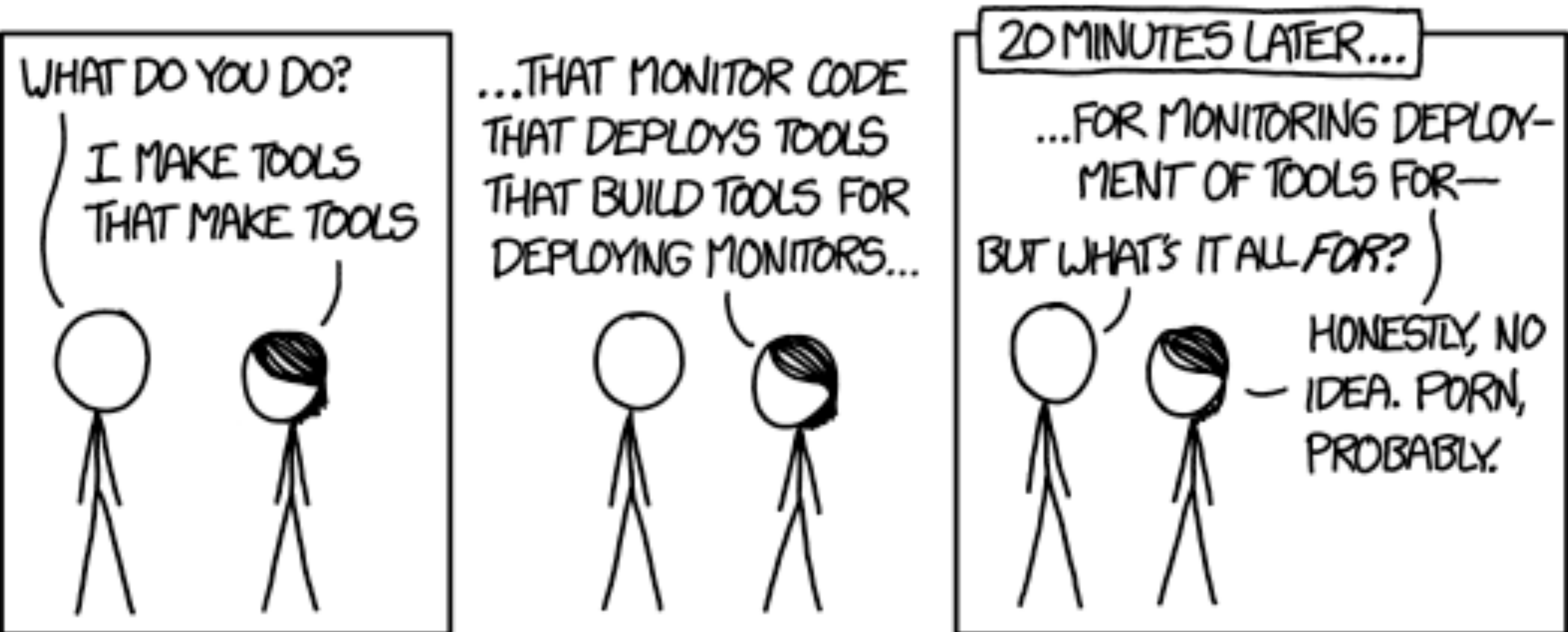
COMPILER DESIGN

AND IMPLEMENTATION

Árpád Goretity
Budapest Swift Meetup, 2016

PART 5

OPTIMIZATION


WHAT DO YOU DO?
I MAKE TOOLS THAT MAKE TOOLS
...THAT MONITOR CODE THAT DEPLOYS TOOLS THAT BUILD TOOLS FOR DEPLOYING MONITORS...
20 MINUTES LATER...
...FOR MONITORING DEPLOY-MENT OF TOOLS FOR—
BUT WHAT'S IT ALL *FOR*?
HONESTLY, NO IDEA. PORN, PROBABLY.

What is optimization?

- Making a program more efficient
  - for some definition of "efficient"
  - usually running time, memory usage, code size
- Not exclusive to compilers!
  - you still have to learn if/when, what, how to optimize manually

Compilers vs Zombies*

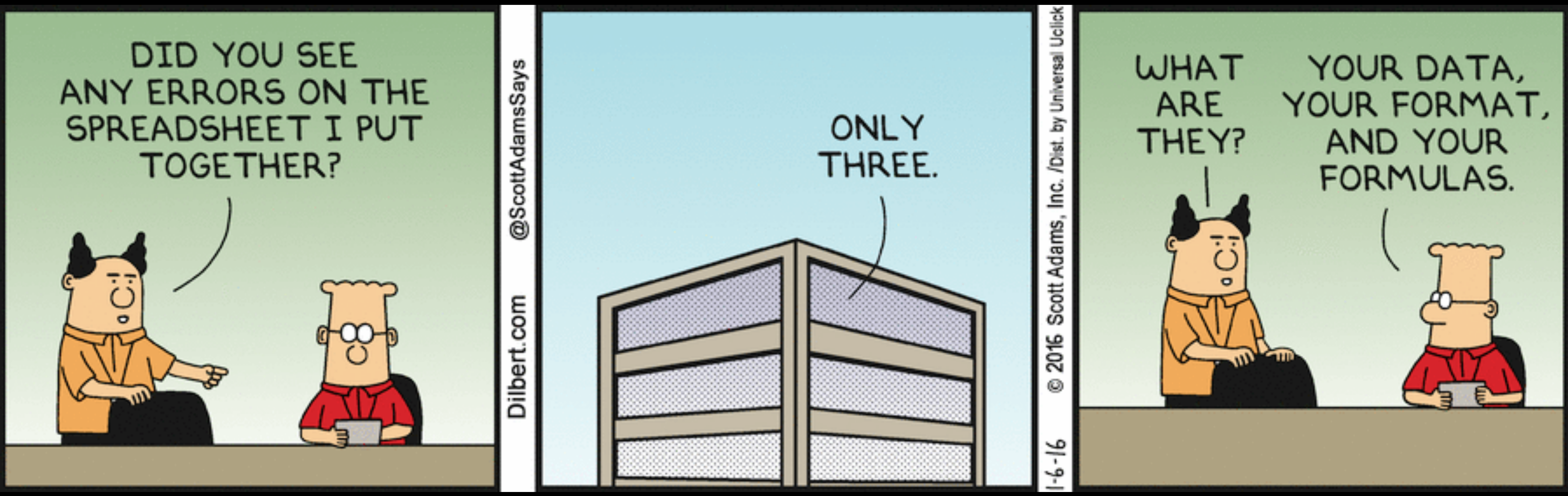
- * Programmers
- Humans perform algorithmic (asymptotic) improvements better – less repetitive, higher-level
- Compilers typically focus on constant factors: less intelligence required, but more tedious
- Nowadays, software requires both!

Why Compiler Opts?

- Good for Architecture and Maintainability of…
  - …the **compiler**: makes front-end simpler, allows it to emit naïve IR (Canonicalization)
  - …**user programs**: no manual squeezing of clock cycles — more functions, readable operations on constants, etc

Compiler Optimization

- Program Analysis + Transformation
- Analysis = Correctness + Viability
  - Correctness: will it preserve **observable** behavior?
  - Viability: will it improve performance?
- Analysis often starts by simple pattern matching


DID YOU SEE ANY ERRORS ON THE SPREADSHEET I PUT TOGETHER?
@ScottAdamsSays
Dilbert.com
ONLY THREE.
© 2016 Scott Adams, Inc. /Dist. by Universal Uclick
1-6-16
WHAT ARE THEY?
YOUR DATA, YOUR FORMAT, AND YOUR FORMULAS.

Compiler Optimization

- Transformation = replacement of instructions with other, equivalent instructions
  - visible behavior (output) + state (memory) must be invariant under the transformation
  - new set of instructions should "work better"

Emergence

- Compiler Optimizations interact intricately: combination is more powerful than individual ones
- Order of Application matters => Hard Problem™
  - non-commutative, non-associative
- Some optimizations can act as canonicalization
  - lay the ground for other stages

Classification

- Abstraction: High-level vs Low-level
- Scope: Local, Global, Inter-Procedural
- Approach: Simplification, Redundancy removal, …
- Language Construct: Data flow, Control flow, …
- Not always clear — but it's not the point…

pause to catch breath

Strength Reduction

- Constant folding and propagation
  - performing computation at compilation time
  - e.g.: Arithmetic instructions
    - Algebraic identities: `x + 0`, `x * 1`, …
- Multiplication and division by 2^N => bit shifts

Strength Reduction

- Function Inlining
- Tail-Call Optimization: tail recursion => loop
- Devirtualization
- Scalar Replacement of Aggregates (for RegAlloc, LICM, …)
- Semantic optimization of high-level types
  - `@semantic("array.count")`
  - ARC Optimization

Devirtualization

```
class Base {
  func foo() { … }
}

class Derived : Base {
  override func foo() { … }
}

let obj: Base = func_returning_derived()
obj.foo()
```

Devirtualization

```
func Base_foo(self: Base) { … }
func Derived_foo(self: Derived) { … }

let Base_vtable = [ Base_foo ]
let Derived_vtable = [ Derived_foo ]
let foo_index = 0

let obj: Base = func_returning_derived()
obj.vtable[foo_index](obj)
```

Devirtualization

Can we do better?

Devirtualization

```
let obj: Base = func_returning_derived()

if obj.vtable == Base_vtable {
  Base_foo(obj)
} else if obj.vtable == Derived_vtable {
  Derived_foo(obj)
} else {
  obj.vtable[foo_index](obj)
}
```

Devirtualization

Even better…?

Devirtualization

```
let obj: Base = func_returning_derived()

if obj.vtable == Base_vtable {
  // stuff Base.foo() does, inlined
} else if obj.vtable == Derived_vtable {
  // stuff Derived.foo() does, inlined
} else {
  obj.vtable[foo_index](obj)
}
```

Removal of Redundancy

- Common Subexpression Elimination
- Copy Propagation
- Elimination of redundant deep copies and moves
  - e.g.: RVO, NRVO in C++
- Elimination of Runtime Checks
  (ABC in a loop, overflow check on constants)

Data Flow, Control Flow

- Many of them fall straight out of SSA
  - Dead Code Elimination
  - Dead Store Elimination
- Loop-Invariant Code Motion
- Loop Unrolling, Loop Unswitching
- Loop Interchange

Loop Interchange

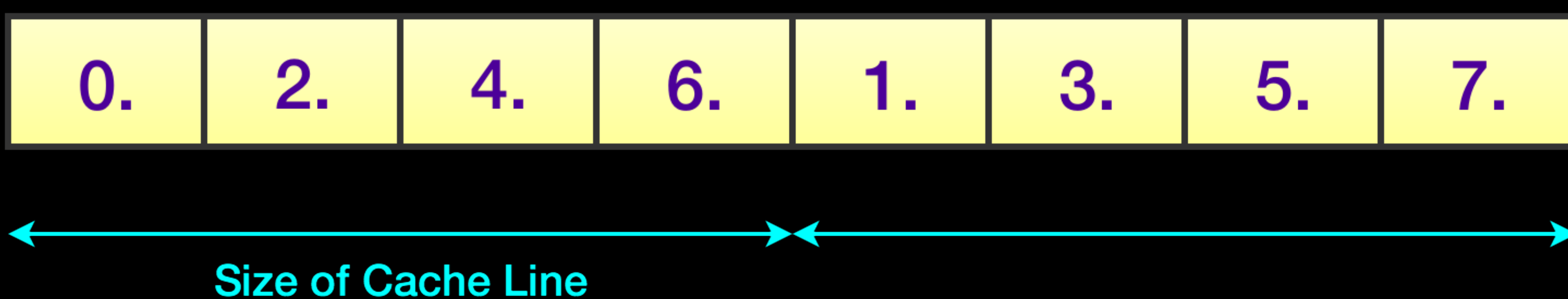
Ensures better cache locality of $\geq$ 2D arrays

Loop Interchange

```
for i = 0; i < COLS; i++ {
  for j = 0; j < ROWS; j++ {
    arr[j][i] += 1;
  }
}
```

Loop Interchange

```
for i = 0; i < COLS; i++ {
  for j = 0; j < ROWS; j++ {
    arr[j][i] += 1;
  }
}
```

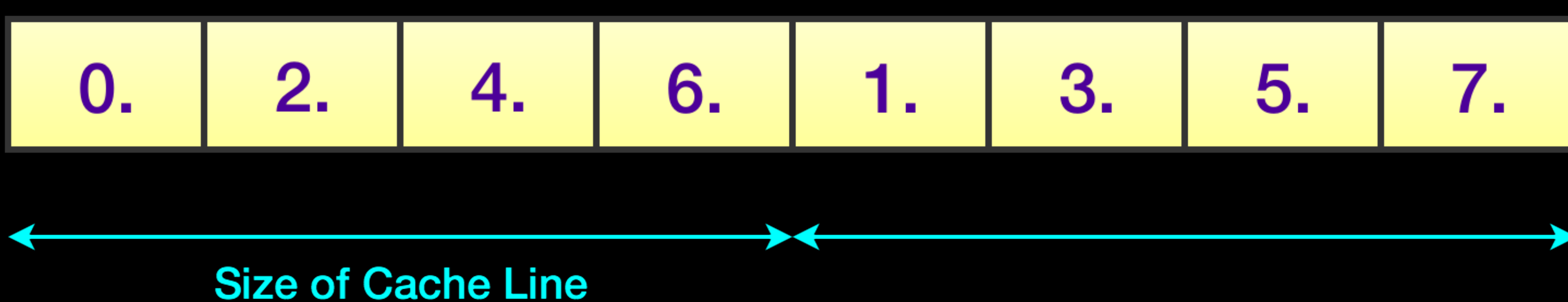


Loop Interchange

A Cache Miss upon every iteration!

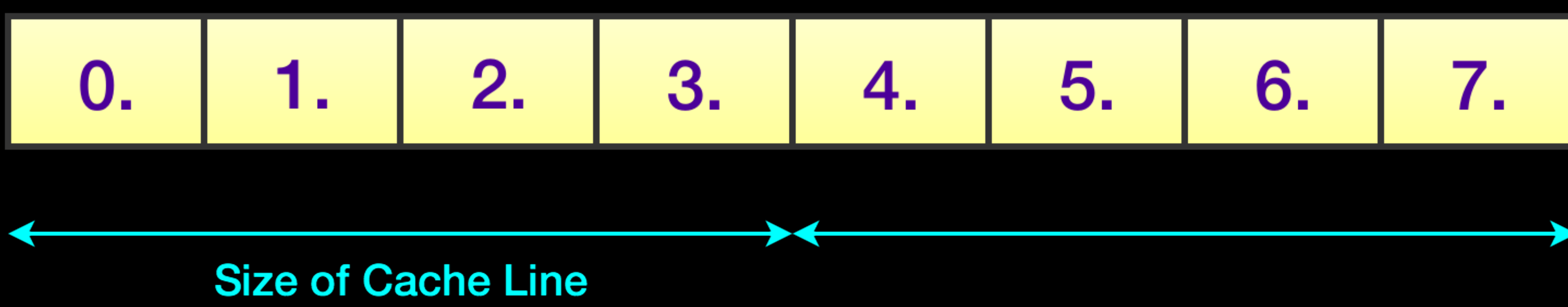
Loop Interchange

```
for i = 0; i < COLS; i++ {
  for j = 0; j < ROWS; j++ {
    arr[j][i] += 1;
  }
}
```



Loop Interchange

```
for j = 0; j < ROWS; j++ {
  for i = 0; i < COLS; i++ {
    arr[j][i] += 1;
  }
}
```



pause to catch breath

Low-level Optimizations

- More like "non-pessimizations" in code generation
- Instruction Selection **(CISC!)**
- Vectorization
- Instruction Scheduling
- Register Allocation
- Peephole Optimizations

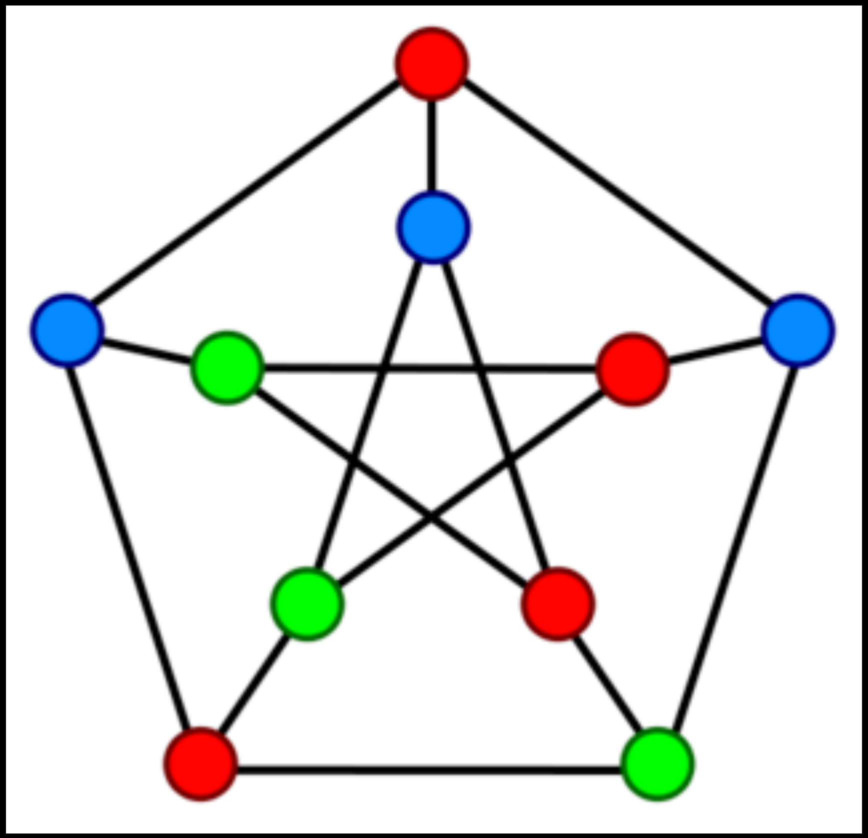
Register Allocation

- So You Wanted to Know What You Will Use Math For in Programming
- Graph theory!
- Goal: squeeze as many hot (frequently used) variables or temporaries into registers as possible
- a.k.a. minimize spilling to stack

Register Allocation

- The Graph Coloring problem
- Given a graph, color each vertex so that no two adjacent vertices have the same color
- Use the minimal number of colors

Register Allocation

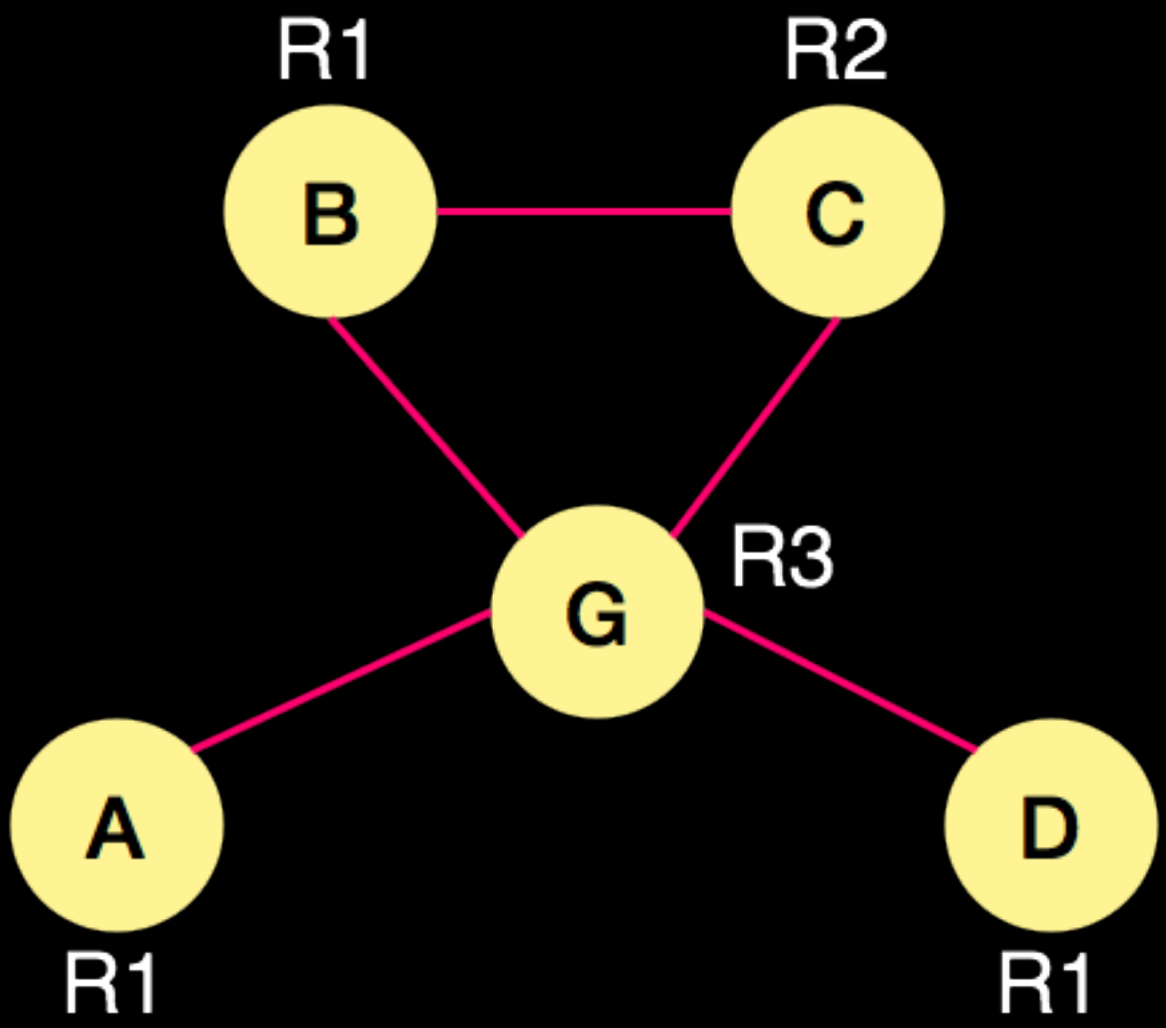


Register Allocation

- Vertices <=> Values (variables, temporaries)
- Edges: Adjacent Vertices <=> lifetimes overlap
- Colors <=> Registers
- No two parallelly-alive values may share a register
  - but others can, and probably should

Register Allocation

```
let g = 1
{
  let a = 2
  print(a)
  let b = 3
  let c = 4
  print(b)
  print(c)
}
{
  let d = 5
  print(d)
}
```



Register Allocation

- Graph Coloring is NP-complete in the general case
- $O(2^N \cdot N)$ for N vertices (variables)
- Compilers need to apply heuristics
  - Greedy coloring, starting with the vertex of highest degree (the one with most neighbors)
  - Usually yields a decent result

And many more…

- There are **lots of** optimizations
- Compiler Optimization is an active area of research
- Modern popular compilers, AOT as well as JIT, introduce new and improved optimizations **daily**
- Lots of clever engineering in there – read papers!

Security Concerns

Security Concerns

- What counts as observable behavior?
  - "redundant" overwriting of security-critical memory buffers, and the `volatile` trick
- Languages with UB: bugs "abused" by compiler
- Compiler Bugs – non-equivalent transformations

Security Concerns

- Malice in Compilers: the "Trusting Trust" problem
- Ken Thompson: Reflections on Trusting Trust (1984)
- Compiler Security: an entire subject on its own…

Questions?