Control flow-sensitive optimizations

In the Druid Meta-Compiler

Matías Demare - Guillermo Polito
Javier Pimás - Nahuel Palumbo

matias-nicolas.demare@inria.fr
github.com/m-demare


Conditional branches are slow

Conditional branches are slow… But why?

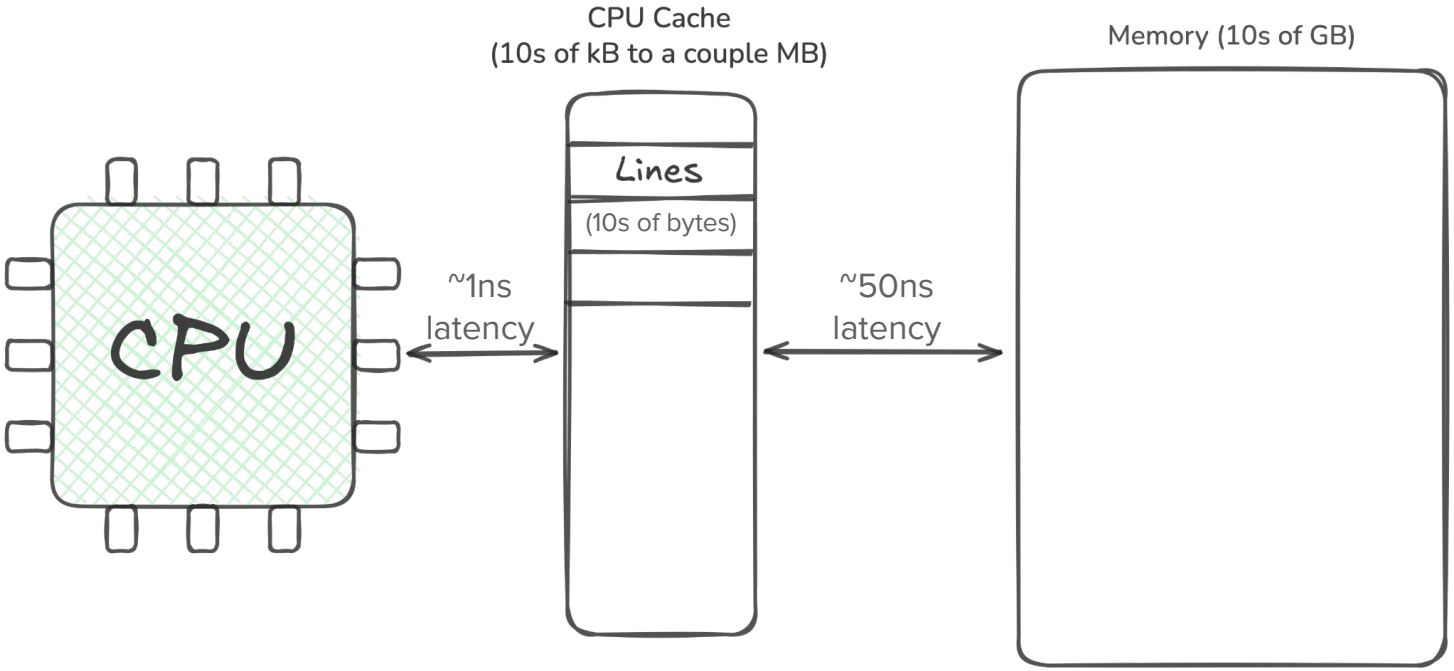
- Complexification of control flow
- Increase in code size
- CPU pipeline stalling

Complexification[1] of control flow

[1] https://english.stackexchange.com/a/607869

- Prevents optimizations
- Makes some compilers' tasks harder
  - Block placement
  - Register allocation

Increase in code size



Has a considerable impact, especially with suboptimal code placement

Pipeline stalling

- Caused by the way modern processors work
- Can have a huge impact

What’s a CPU pipeline?

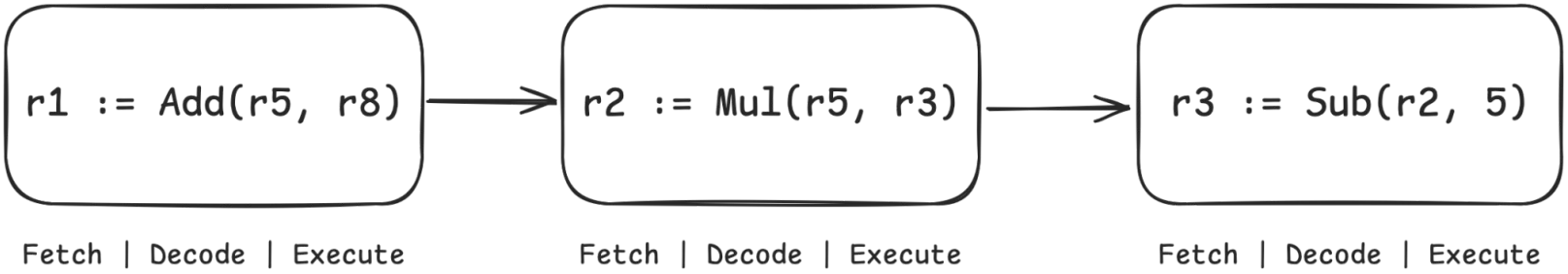
```
r1 := Add(r5, r8)
r2 := Mul(r5, r3)
r3 := Sub(r2, 5)
```

Assuming 3 cycles per instruction

What's a CPU pipeline?

```
r1 := Add(r5, r8)
r2 := Mul(r5, r3)
r3 := Sub(r2, 5)
```

Assuming 3 cycles per instruction → 9 cycles in total



...or does it?

```
r1 := Add(r5, r8)
r2 := Mul(r5, r3)
r3 := Sub(r2, 5)
```

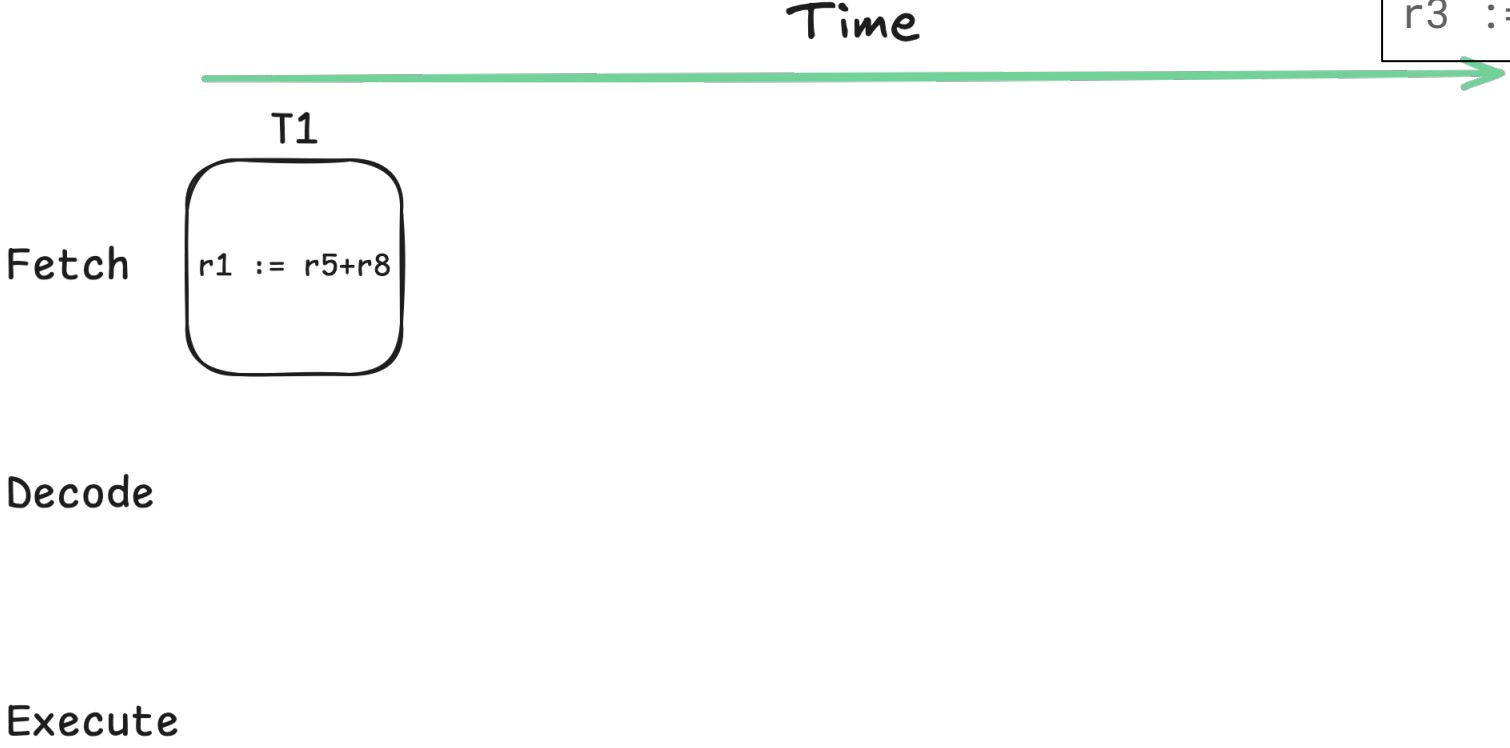
Fetch

Decode

Execute

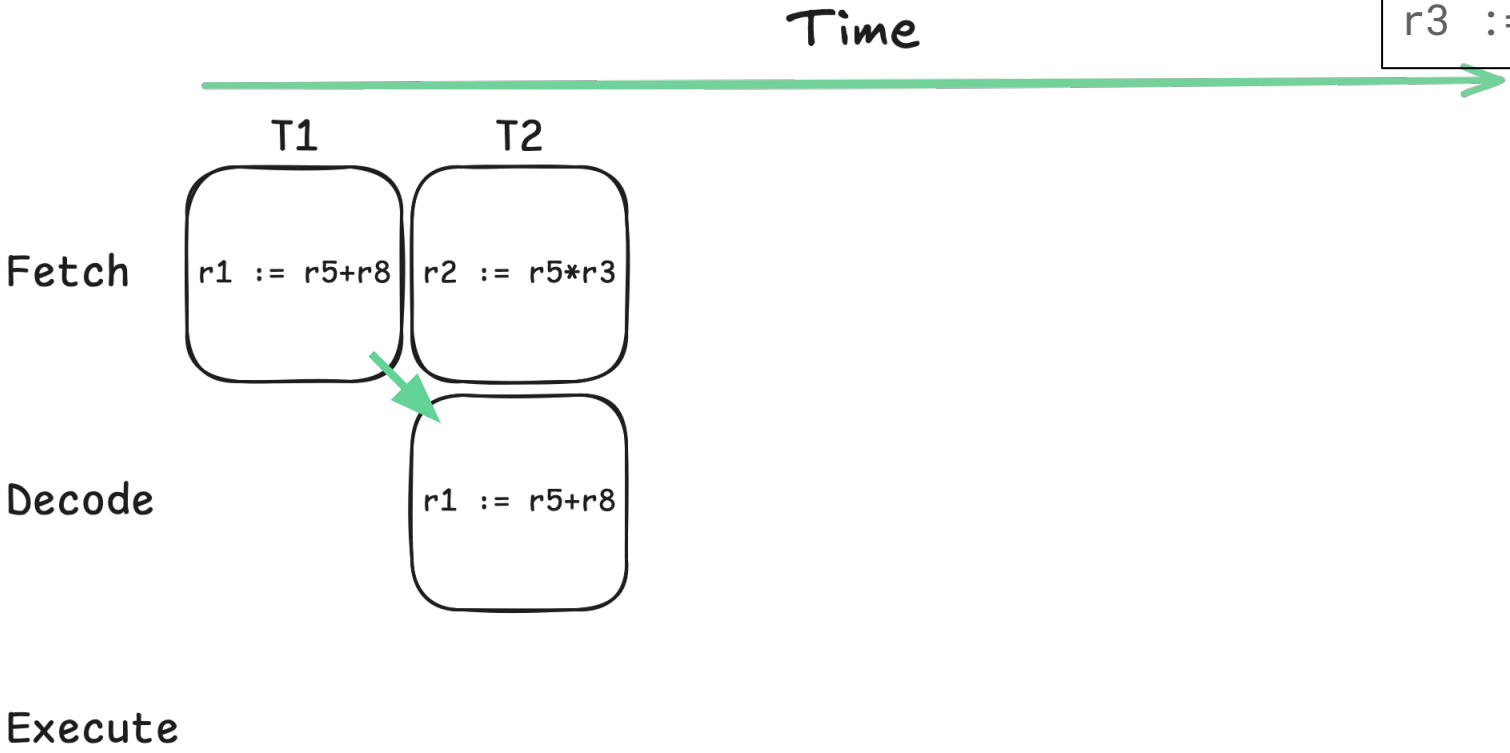
...or does it?

```
r1 := Add(r5, r8)
r2 := Mul(r5, r3)
r3 := Sub(r2, 5)
```



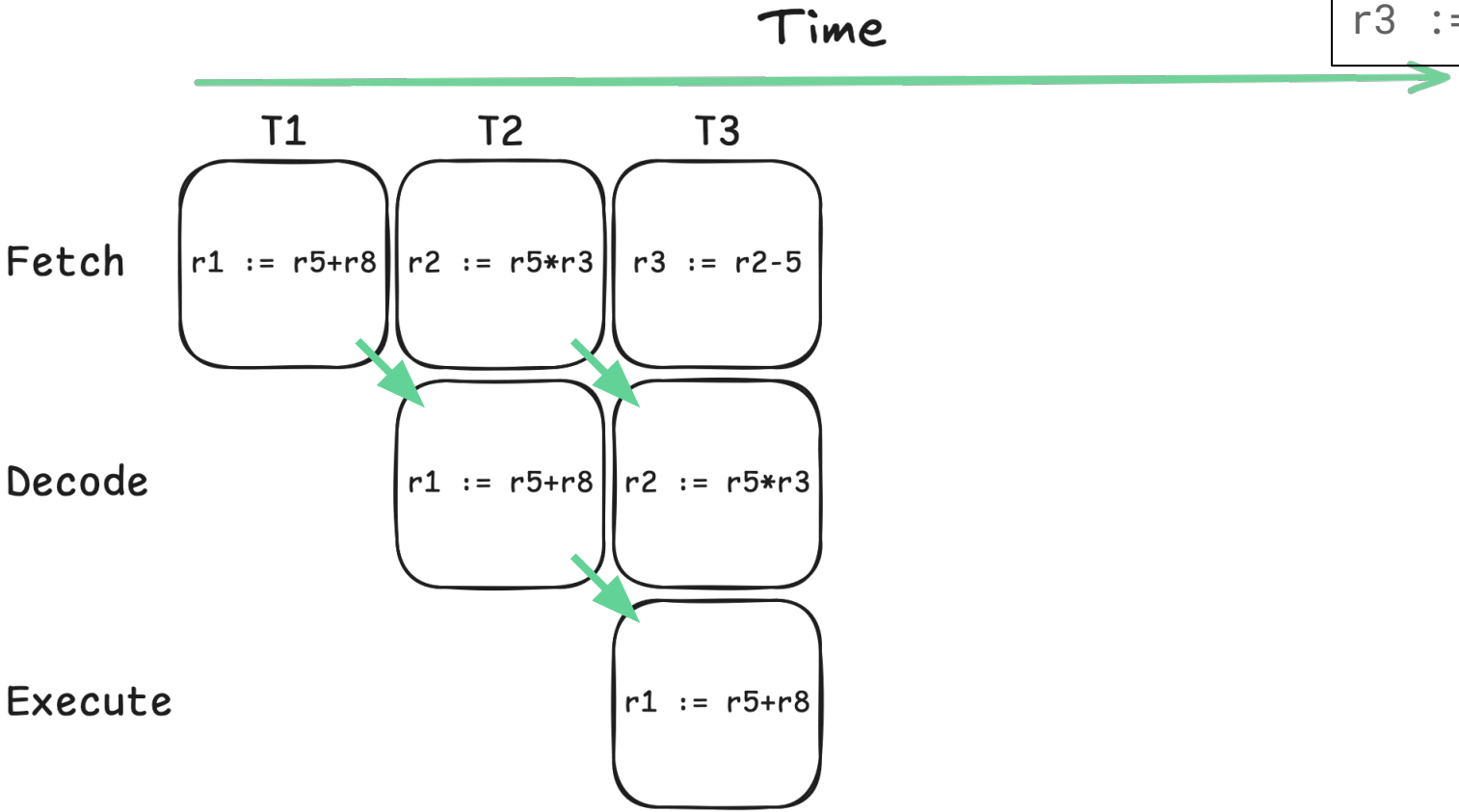
...or does it?

```
r1 := Add(r5, r8)
r2 := Mul(r5, r3)
r3 := Sub(r2, 5)
```



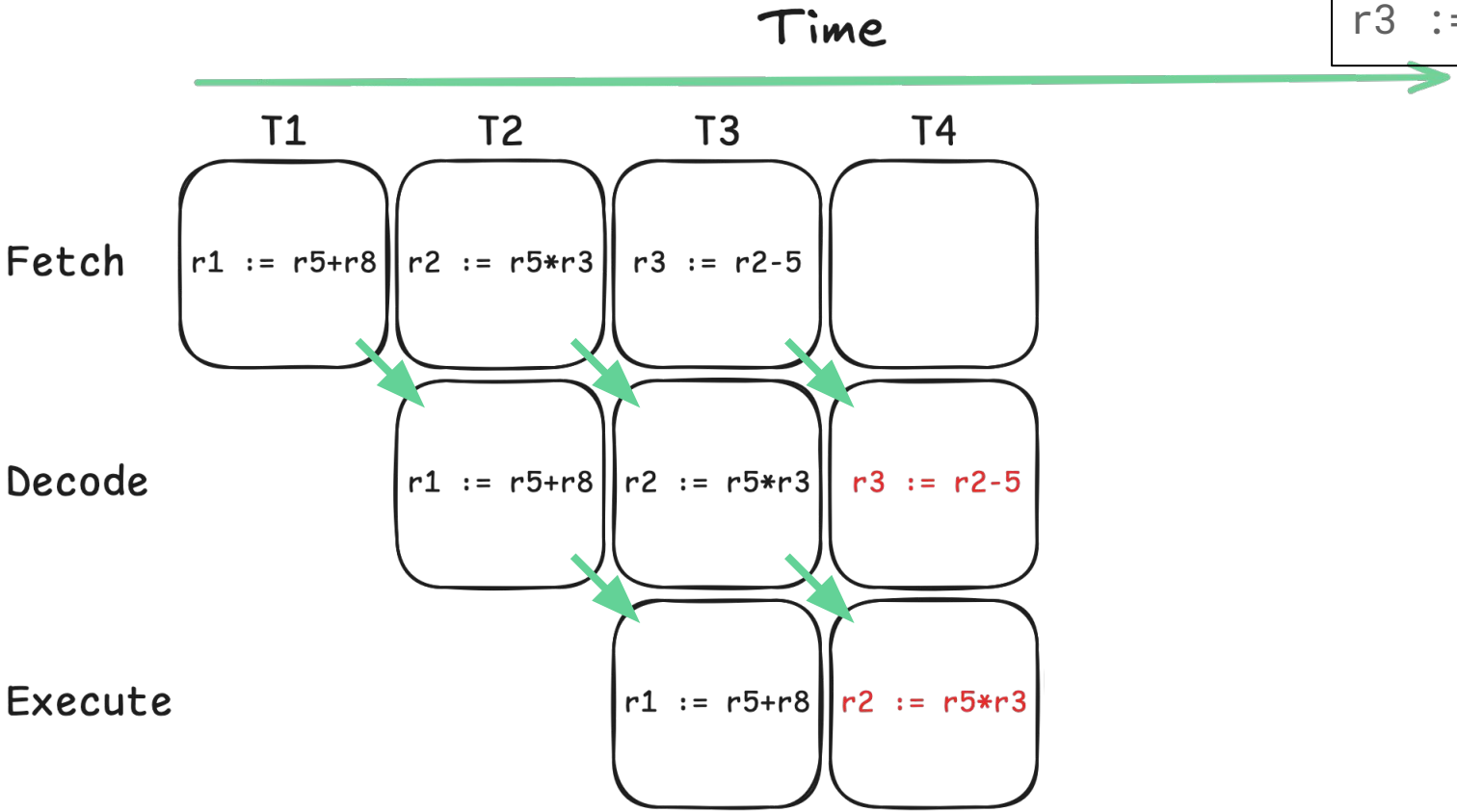
...or does it?

```
r1 := Add(r5, r8)
r2 := Mul(r5, r3)
r3 := Sub(r2, 5)
```



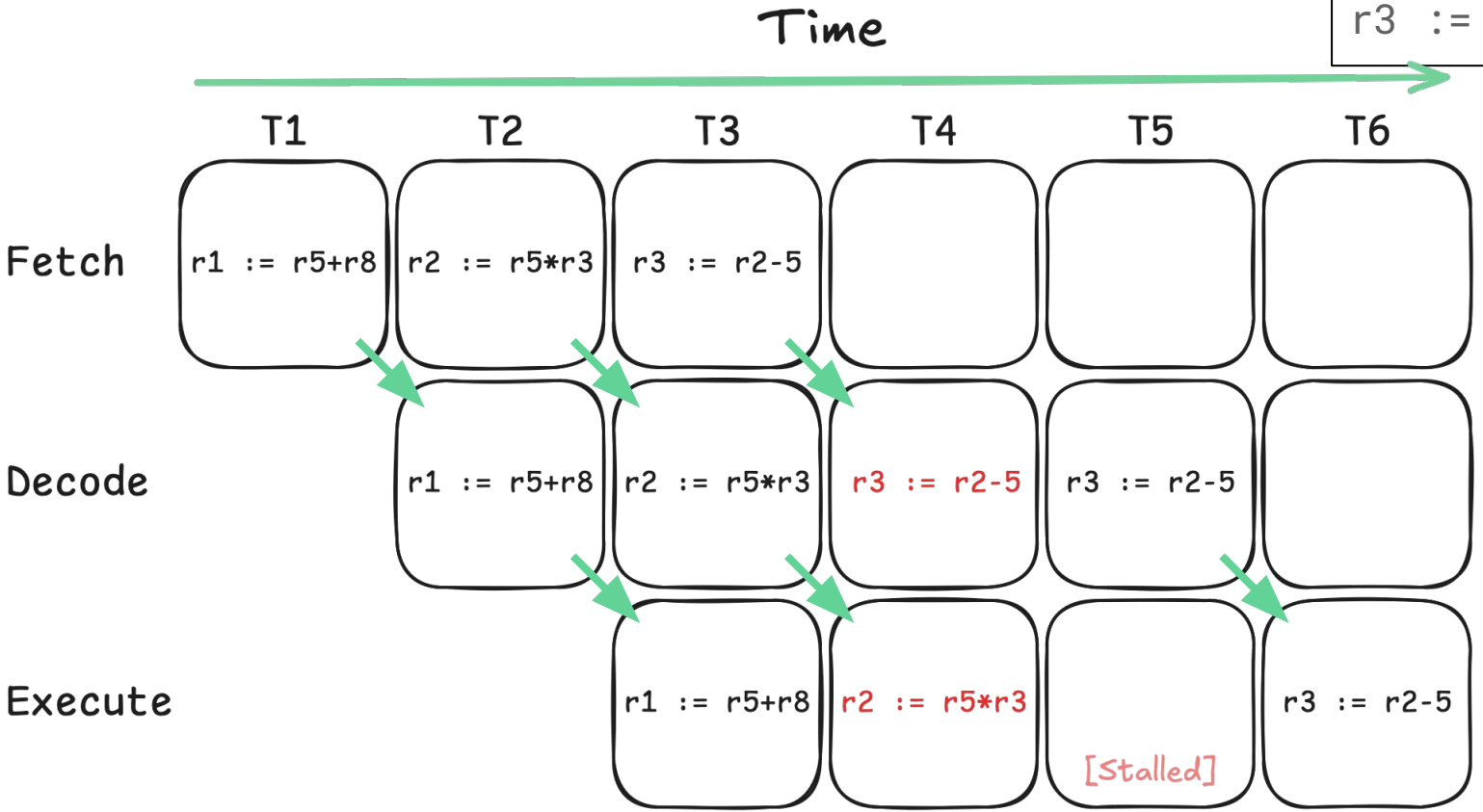
```
r1 := Add(r5, r8)
r2 := Mul(r5, r3)
r3 := Sub(r2, 5)
```

...or does it?



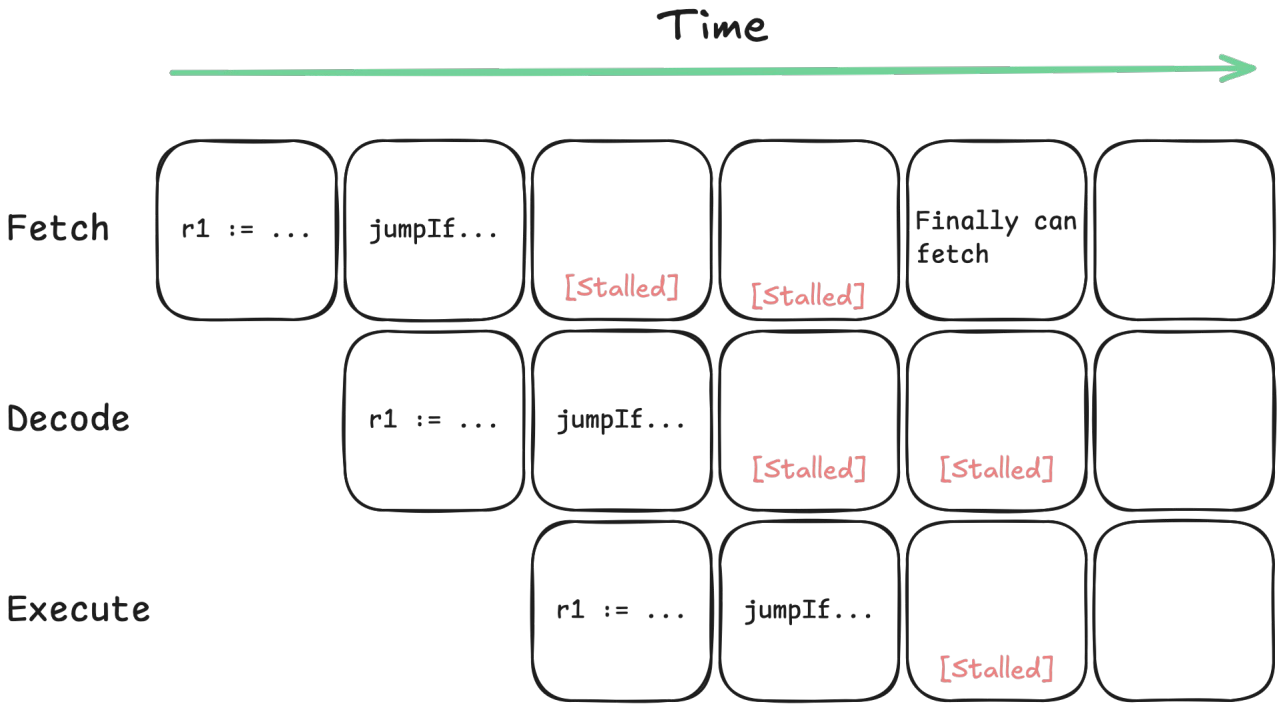
...or does it?

```
r1 := Add(r5, r8)
r2 := Mul(r5, r3)
r3 := Sub(r2, 5)
```



The issue with branches

```
r2 := Add(r5, r8)
jumpIf (r1 > 5)
  to X
[other instructions…]
X: …
```



And it gets worse...

- The deeper the pipeline is, the longer the stall
  - AMD Zen uses 19 stages
  - Intel Lion Cove uses 10 stages
- Other CPU features can make this even more costly (see indirect branches, superscalar microarchitectures, etc)

CPUs try to solve this

- Branch prediction:
  - **Guess** which branch will run, and start executing it
  - Discard results if the guess was wrong
- Eager speculative execution
  - Execute **both** branches simultaneously
  - Discard the results from the one that ends up being “wrong”

Still, the best branch is no branch at all

“But… I don’t write redundant conditionals!!”

Sadly, compilers write them for you

- Function inlining
- Lowering of high-level features. E.g.,
  - Array bounds checks
  - Runtime type checks
  - Polymorphism + message sends

“But… I don’t write redundant conditionals!!”

Sadly, compilers write them for you

- Function inlining
- Lowering of high-level features. E.g.,
  - Array bounds checks
  - Runtime type checks
  - Polymorphism + message sends

```
[ i < array size ] whileTrue: [
    var := array at: i.
    i := i + 1.
]
```

```
Start:
  jumpIf (i >= array size)
    to End
  jumpIf (i >= array size)
    to Error
  var := MemLoad(array + i)
  i := Add(i, 1)
jumpTo Start
End: …
```

Goal: Detect and eliminate **dead branches**

Dead branches: cannot be reached in any execution of the program.

Detecting them implies determining if a given condition is satisfiable **in its context**

```
x < 5 ifTrue: [
    x > 10 ifTrue: [
        “Unreachable code”
    ]]
```

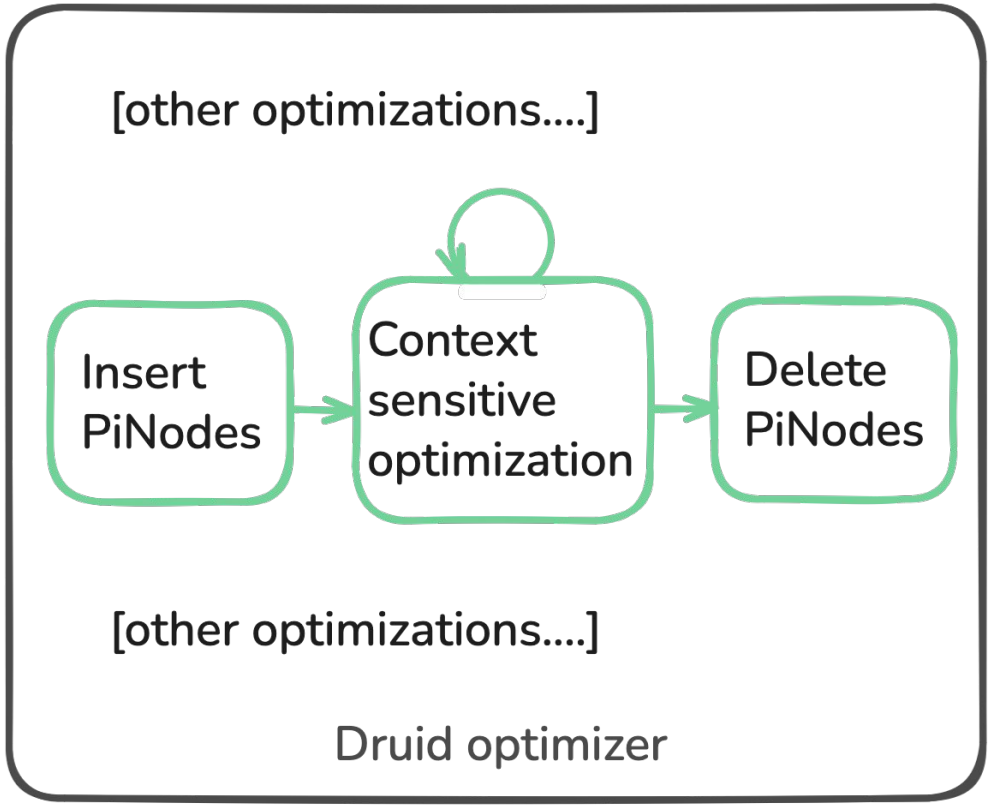
PiNodes: Representing Constraints on Variables

PiNodes: Representing Constraints on Variables

```
x < 5 ifTrue: [
    x > 10 ifTrue: [
        “Unreachable code”
    ]
]
```

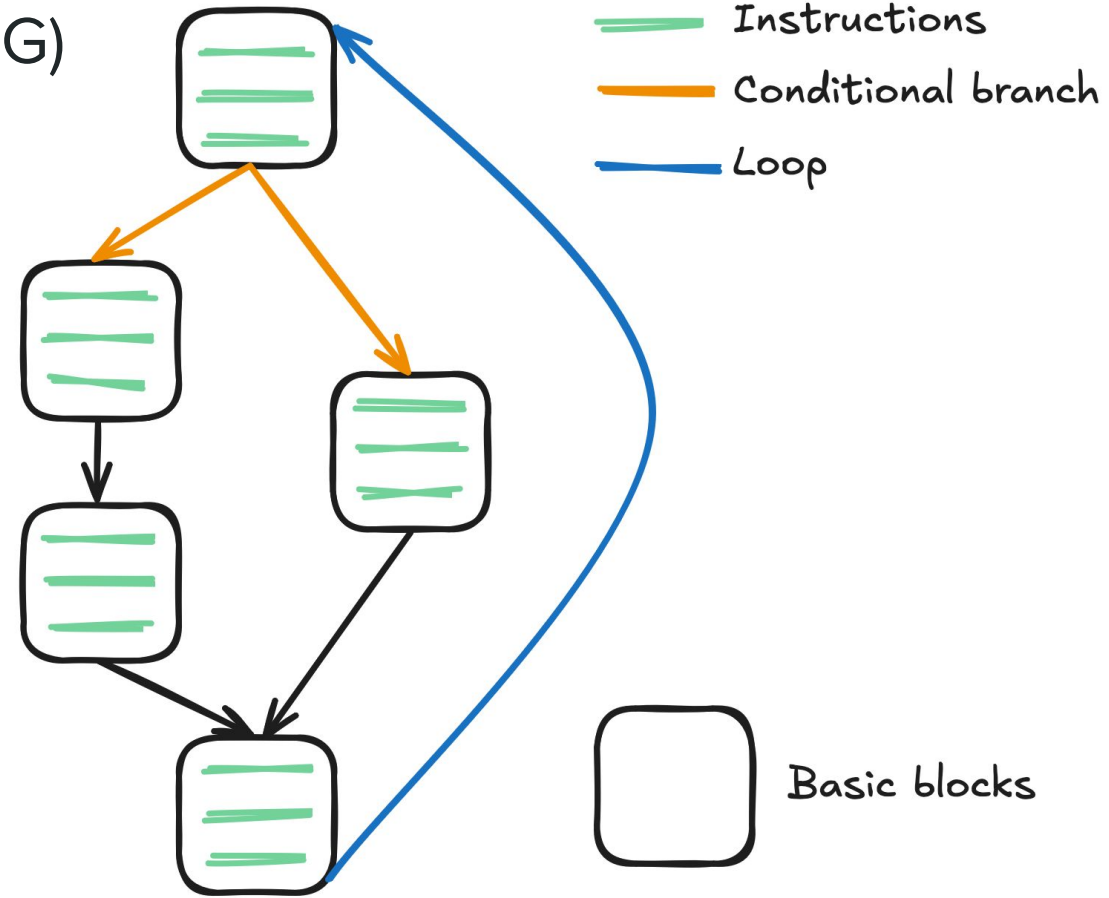
```
x1 < 5 ifTrue: [
    x2 := π(x1, <5)
    x2 > 10 ifTrue: [
        x3 := π(x2, >10)
        “Unreachable code”
    ]
]
```

Optimizing with PiNodes



Control Flow Graph (CFG)

- Graph based representation of a program
- Nodes = **Basic blocks**
- Edges = Jumps



SSA

- Variables are assigned exactly once

```
x := 5.
x := 27.
x < 10 ifTrue: [
    y := x.
] ifFalse: [
    y := 13.
].
z = y + x
```

```
x1 := 5.
x2 := 27.
x2 < 10 ifTrue: [
    y1 := x2.
] ifFalse: [
    y2 := 13.
].

z1 = ?? + x2
```

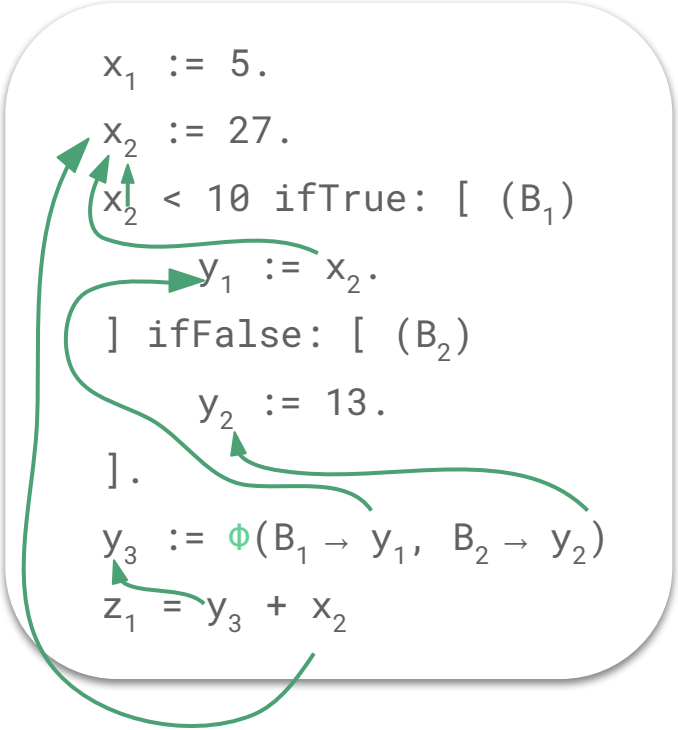
SSA

- Variables are assigned exactly once
- Φ-functions represent variables at merge points

```
x := 5.
x := 27.
x < 10 ifTrue: [
    y := x.
] ifFalse: [
    y := 13.
].
z = y + x
```

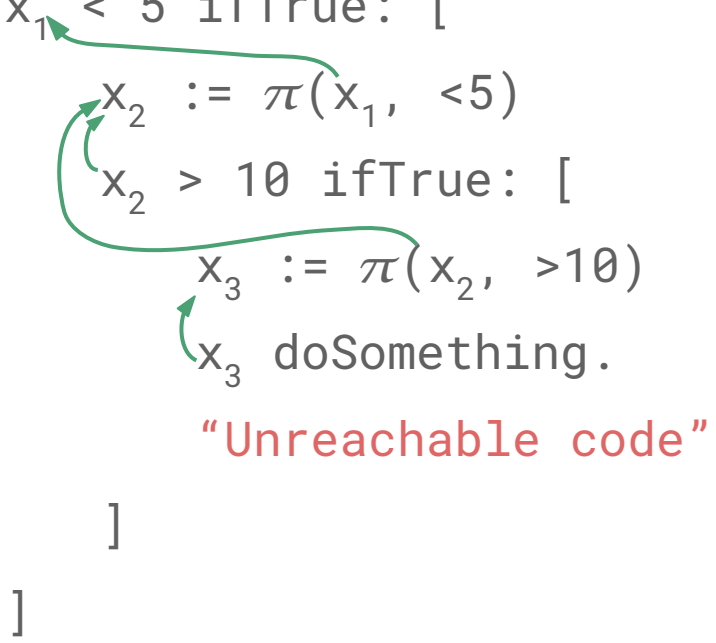
```
x1 := 5.
x2 := 27.
x2 < 10 ifTrue: [ (B1)
    y1 := x2.
] ifFalse: [ (B2)
    y2 := 13.
].
y3 := ϕ(B1 → y1, B2 → y2)
z1 = y3 + x2
```

SSA - use-def chains



PiNodes: representing constraints on variables

```
x < 5 ifTrue: [
    x > 10 ifTrue: [
        x doSomething.
        “Unreachable code”
    ]
]
```



Optimizations - Dead branch elimination

Constant dead branch elimination

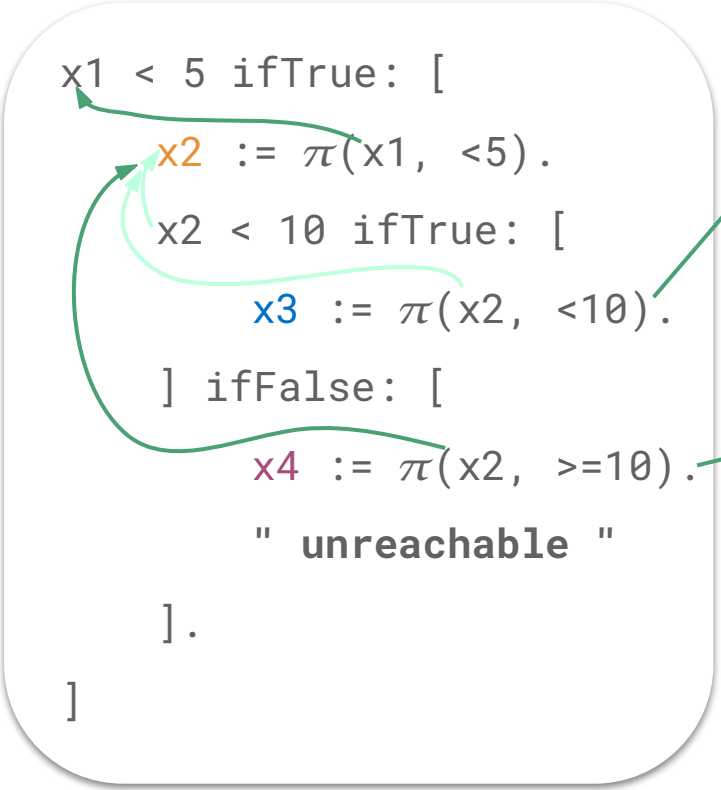
```
x1 < 5 ifTrue: [
    x2 := π(x1, <5).
    x2 < 10 ifTrue: [
        x3 := π(x2, <10).
    ] ifFalse: [
        x4 := π(x2, >=10).
        " unreachable "
    ].
]
```

x3 := π(x2, <10) and x2 := π(x1, <5)

Is (-∞; 5) ∩ (-∞; 10) empty?

NO ⇒ Reachable

Constant dead branch elimination



x3 := π(x2, <10) and x2 := π(x1, <5)

Is (-∞; 5) ∩ (-∞; 10) empty?

NO ⇒ Reachable

x4 := π(x2, >=10) and x2 := π(x1, <5)

Is (-∞; 5) ∩ [10; ∞) empty?

YES ⇒ Unreachable

Constant dead branch elimination

```
x1 < 5 ifTrue: [
    x2 := π(x1, <5).
    x2 < 10 ifTrue: [
        x3 := π(x2, <10).
    ] ifFalse: [
        x4 := π(x2, >=10).
        " unreachable "
    ].
]
```

x3 := π(x2, <10) and x2 := π(x1, <5)

Is (-∞; 5) ∩ (-∞; 10) empty?

NO ⇒ Reachable

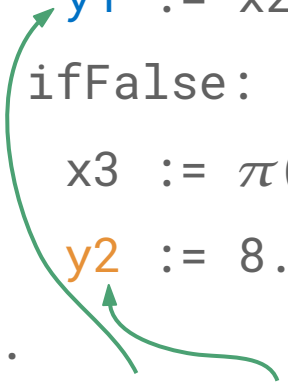
x4 := π(x2, >=10) and x2 := π(x1, <5)

Is (-∞; 5) ∩ [10; ∞) empty?

YES ⇒ Unreachable

Constant dead branch elimination

```
x1 < 5 ifTrue: [
    x2 := π(x1, <5).
    y1 := x2.
] ifFalse: [
    x3 := π(x1, >=5).
    y2 := 8.
].
y3 := Φ(y1, y2)
```



What are the possible values of y3?

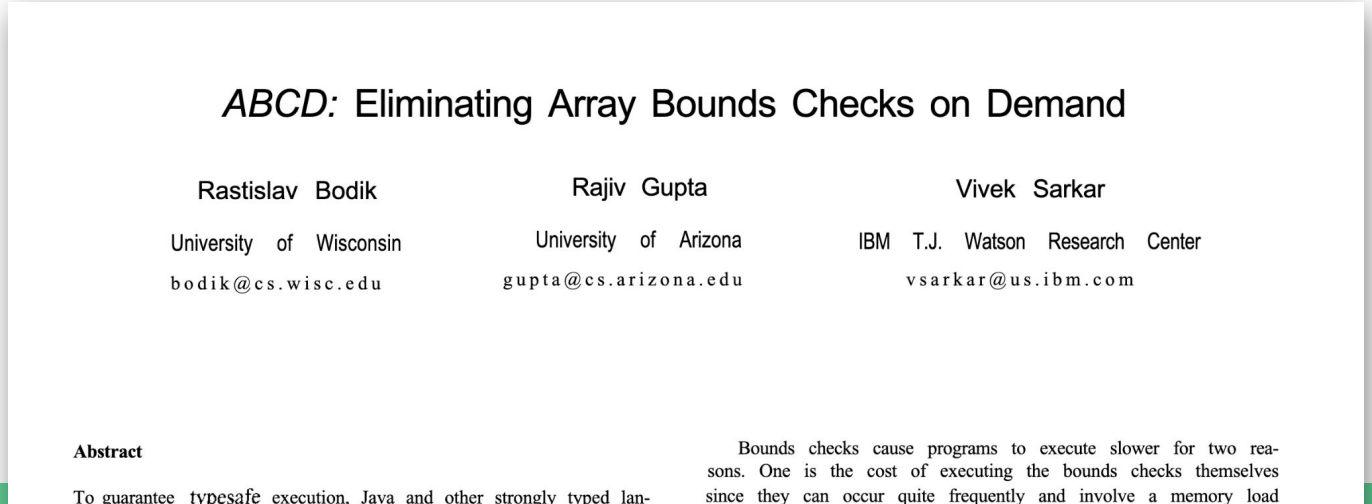
The union between the possible values of y1 and y2

(-∞; 5) ∪ {8}

ABCD method

More powerful:

- Models the relationship between variables
- Models the effect of basic arithmetic operations (addition and subtraction)



ABCD: Eliminating Array Bounds Checks on Demand

Rastislav Bodik
University of Wisconsin
bodik@cs.wisc.edu

Rajiv Gupta
University of Arizona
gupta@cs.arizona.edu

Vivek Sarkar
IBM T.J. Watson Research Center
vsarkar@us.ibm.com

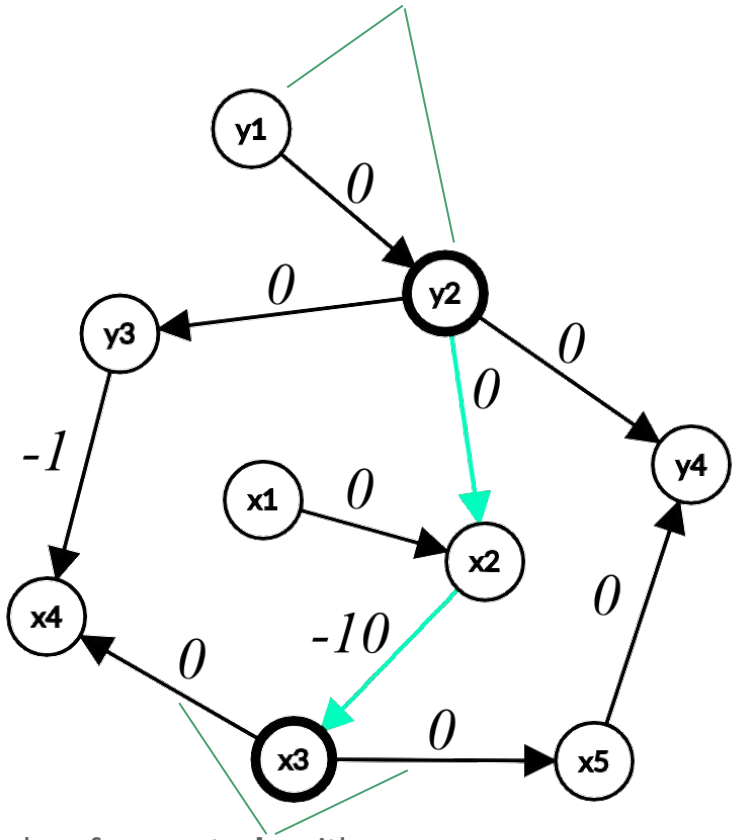
Abstract

To guarantee typesafe execution, Java and other strongly typed lan-

Bounds checks cause programs to execute slower for two reasons. One is the cost of executing the bounds checks themselves since they can occur quite frequently and involve a memory load

ABCD method

```
x1 <= y1 ifTrue: [
    x2 := π(x1, <=y1).
    y2 := π(y1, >=x1).
    x3 := x2 - 10.
    x3 < y2 ifTrue: [ "tautology"
        x4 := π(x3, <y2).
        y3 := π(y2, >x3).
    ] ifFalse: [      "unreachable"
        x5 := π(x3, >=y2).
        y4 := π(y2, <=x3).
    ] ].
```



An edge from **a** to **b** with weight **w** means that **b** - **a** ≤ **w**.

Experiments and results

Experimental Context

Druid

- source-to-source meta-compiler
- uses many optimization passes
  - Analysis and code transformation

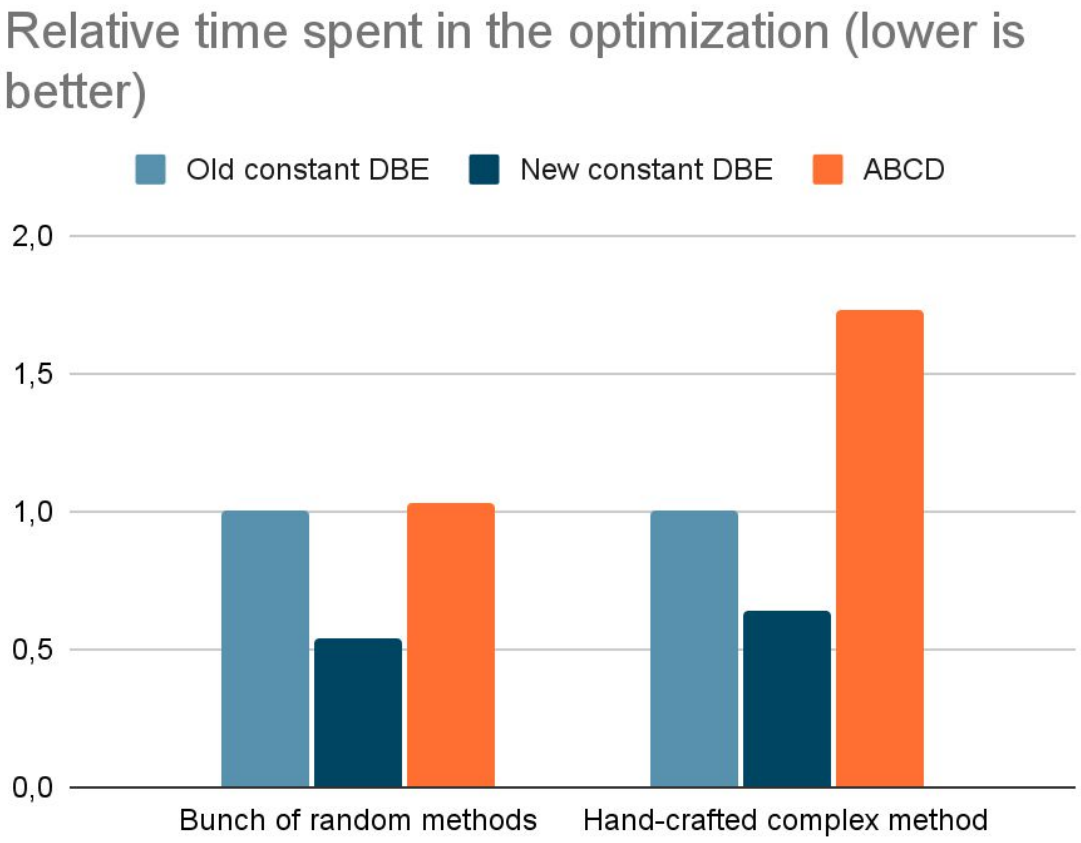
Old DBE vs PiNodes

- Druid already had a DBE pass
- Worked by computing all paths a variable was alive in
- Questions:
  - Is our new constant DBE method faster?
  - Is ABCD, the more powerful method, slower?

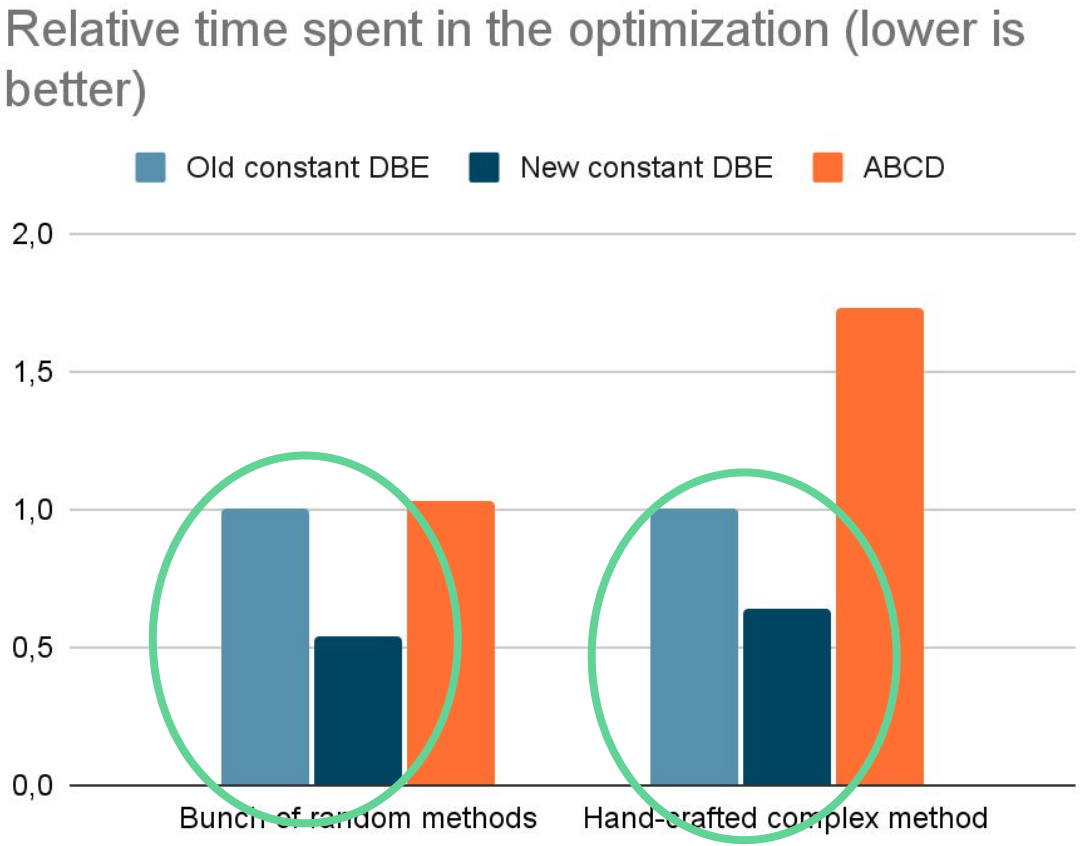
Measuring Compile Time Improvement

- Used two benchmarks to compare time spent optimizing:
  - Compiled all methods of a test class
  - Compiled one hand-crafted method with an intentionally complex control flow

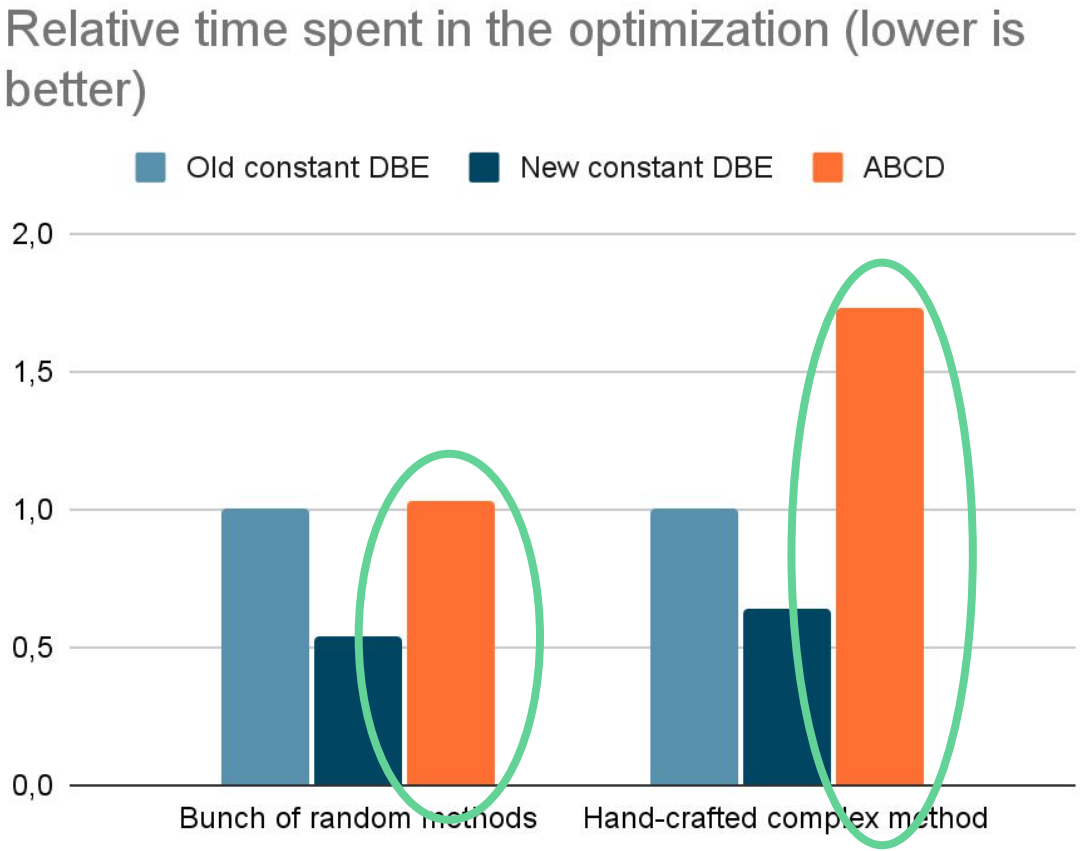
Results


Relative time spent in the optimization (lower is better)
Old constant DBE
New constant DBE
ABCD
2,0
1,5
1,0
0,5
0,0
Bunch of random methods
Hand-crafted complex method

Results


Relative time spent in the optimization (lower is better)
Old constant DBE
New constant DBE
ABCD
2,0
1,5
1,0
0,5
0,0
Bunch of random methods
Hand-crafted complex method

Results


Relative time spent in the optimization (lower is better)
Old constant DBE
New constant DBE
ABCD
2,0
1,5
1,0
0,5
0,0
Bunch of random methods
Hand-crafted complex method

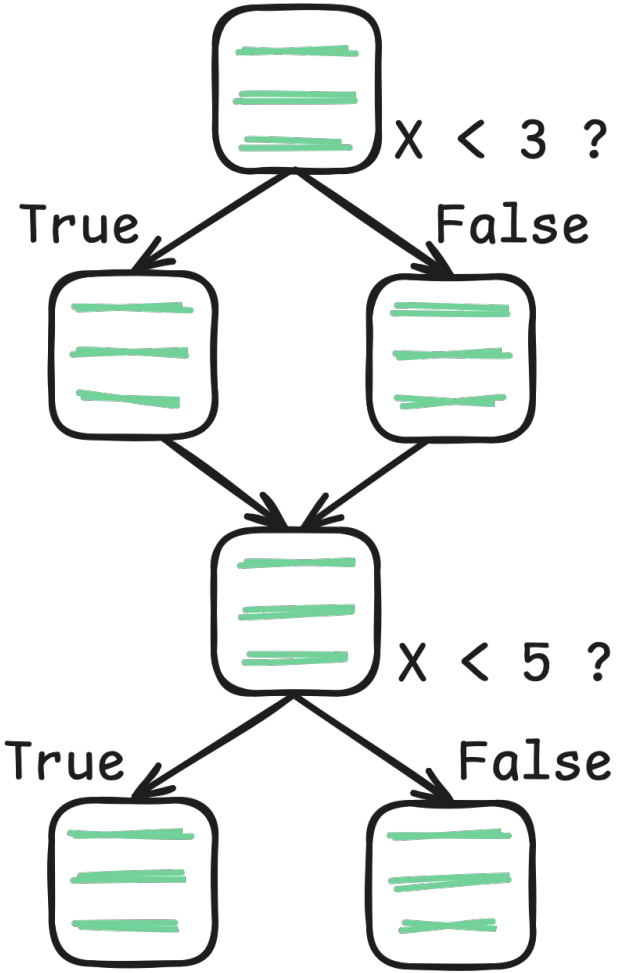
Future work

Future Work

- Stronger constraint solving - Z3
- Measuring run time improvements
- Looking for more optimization opportunities
  - Using the Druid optimizer for high-level Pharo code
  - Message splitting

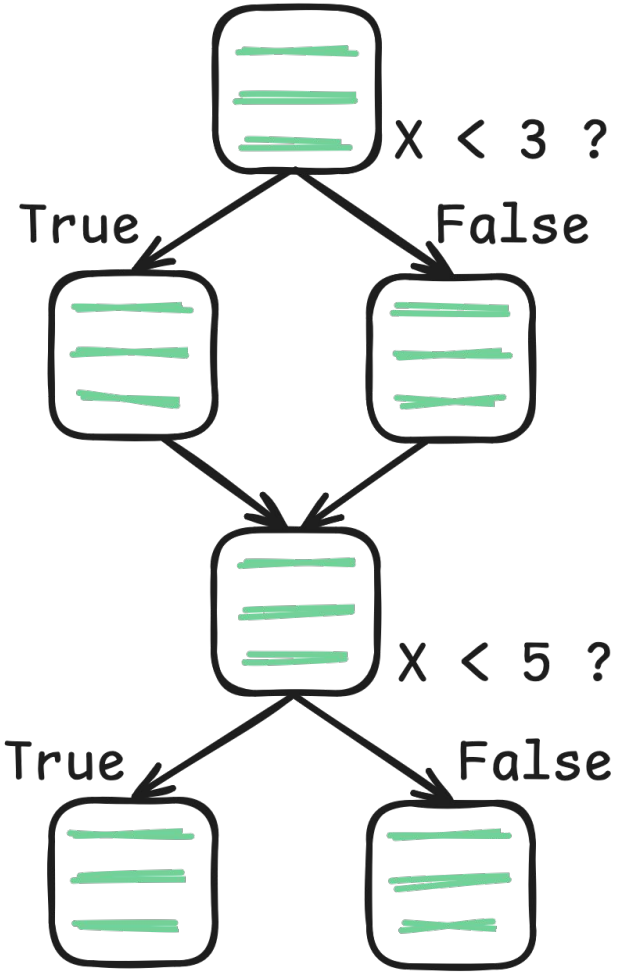
More Opportunities for Complex Control Flows

```
x < 3 ifTrue: [
    y doSomething.
].
x < 5 ifTrue: [
    z doSomethingElse.
].
```



Future: Message splitting

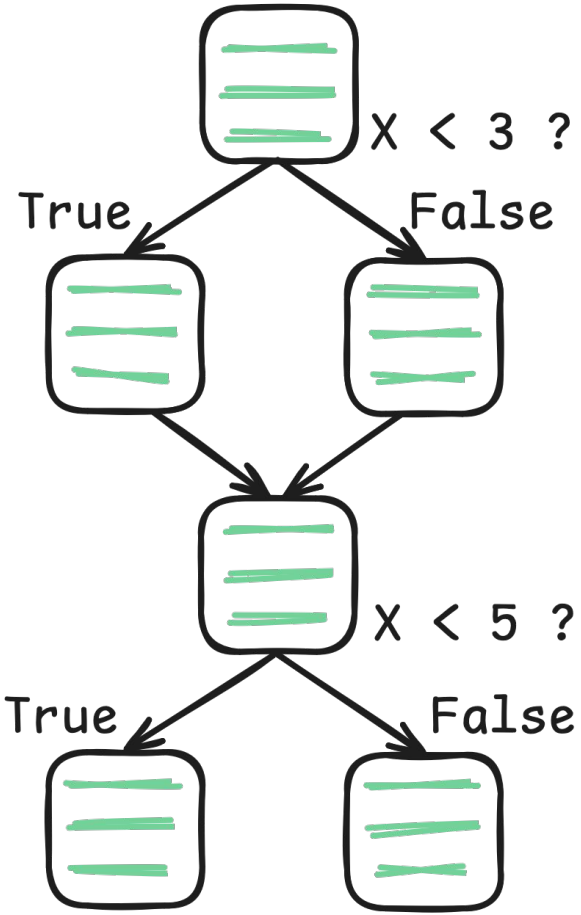
```
x < 3 ifTrue: [
    y doSomething.
].
x < 5 ifTrue: [
    z doSomethingElse.
].
```

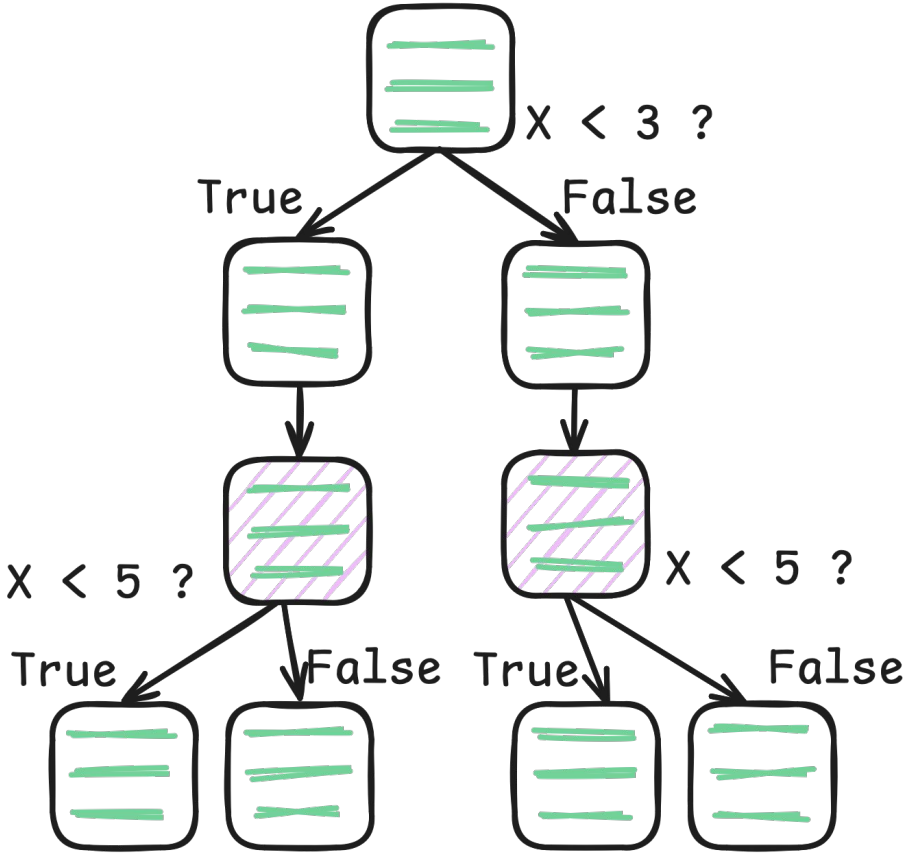


Iterative Type Analysis and
Extended Message Splitting: Optimizing
Dynamically-Typed Object-Oriented Programs*

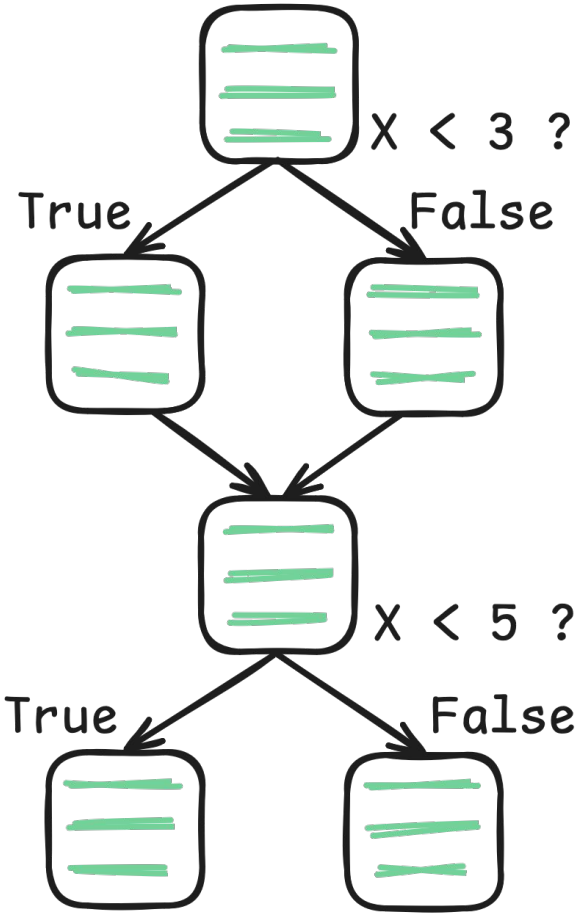
CRAIG CHAMBERS (craig@self.stanford.edu)
DAVID UNGAR† (ungar@self.stanford.edu)
Computer Systems Laboratory, Stanford University, Stanford, California 94305

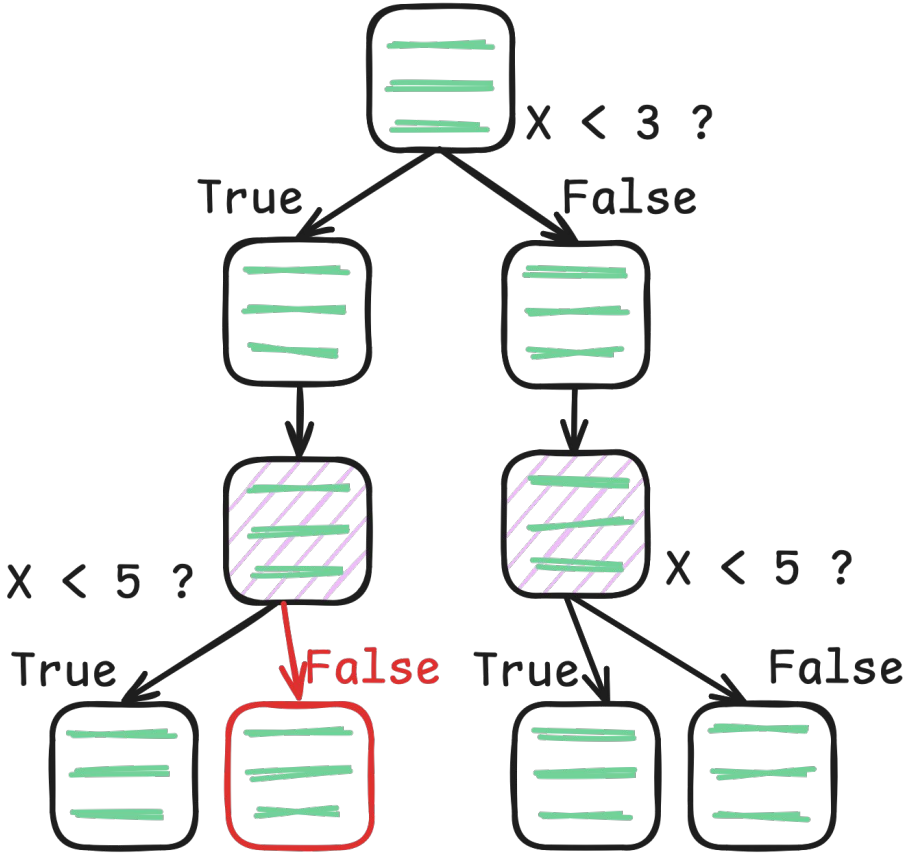
Future: Message splitting



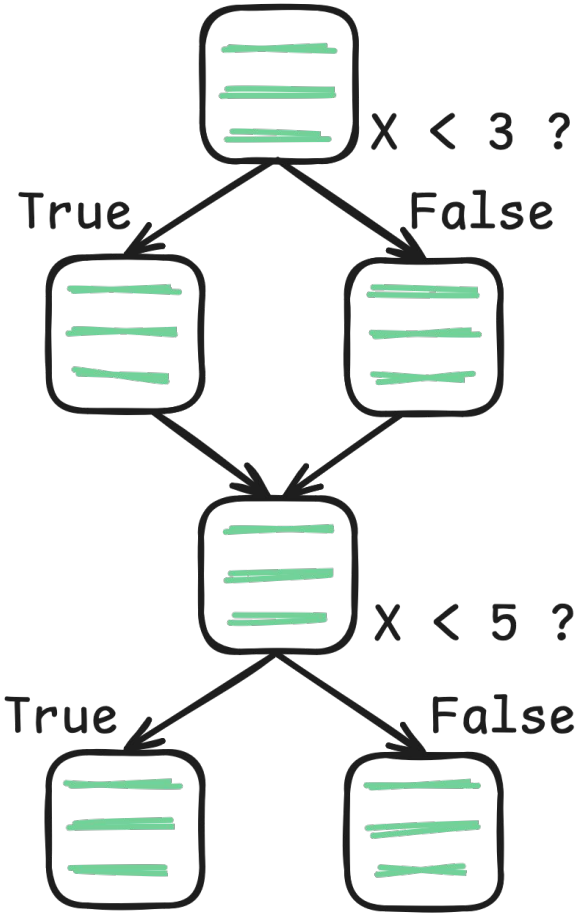


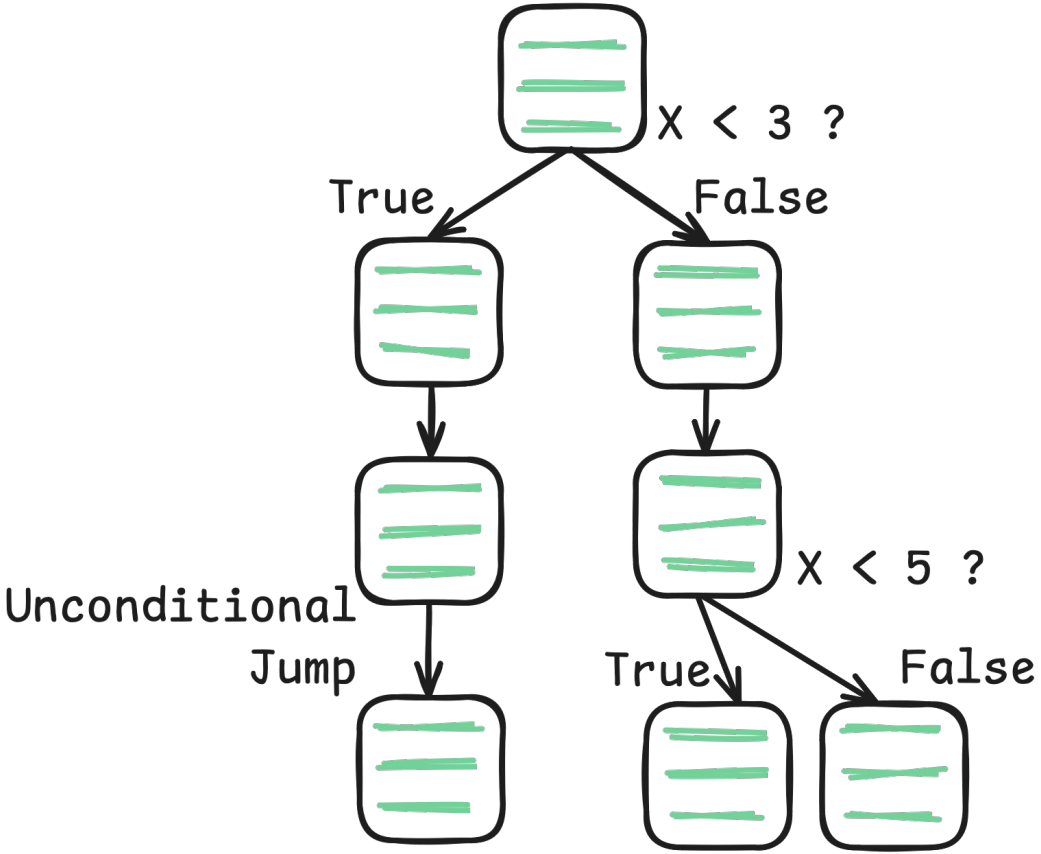
Future: Message splitting





Future: Message splitting





Conclusions

- Branches make code slow
- It's common to have some dead branches in your code
- PiNodes represent constraints on SSA variables, and can be used for DBE

Matías Demare - Guillermo Polito
Javier Pimás - Nahuel Palumbo

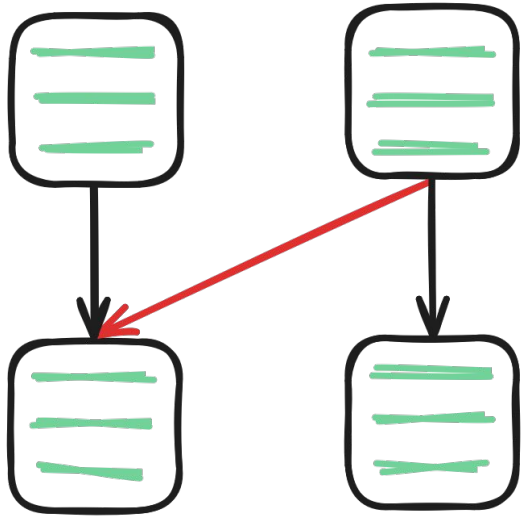
matias-nicolas.demare@inria.fr

github.com/m-demare

Addendum

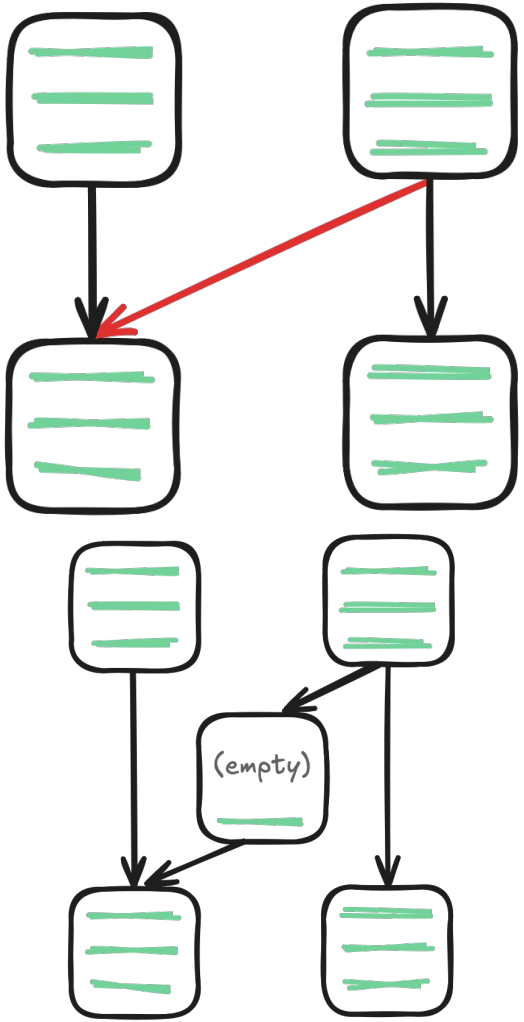
Critical Edges

- Edges whose successor has multiple predecessors, and whose predecessor has multiple successors
- They are annoying for PiNode insertion, because the successor is not dominated by the block containing the condition



Breaking Critical Edges

- Remove that edge, and insert
- Insert a new basic block with just an unconditional jump to the critical edge's target in its place



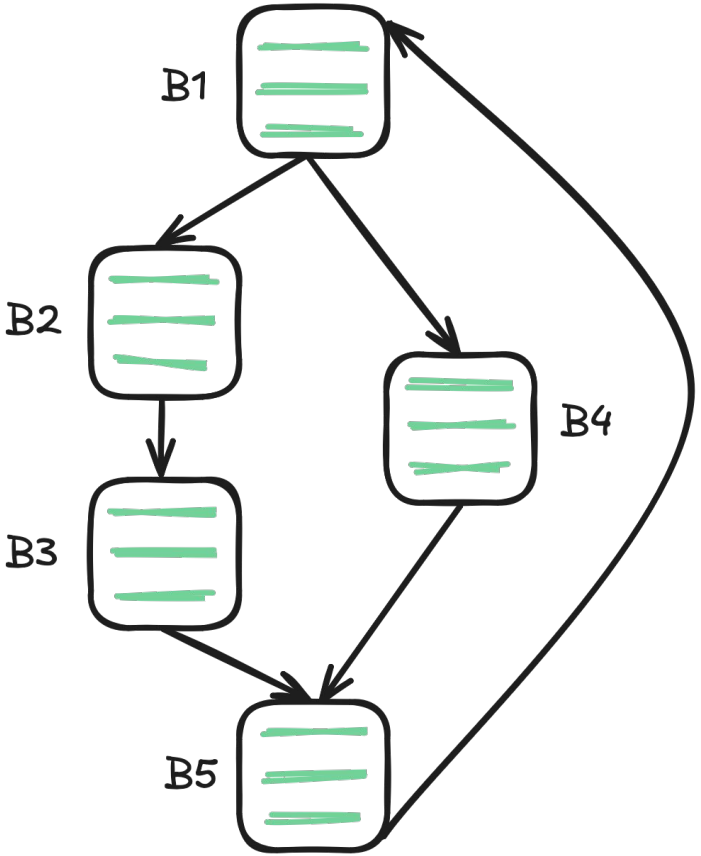
Domination

- B_1 dominates B_2 if every path from the entry node to B_2 must go through B_1

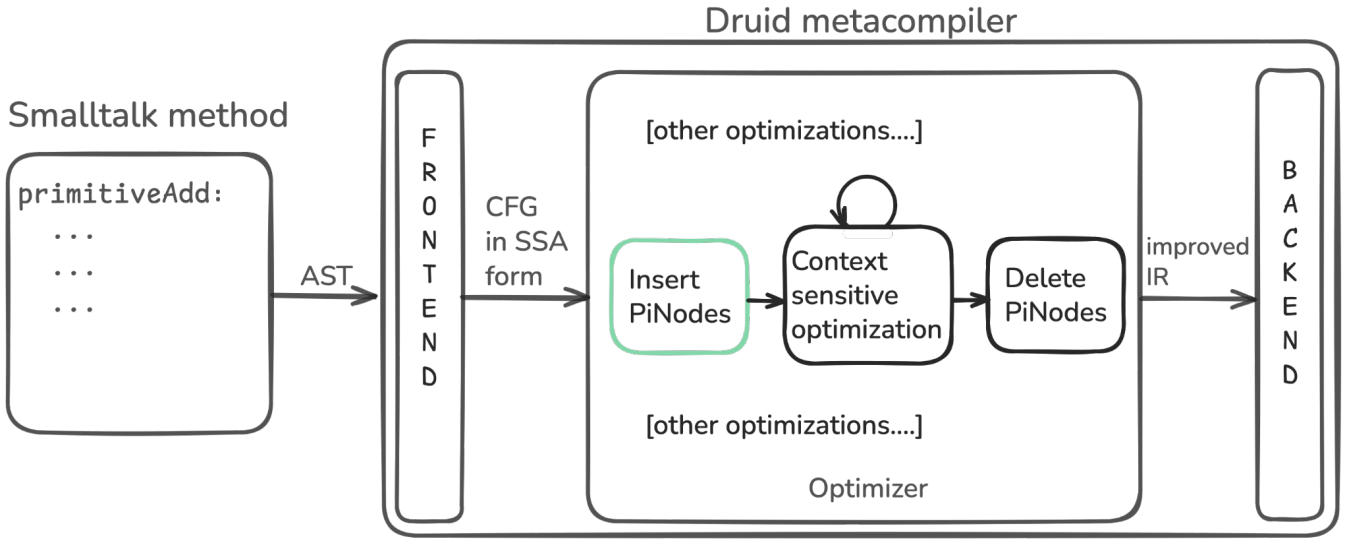
B_1 dominates B_1, B_2, B_3, B_4, B_5

B_2 dominates B_2, B_3

B_3, B_4, B_5 only dominate themselves

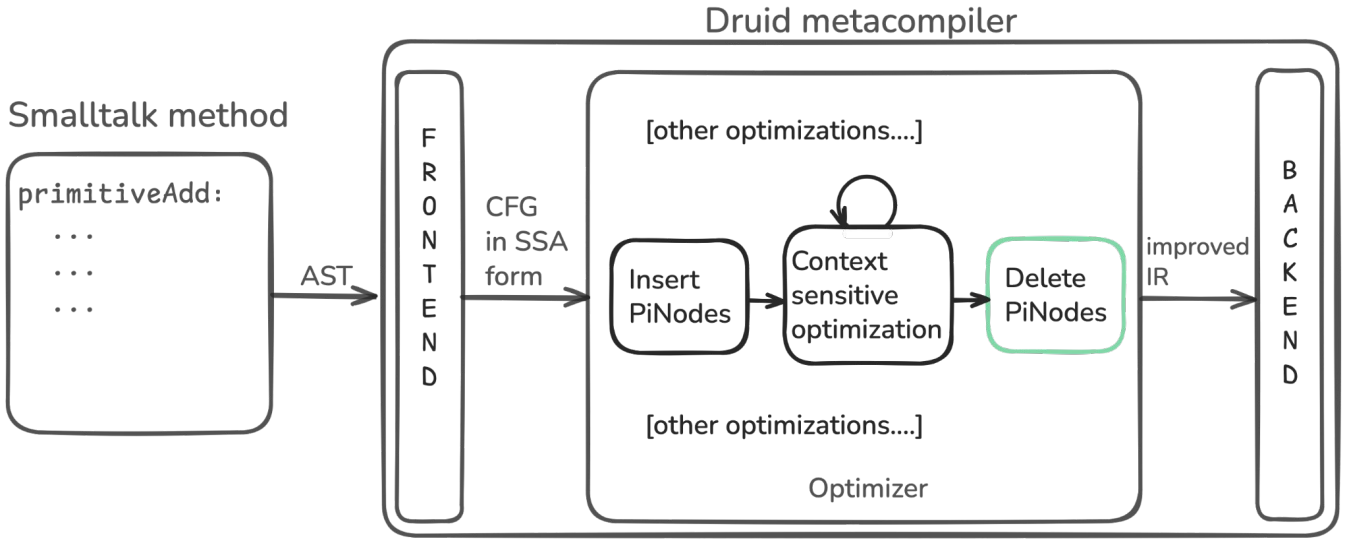


The PiNode Framework - insertion



- Break critical edges
- Insert PiNodes in each successor of a condition (one for each variable involved)
- Replace usages in dominated blocks

The PiNode Framework - deletion



Simple copy propagation algorithm: replace each usage of the PiNode for a usage of the copied variable

Dead branch elimination

Basic pseudocode of the algorithm

```
cfg piNodesDo: [ :piNode |
    piNode ifNotSatisfiable: [
        unreachableBlocks add:
            piNode basicBlock.
    ].
].
cfg removeJmpsTo: unreachableBlocks.
cfg removeBlocks: unreachableBlocks.
```

Message splitting (with code)

```
x < 3 ifTrue: [
    y doSomething.
] ifFalse: [].
x < 5 ifTrue: [
    z doSomethingElse.
].
```

```
x < 3 ifTrue: [
    y doSomething.
    x < 5 ifTrue: [
        z doSomethingElse.
    ].
] ifFalse: [
    x < 5 ifTrue: [
        z doSomethingElse.
    ].
].
```

Message splitting (with code)

```
x < 3 ifTrue: [
    y doSomething.
] ifFalse: [].
x < 5 ifTrue: [
    z doSomethingElse.
].
```

```
x < 3 ifTrue: [
    y doSomething.
    x < 5 ifTrue: [
        z doSomethingElse.
    ].
] ifFalse: [
    x < 5 ifTrue: [
        z doSomethingElse.
    ].
].
```