Overview

1. Programming Languages (PL)

4. Program Analysis (PA)

3. Optimization

2. Runtime Environments

PL Categories

lambda calculus (function definition, application, recursion)

- correctness! (efficient?)
- no state (no assignment) instead of loops!
- no side effects
- good for parallelization

ML, Haskell, LISP, ...

Functional Programming Languages

stack
heap (for closures)
↳ garbage collection

declarative

logic formulae (eg. Horn logic)

- correctness!
- logical interpretation
- proof search = execution

not just procedural

Logic Programming Languages — Prolog

theorem prover
resolution
unification

polymorphism (types, procedures)

C++, Java, Smalltalk, ...

Object-Oriented Programming Languages

- correctness!
- abstraction
- encapsulation

inheritance
dynamic binding

concurrent, parallel, distributed

- correctness!
- shared memory
- message passing

Concurrent Programming Languages — Go, Occam, ...

communication
coordination

} GC helps!

Dynamic Heap Management

slow vs. fast, external fragmentation, internal fragmentation

partitioning

random deallocation order

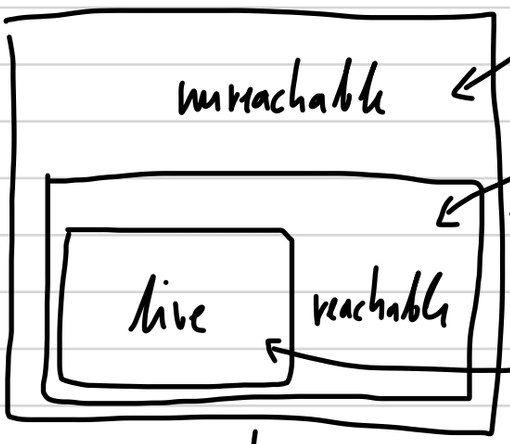
allocate, deallocate contiguous memory chunks

Memory Allocators

- Stack
- Buddy
- Doug Lea
- TLSF
- Compact-fit } real-time
- ...
- Short-term Memory
- ...

coalescing
compaction

- model memory depends on program input
- malloc, free
- dangling pointers
- memory leaks



deallocate

via roots:
- globals
- locals, params → stack

GC approximates object liveness through reachability (or unreachability)

Garbage Collectors

- Mark-Sweep
- Semi-Space
- Generational (Nursery, Mature Space)
- ...
- Metronome
- ...

tracing
reference counting

- safety (dangling pointers)
- memory leaks
- O(heap size)
- reachable memory leaks still possible
- latency → stop the world!

concurrent, incremental → real-time

Optimization Classes

	compiler	output
time	compilation performance	execution performance
space	code size data size	code size data size

→ no global optimum because goals are contradictory

Optimization classes:

1. modification of source code (not considered here)
2. modification of generated code:
   - target-independent
   - target-dependent
   - "context-free"

   } possible with simple-pass compiler
   - "context-sensitive" (requires internal representation of source code)

     ↳ multi-pass compiler

Single Pass

1. target-independent:

```
x + 0 == x
x * 2 == x + x == x << 1
b && true == b || false == b
b || !b == true
...
```

2. target-dependent:

- RISC vs. CISC (instructions combining multiple RISC instructions)
- ARM vs. Intel

3. "context-free":

- constant folding
- multiplication by a power of 2 can be replaced by a left shift operation:

```
x * 2^k == x << k
```

- division by a power of 2 can be replaced by a right shift operation:

```
x / 2^k == x >> k
```

...

Context

1. elimination of common subexpressions:

```
x = (a + b) / c;
y = (a + b) / d;
```

↓

```
u = a + b;
x = u / c;
y = u / d;
```

~> optimizes # of arithmetic operations but not # of assignments

2. array addressing and hidden common subexpressions:

```
a[i][j] = a[i][j] + b[i][j];
```

distance

~> same multiplication done three times $(i \cdot size(array), j \cdot size(element))$

~> same address computation for contiguously allocated arrays

$(i \cdot size(array) + j \cdot size(element))$

3. loop invariants:

~> code motion

```
while (i < 10) {
  z = x + y;
  i = i + 1;
}
```

```
while (i < 10) {
  a[i] = b[i] + c[i];
  i = i + 1;
}
```

4. recurrence relations:

eliminate

$address(a[i]) = address(a) + i \cdot size(element)$

~> $address(a[i+1]) = address(a[i]) + size(element)$

5. register allocation:

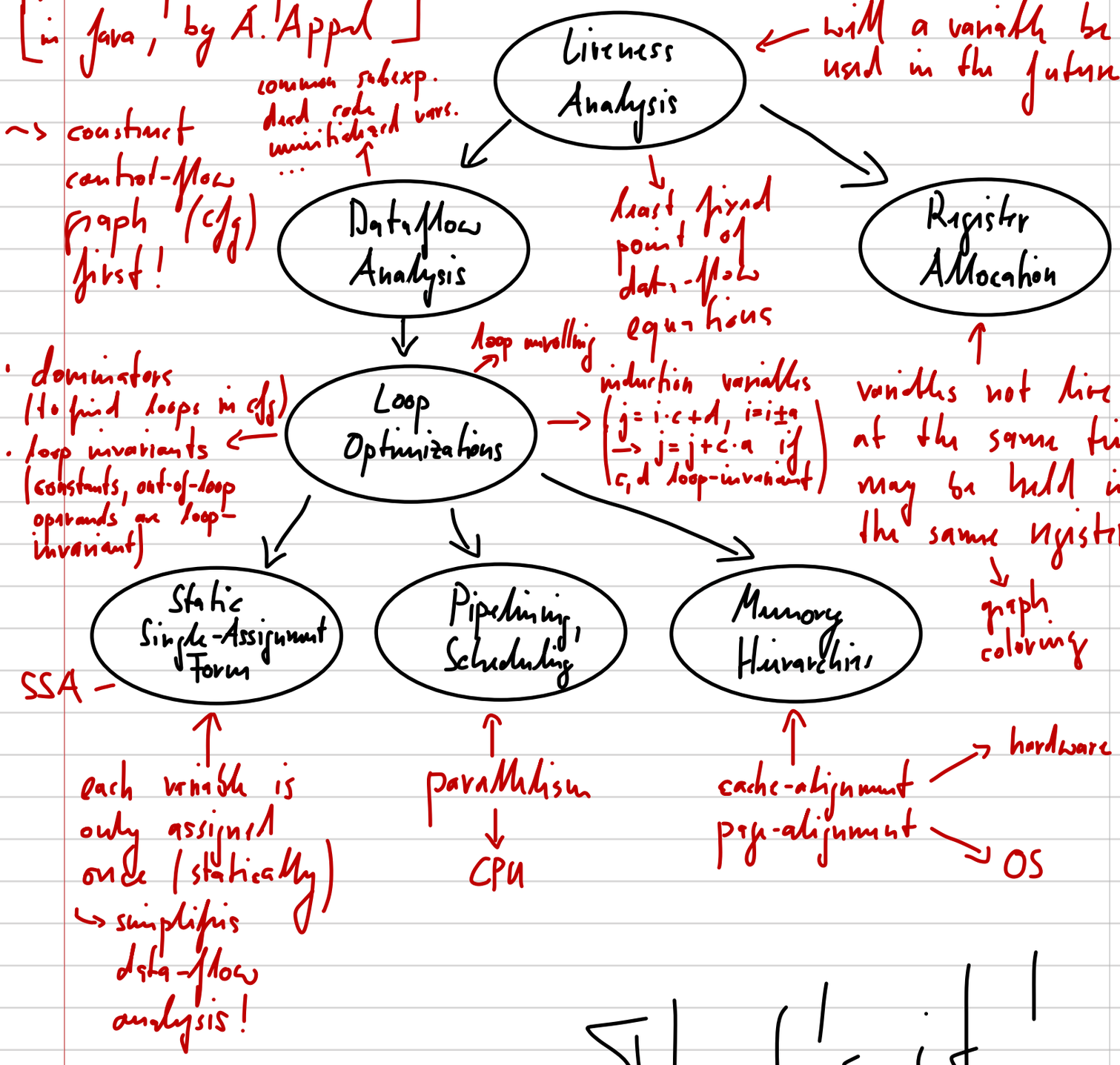
~> using registers only for intermediate results is wasteful

~> use registers also for variables: graph coloring

(too few registers result in spilling)

PA and Optimization

[Modern Compiler Implementation in Java, by A. Appel]



That's it!

Thanks a lot!