page Hit)

1) Processor sends virtual address to MMU.
2) MMU fetches PTE from page table in memory
3) MMU sends physical address to cache/memory
4) Cache/memory sends data word to processor.

2 memory accesses

Address Translation (Page Fault)

1) Processor sends virtual address to MMU.
2) MMU fetches PTE from table in memory.
3) Valid bit is 0, so MMU triggers page fault exception
4) Handler identifies victim
5) Handler pages in new page & updates PTE in memory.
6) Handler returns to original process restarting faulting instruction

Size of the PT

32 bit → VA 32

|Page| = 4KB (2^{12})

|PM| = 16B (2^{30})

|VM| = 4GB (2^{32})