Emery Berger

emeryberger.github.io / CompSCI-630

Systems Lunch @ Friday — Fall

random number

TA Abhinav Jangda

github!

Hot CRP

emery@cs.umass.edu

- SCRIBES!
- course load:
  - 1 core / semester for MS
  - " 1st semester MS/PhD
  - > 1 core = too much!
- schedule - yes, we have a midterm & final!
- reviews

FORTRAN (& COBOL)

1954

military - Calculating artillery trajectories
- decryption

first computers - people!

general-purpose computing machine

automatic computers

Turing machine

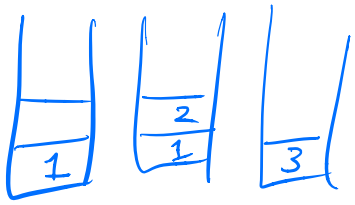
von Neumann - machine] data & code together

Harvard architecture

fetch-decode-execute cycle

assembly language

- lowlevel — direct access to memory
  - operations not abstract
- stack architecture
- register architecture



PUSH 1
PUSH 2
ADD
POP

A, B, C, D
AX BX CX DX

MOV 1, A
MOV 2, B
ADD A, B, C

"Intel" "ARM" (subset of instr.)

JVM, .NET, Python bytecodes, WASM

- portability

ASM ≠ portable

backward compatibility

- hard to read, hard to debug

---

1. PORTABLE
2. READABLE
3. WRITABLE

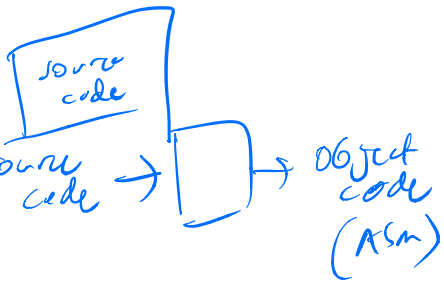
- COBOL – manages } domain-specific languages
  FORTRAN – scientific }

④ FAST

Python
Ruby

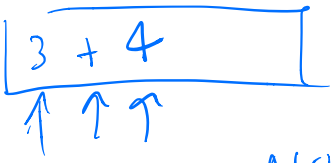
Matlab
LOGO
SQL — non Turing complete
CUDA
Excel
—

interpreted
vs.
compiled

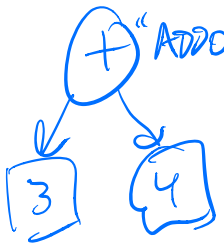


matmul
4k * 4k
~ 7 min.

numpy —
~ .4 s



Abstract syntax tree (internal representation IR)




```
switch (op) {
  :
case ADDOP:
  result = t->left +
           t->right;
  break;
  pc++;
  :
}
```

ADD A, B, C

programmer time MORE VALUABLE than cycles
— modern POV

- productivity
- debuggability
~ 100x OK !?)

interpreter → just-in-time compiler (JIT)

Python PyPy

JavaScript V8
Chrome

WASM

---

predates - parsing
- compiling
- program analysis
- typechecking
- scope
- object-orientation
- modularity
- structured prog.

FORTRAN
1st optimizing compiler

I, J, K integer

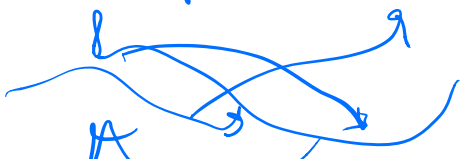
X, Y, Z float

```
DO  10  I = 1,10
10  CONTINUE
```

(was comma) line numbers

BASIC

"spaghetti code"



control flow

DO 10 I = 1.10

DO10I = 1.10

FORTRAN 77 , FORTRAN 90 — evolving

LEGACY

FAST The Fastest PL today! 2019 !!

optimization

static analysis + transformations

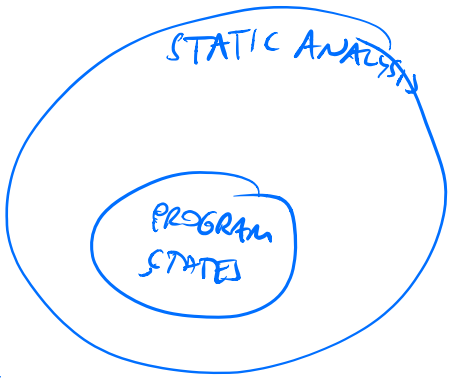
semantics-preserving

compile time × ~~runtime~~

halting(p)

if (false)

$X \in \{1, 2, 3\}$



conservative view

over-approximation

Halting problem

halts (program_text)

⇒ true if program always terminates

⇒ false otherwise

p' if halts(p) run forever

else terminate

(1) Constant folding

$x = 3 + 4$

⇓

$x = 7$

* PI * PI

(2) Constant propagation

$x = 7$ $z = 12$

$y = x + 3 + z$

⇓

$x = 7$ $z = 12$

$y = 22$

(3) ``strength reduction''

$x = x \; (**) \; 3$ — expensive

$\Downarrow$

$x = x * x \; (*) \; x$ — cheap

/ %

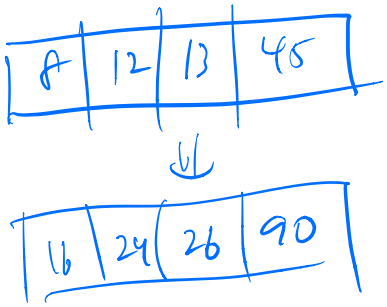
$x = x \;\%\; 8$

$\Downarrow$

$x = x \;\&\; 7$

111

④ vectorization




```
for (i = 0; i < 16; i++) {
    a[i] *= 2;
}
```

$\Downarrow\Downarrow$

vector ops (4x speedup)

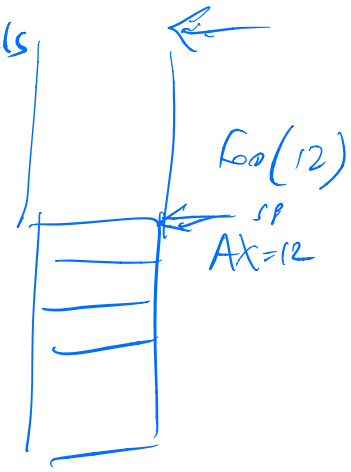
⑤ inlining

function prolog

copy arguments into locals
create stack frame

```
foo(int x) {
  ...
  return i1   jump
}
```

stack resident



epilog

copies out return value
restores registers
pop frame
returns (jump)

```
int add(int a, int b) {
  return a + b;
}

foo() {
  z = add(a, b)
}
```

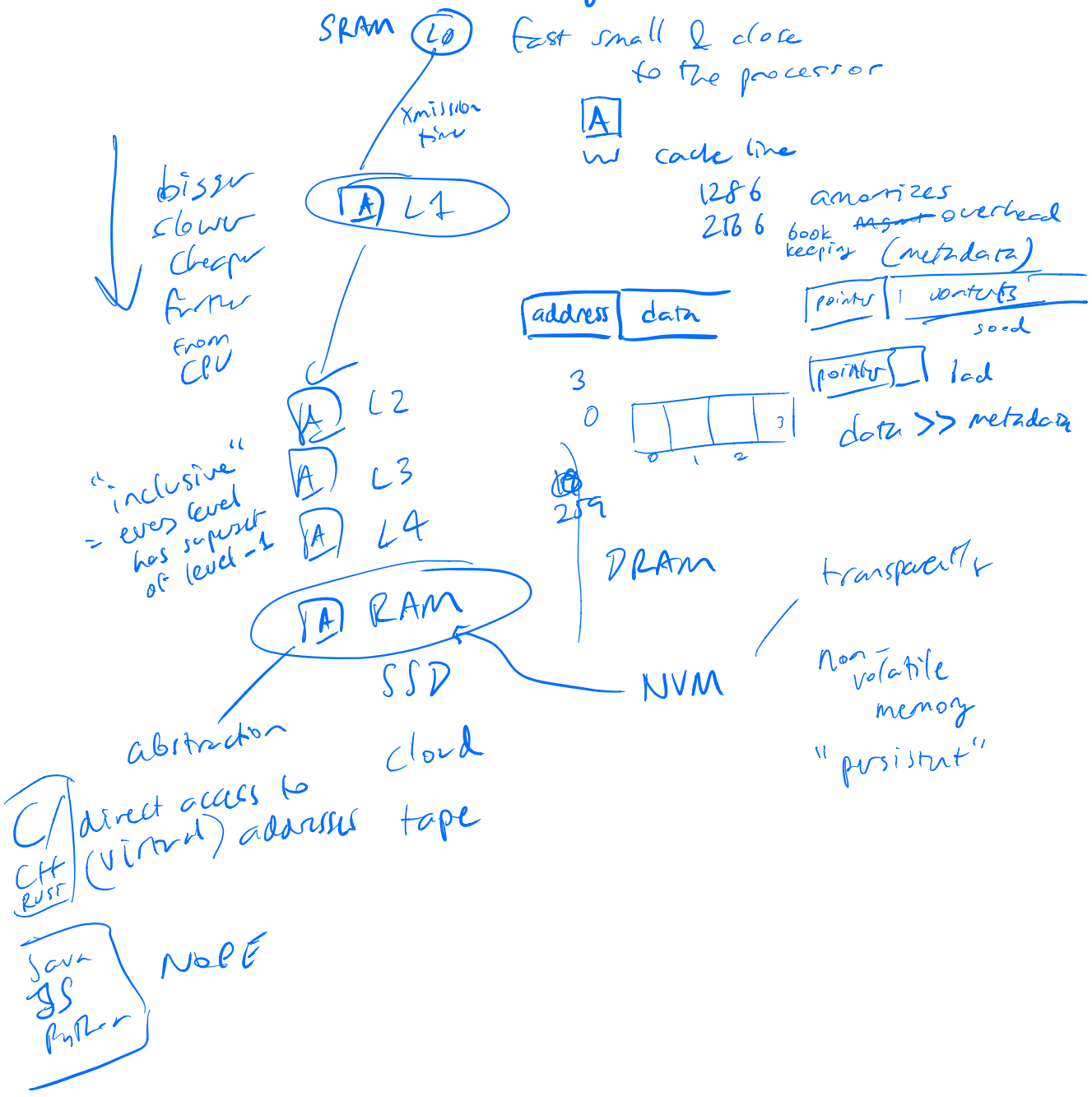
⇓

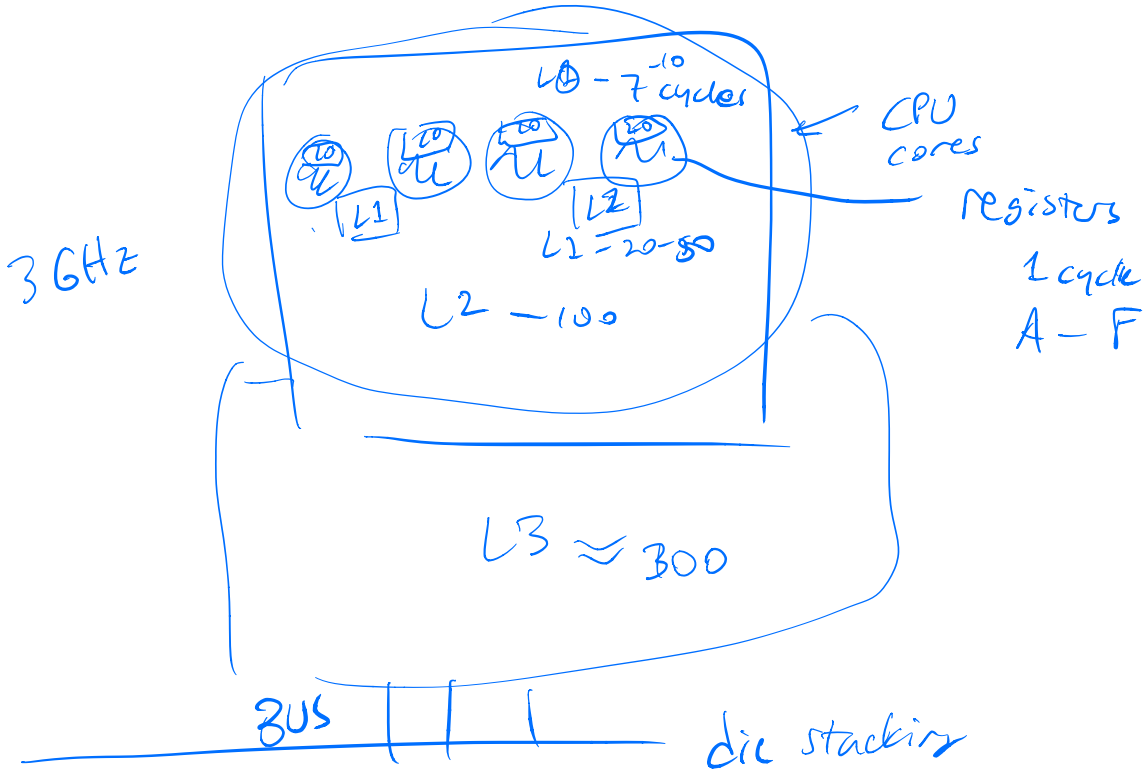
```
foo() { z = a + b; }
```

- code size
- I$ instruction cache

---

memory hierarchy





RAM "1 GB/s"

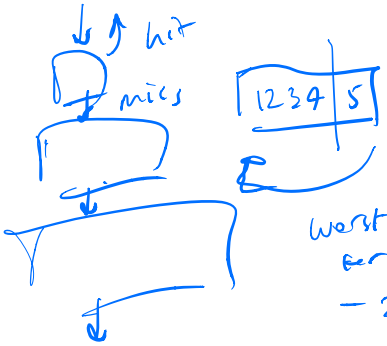
RAM bandwidth

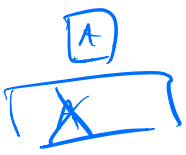
~~1000~~

10,000

caches — "replacement policy"

eviction "LRU"





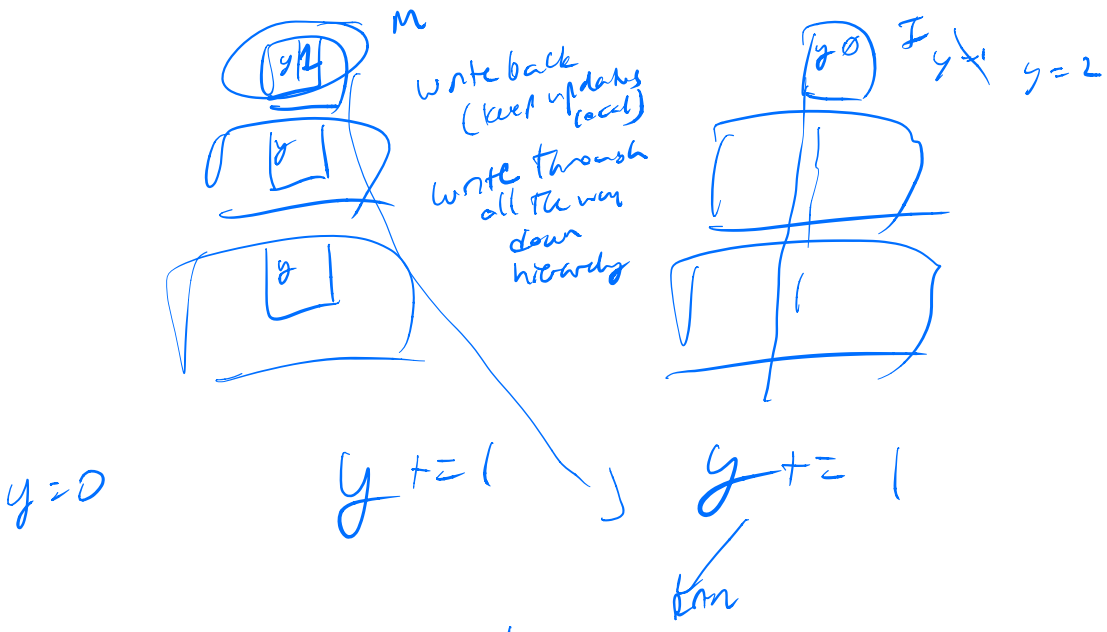
latency

wait time

caches hide latency

worst-case for LRU

- ≈ OK for inclusive
- catastrophe for exclusive



cache coherence

MESI protocol

- Modified — I changed it
- Exclusive — just me
- Shared — several CPUs have it (unmodified)
- Invalid — trash

approx LRU implemented

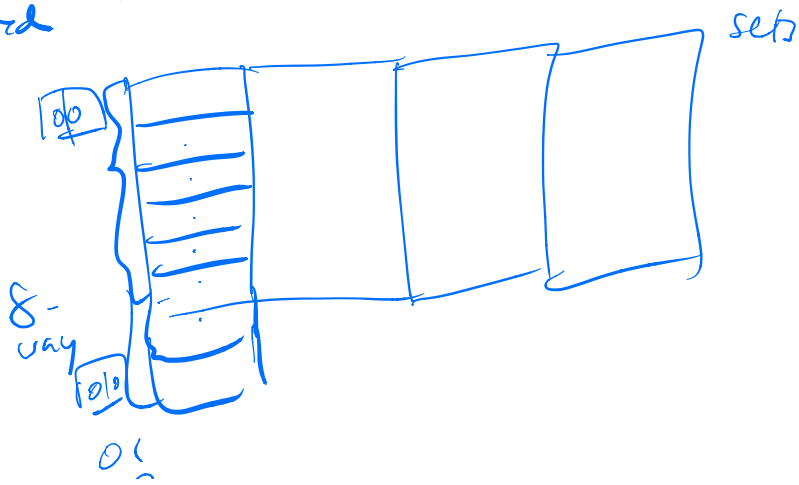
(addr % #cache line size) / cache line size

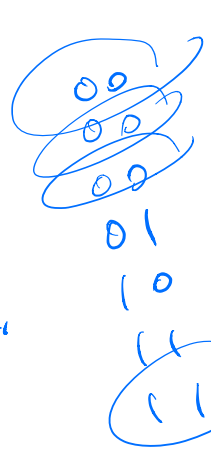
Time

3 addr

5 addr

addr



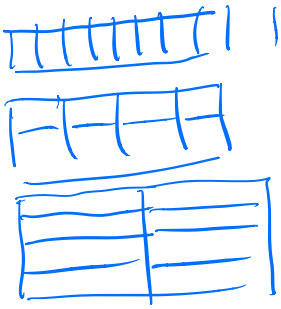


Cache misses
Can in different
types -

Capacity
Conflict — "hash collision"
Coherence 3C's model

saturated counter

direct mapped — 1-way
2-way
4-way
8-way
16-way
fully associative — ∞ way



TLB
translation lookaside buffer

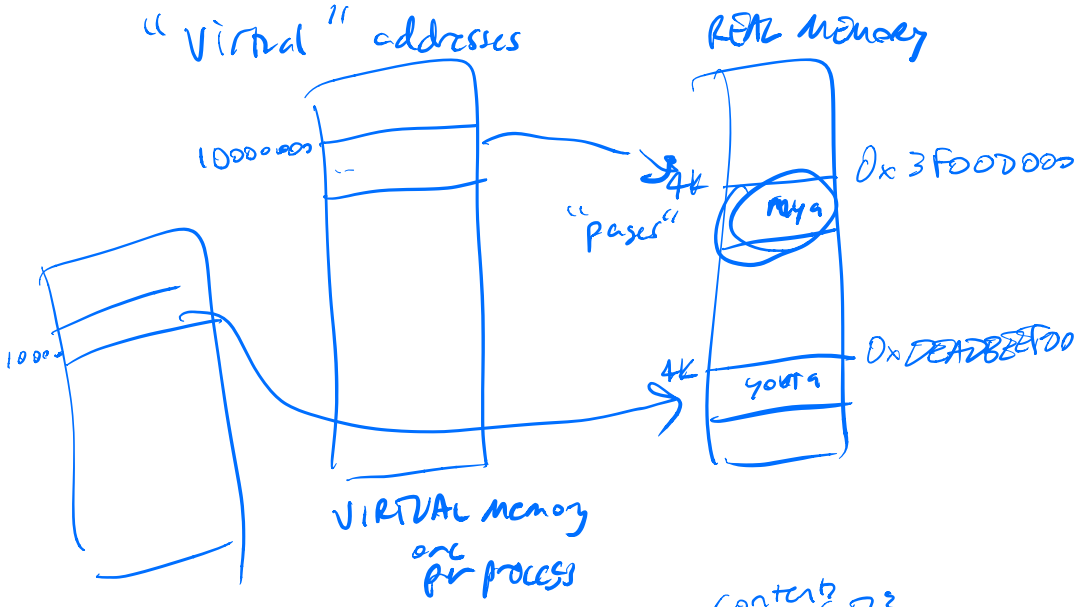
CPU cache

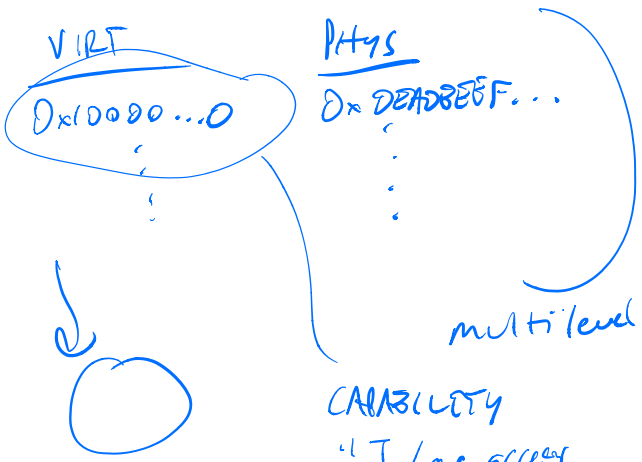
cache of virtual → physical addresses

```
#include <stdio.h>
int main() {
    int a;
    printf("&a = %p\n", &a);
    return 0;
}
```

0x10000000
address

process isolation





CAPABILITY
"I have access to this memory"

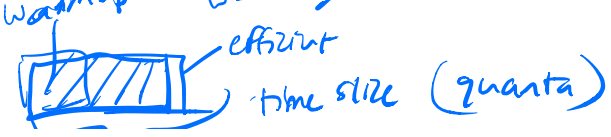
TLB flush
— discard all TLB entries

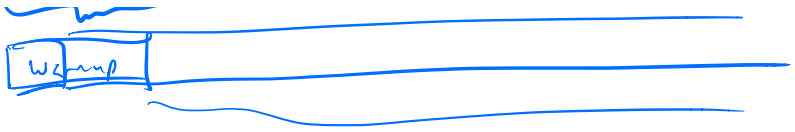
---

time slicing
scheduling

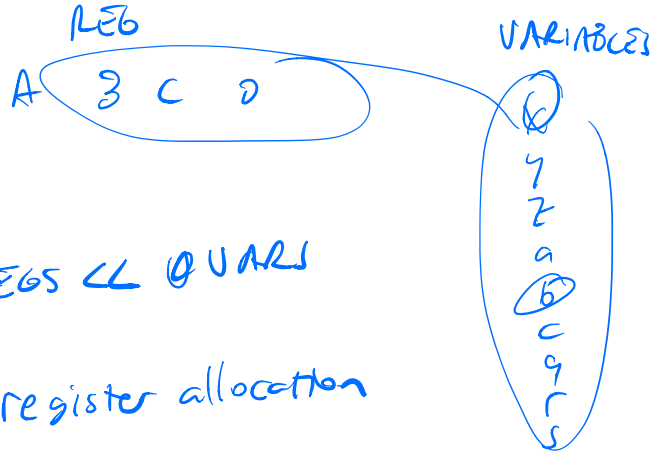
"cold" cache
warming up the cache

CONTEXT SWITCH overhead
processes ↑
threads ↑





amortize warmup
across the queries



REGS << VARS

register allocation

"spill" — copy contents of register to mem "eviction"!

straight-line code
— predictable!

"graph coloring" NP complete!

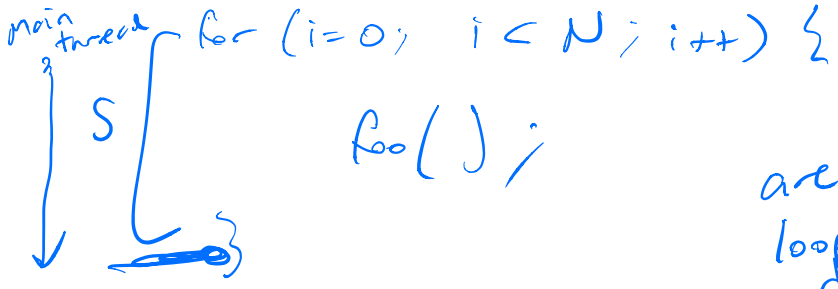
Why is FORTRAN fast?

~~OMP PARALLEL~~
DO I = 1, 100
DO J = 1, 100
A[I][J] = A[I][J-1]+1
ENDDO
ENDDO

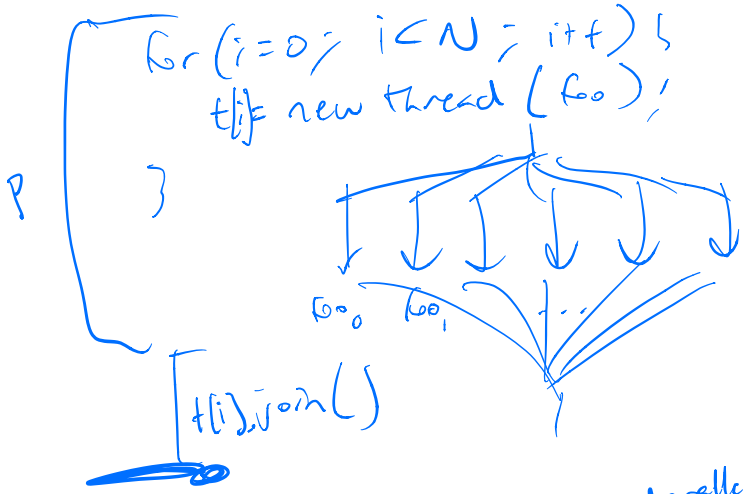
pragma omp
OpenMP
C
~~"hint"~~

directives

automatic parallelization
"holy grail"



are there loop carried dependencies?



result(S) ≡ result(P)

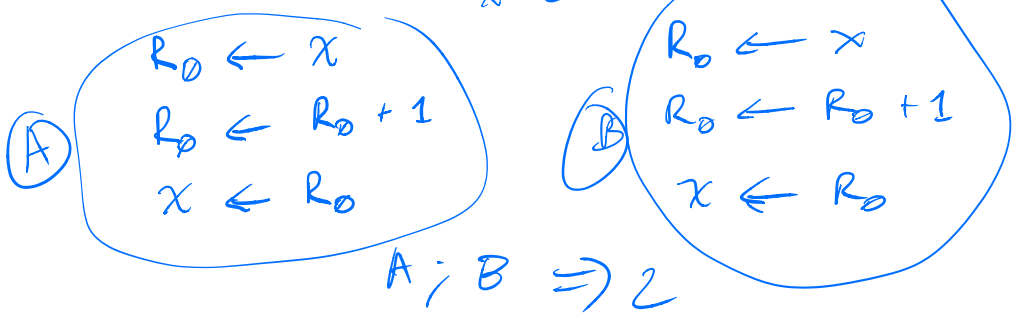
parallel prog semantically equivalent to seq prog.

does foo() have side effects?

```
foo() {
  x++;
}
```

(global)

read x into a register
increment it
write the result back
x = 0

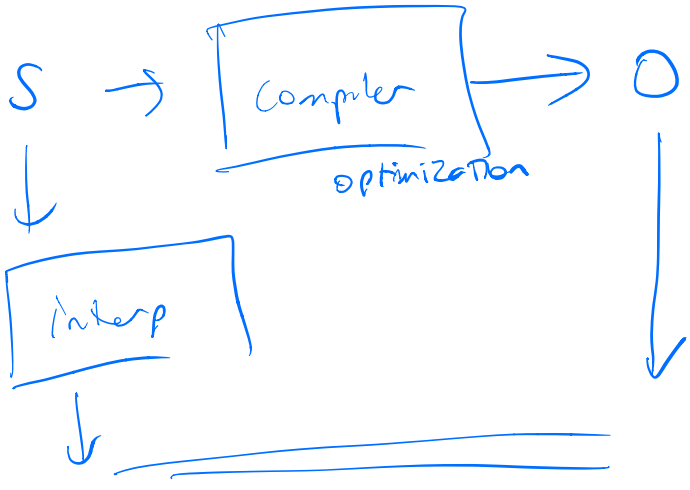


A ; B ⇒ 2

$X \in \{1, 2\}$

non-determinism!

race condition concurrency errors




```
for (i=0; ic
  x = 12
  y = x + 100
  z = y ** 100
  q = 2!
  cout << q;
```

dead code elimination

```
foo() {
~ x ~
}
```

```
foo (int i) {
  ~ i ~
  return ack(i, i+1);
}
```

Ackermann

```
foo(int *a, int i) {
  a[i] = ack(i, i+1);
}

foo(int *a, int *b, int i) {
  a[i] = ...
  b[i] = a[i] + 1;
}
```

```
int x[200];  int *y = &x[0];
v = foo(x, y, i)
```

aliasing

alias analysis
pointer analysis

p { }
q { }

$p \cap q = \emptyset$

```
int *p;   p ∈ {}
if (...) {
  p = q;      p ∈ {q}
} else {
  p = r;      p ∈ {r}
}
```

symbolic execution

abstract interpretation

$p \in \{q, r\}$

```
x = 12
x = x - 100
x . . . .
```

X O, N, P

```
if (x > 0) {
    |
}
```

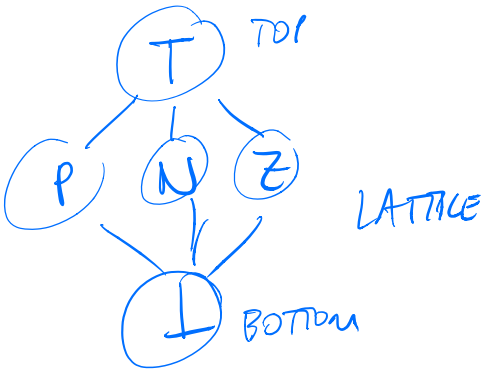
does this run ever?

```
x = ___
x = x - 1
```

$\{\emptyset, N, P\}$

$\emptyset \Rightarrow \emptyset$
$N \Rightarrow P$
$P \Rightarrow N$

CONSERVATIVE APPROX PROG BEHAVIOR



$\{x : T\}$

```
if (-) {
    x = -1
} else
    x = 1
}
```

$\{x : N\}$

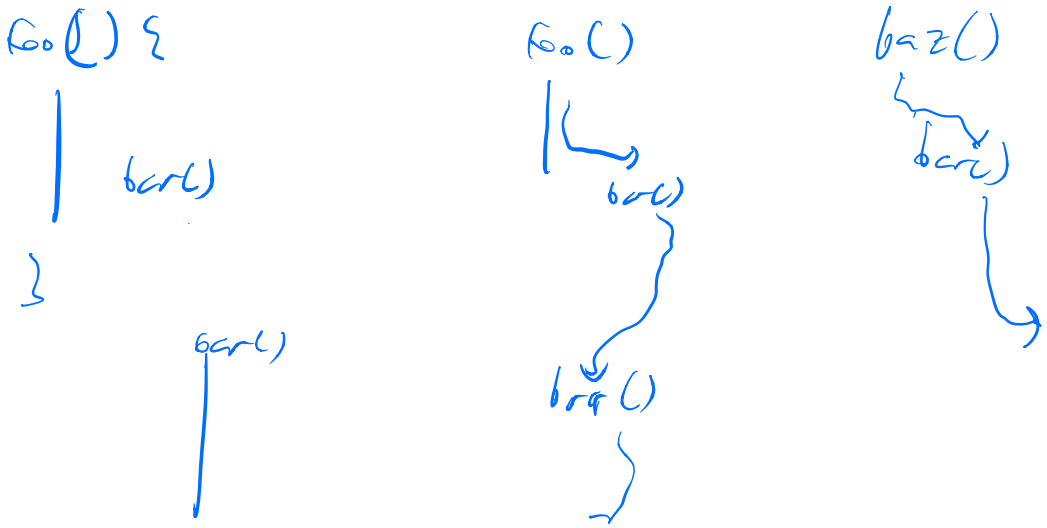
$\{x : P\}$

$\{x : \bot\}$

```
if
    x : P
else
    x : P
```

$x : P$

intraprocedural vs. interprocedural



Context sensitivity

Flow sensitivity

Steensgard's algorithm

RCR sensitivity

false positives

false negatives

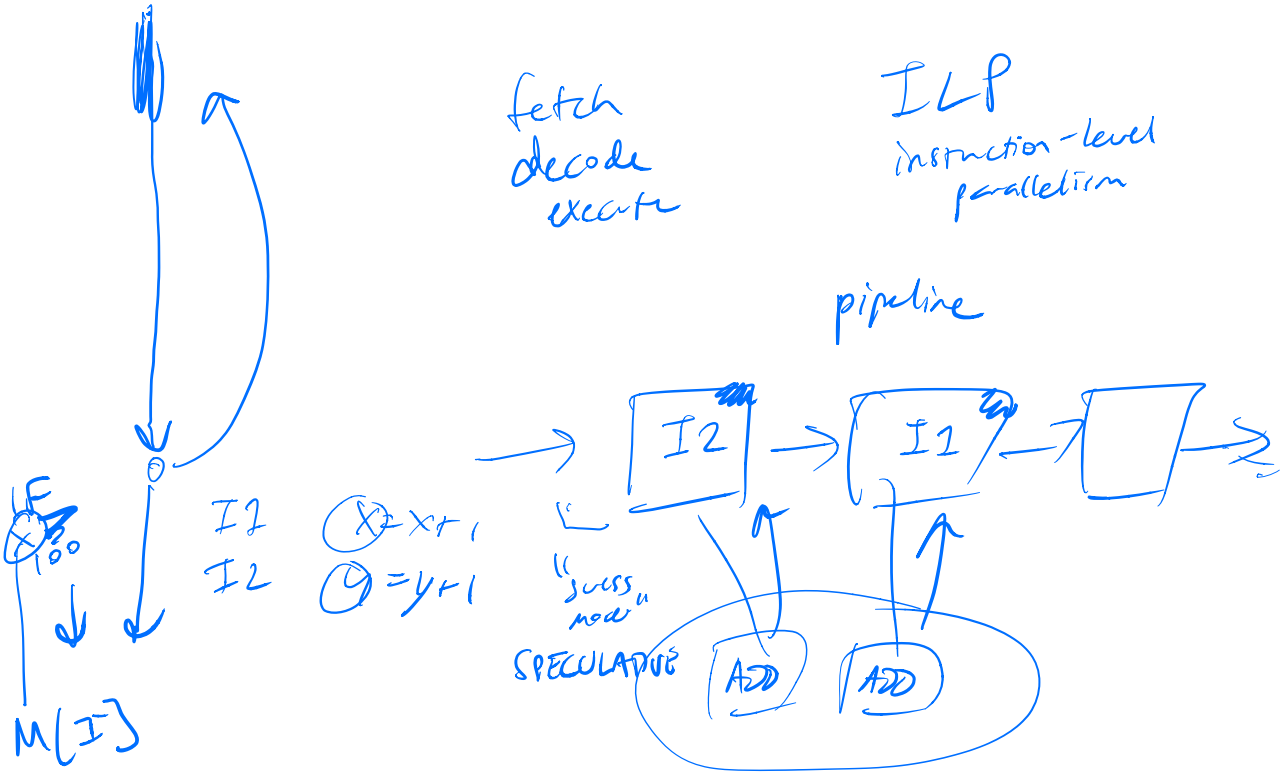
precision - recall tradeoff

$\top$ — { ⊘ }

if (x == ——)

y = x / ⊘ ⊤

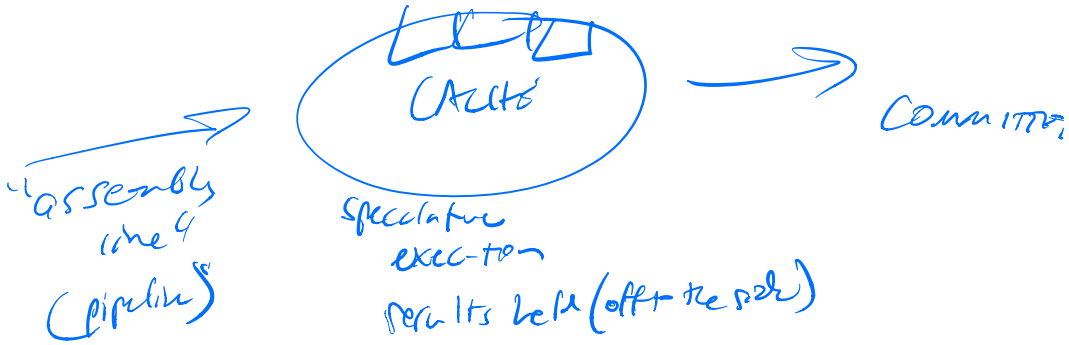
—W



PIPELINE STALL

"bubble"

"BACK EDGE TAKEN"



Speculative execution

hide memory latency

SPECTRE / Meltdown

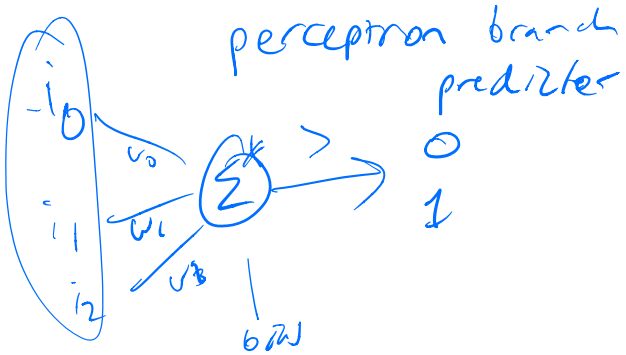
timing channel — covert channel

side channels

last branch taken — 85-90% effective

PHT

MODERN BRANCH PREDICTORS — 99.9% effective



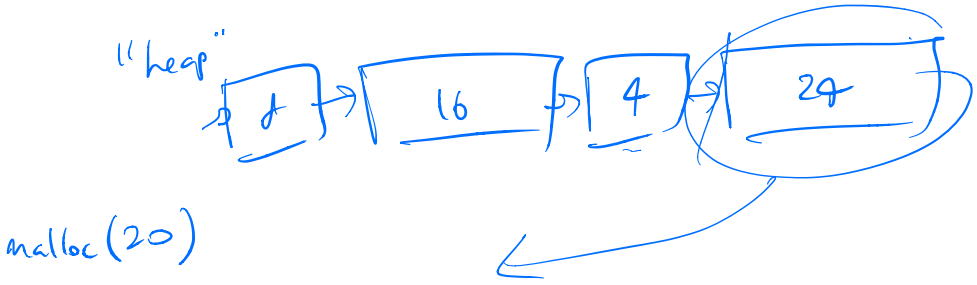
---

dynamic memory allocation

global

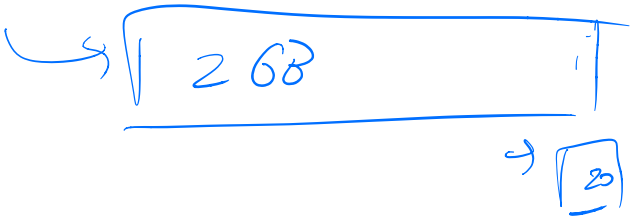
stack variables

dynamic memory — "the heap"



Project 2

First-fit — SPACE INEFF



best-fit

SPACE EFFICIENT

SLOW ....

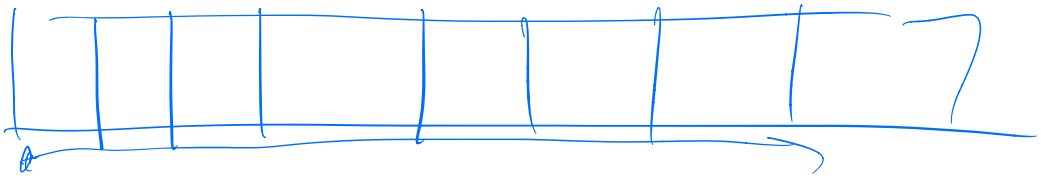
$n \; O\left(\log \frac{M}{m}\right) * n$

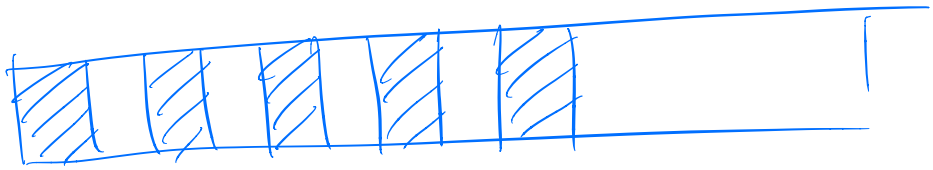
M — biggest

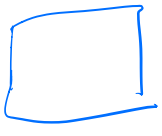
m — smallest

worst-case "fragmentation" = mem CONSUMED / mem requested

$O(M \times m)$ best fit







Compaction

NOT IN C/C++

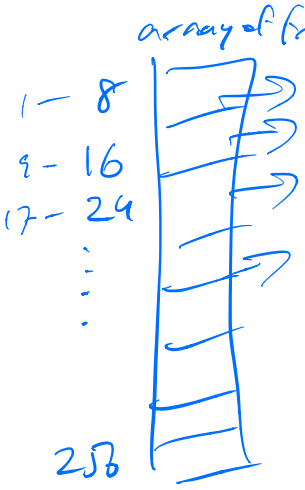
(LIST)

Python, Ruby

"custom allocators"
"ad hoc allocators"

malloc — "too slow"

Recognizing Custom Mem. Allocation



Py_Object_malloc

```
if (sz <= 256) {
    use small obj.
} else
    use malloc
```

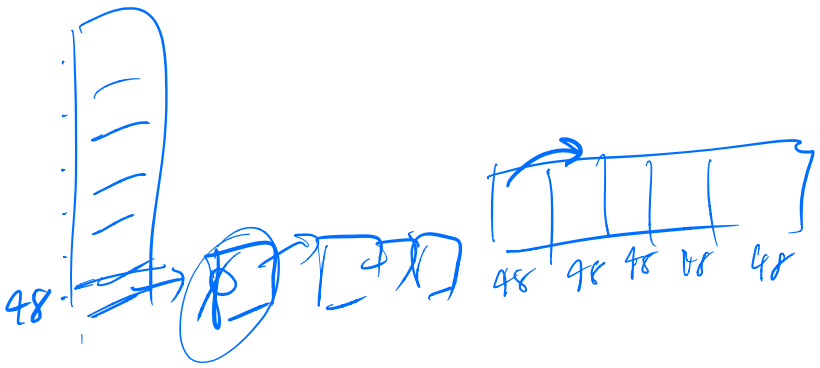
		C	Python
int	≈ 32 bits	4 b	24/28
long	≈ 64 bits	8 b	

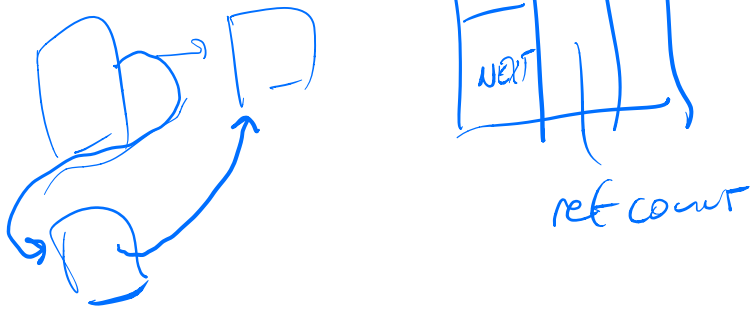
"a" [a][0] NUL 1/16 50

~~{"a": 2}~~

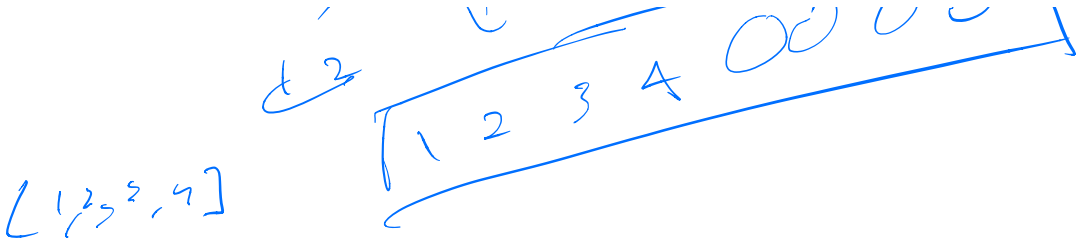
240

60 20 75 100 140





$x = [1, 2, 3]$

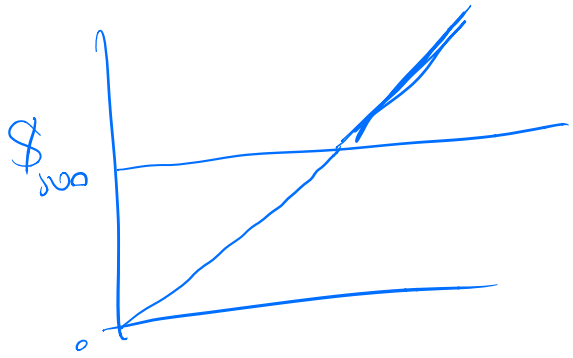


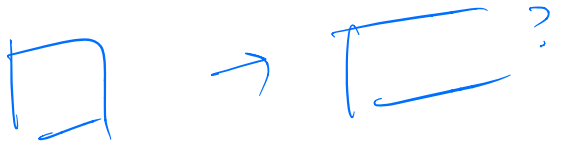
ski rental problem

$50 rent

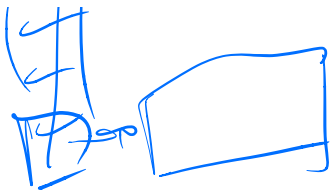
$500 buy

rent until you reach cost of buying



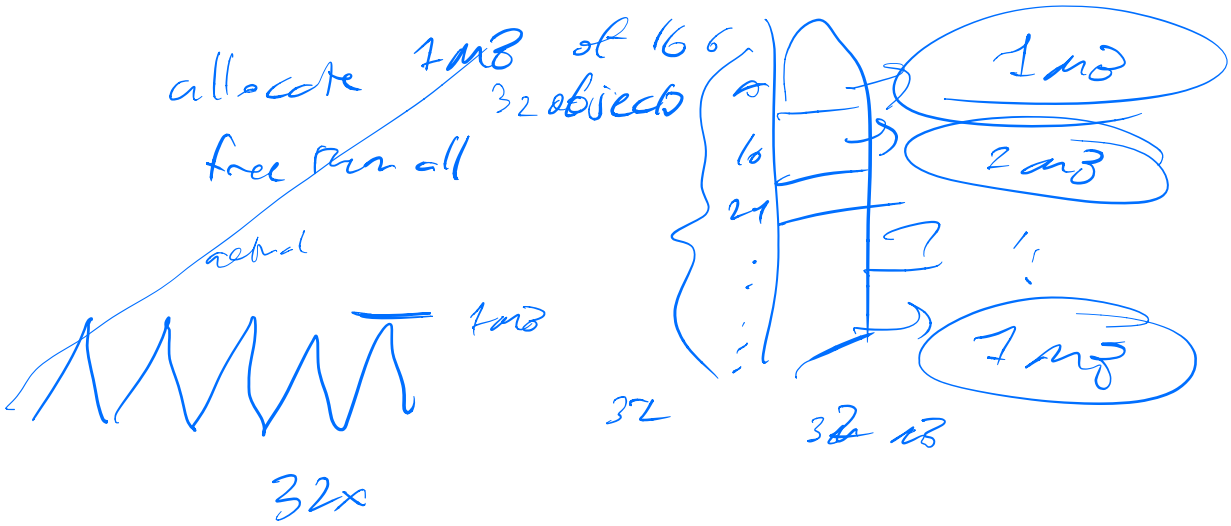


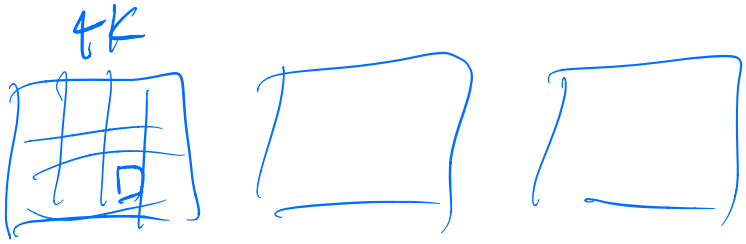
segregated size classes



allocate 1 MB of 8 byte objects
free them all

allocate 1MB of 16 6
32 objects
free them all



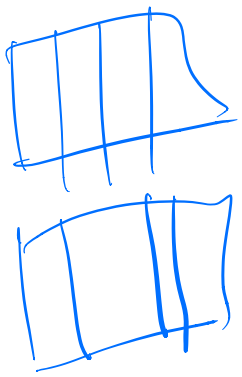


Space-time tradeoff

f(a) =

0	a
1	b
2	c

f(0) = a
f(1) = b
f(2) = c



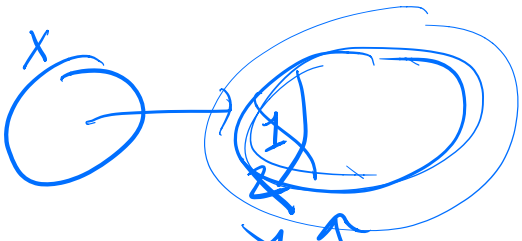
Reference counting

malloc
free

Python Ruby JS Java PHP

Lisp

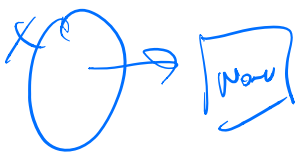
garbage-collected
(no "delete"/free)



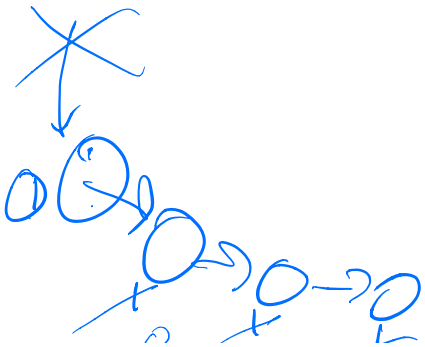
x = []

y = x

x = None



RC = 0 ⇒ safe to reclaim



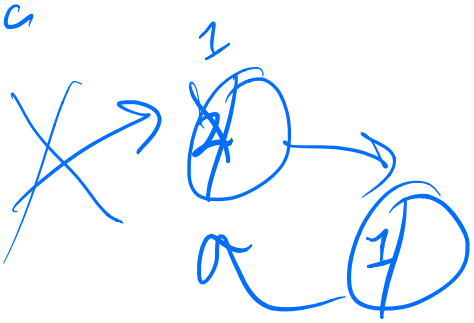
y = None

doubly linked list

a = {'key': 0}

a['key'] = a

CYCLE

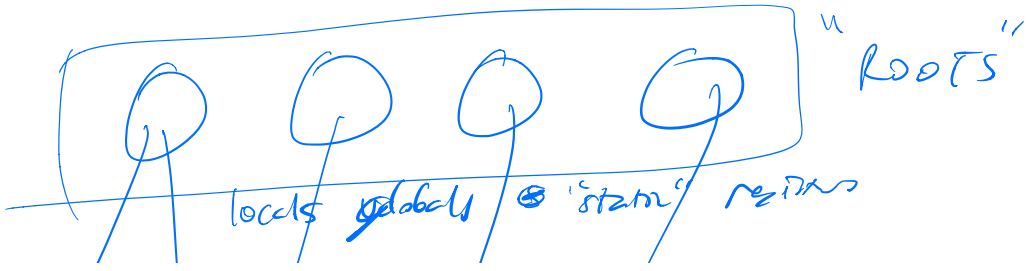


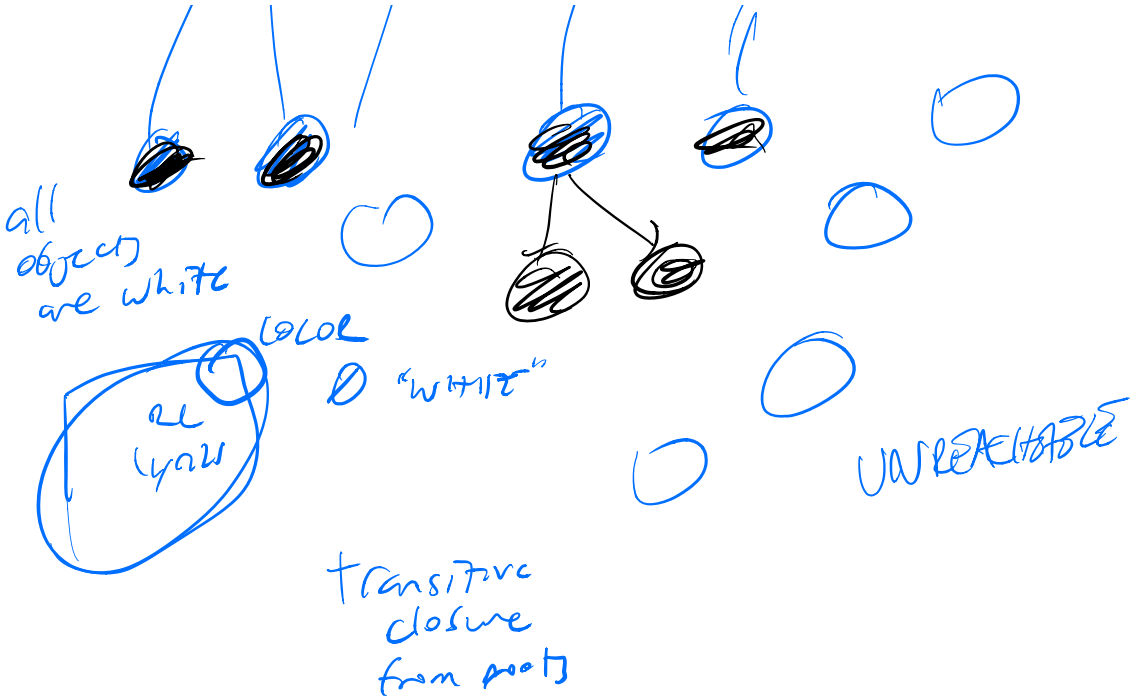
RC: (1958) incomplete GC algorithm

---

mark-sweep (1958)

"generational" GC (1983?)





David Ungar
Smalltalk

self

Ole Agesen

Urs Hölzle
Craig Chambers

prototype oriented

dynamic

GC algorithms

JIT compilation

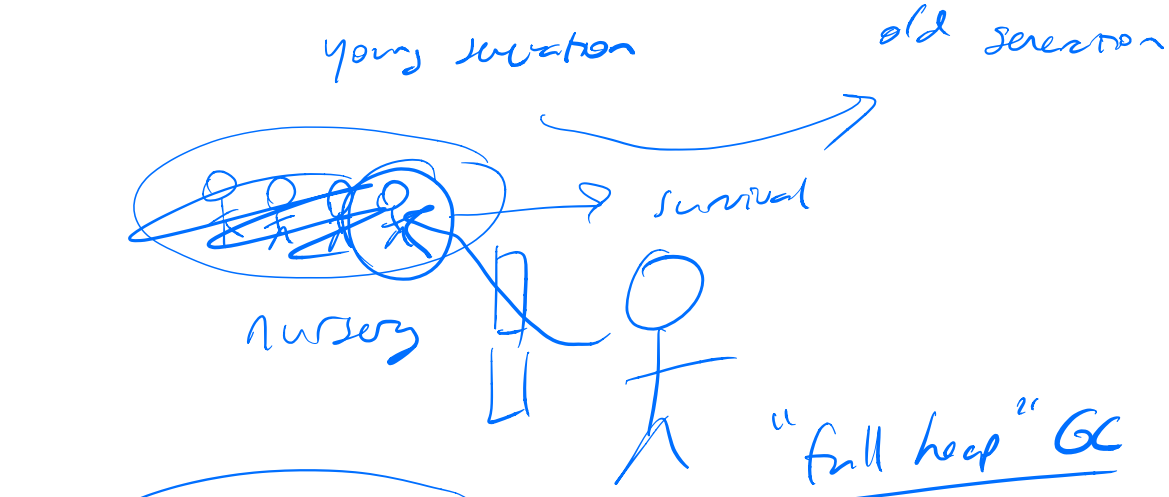
dynamic languages

Lars Bak

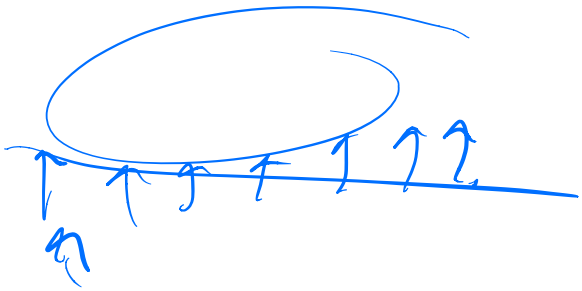
polymorphic inline cache
"PIC"

Google → V8

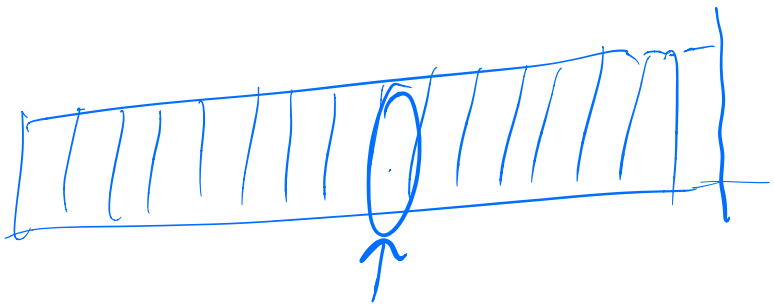
most objects die young

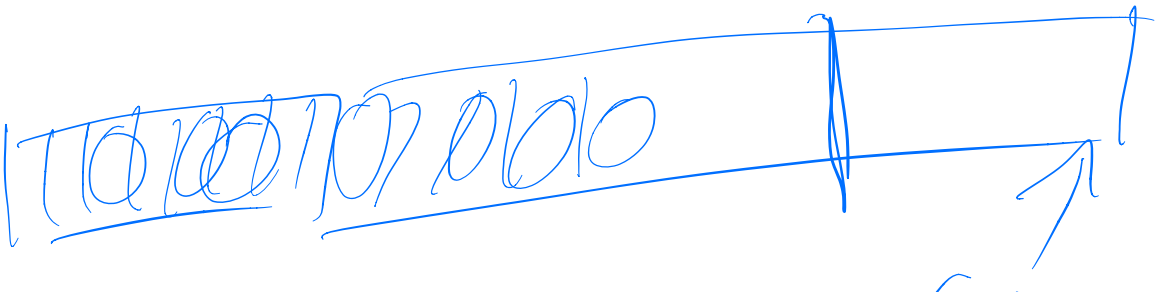


"full heap" GC
mark-sweep



gen GC →
takes pressure off the full heap GC



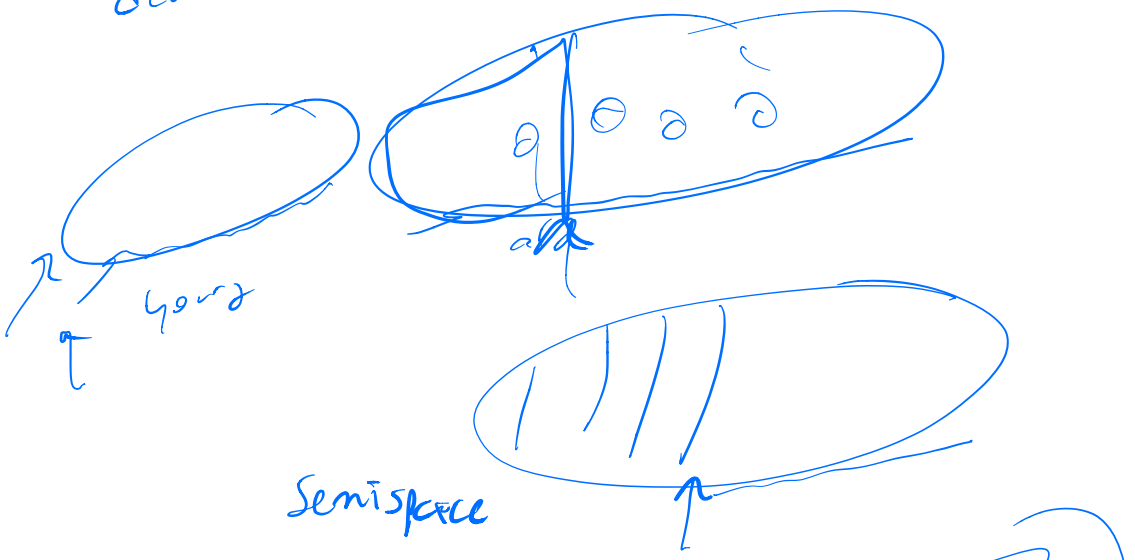


GC

~ 2 ⇒ 5x

Quantifying the cost of GC vs. Explicit dem Mgmt

"dead" objects



immortal

```
for i in range(n)
  for j in range(n) pretending
    for k in range(n)
      C[i][j] += A[i][k] * B[k][j]
```

- .pyc

PyPy