Hardware-Aware Domain-Specific Transformations

Hugo Pompougnac

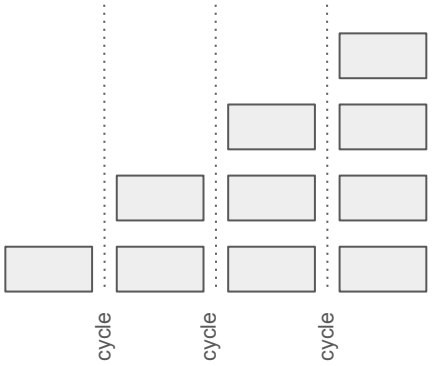
Sasha Lopoukhine
Tobias Grosser

Objective: Fast Micro-Kernels

$$Z=X_i+Y_i$$

Linear Algebra

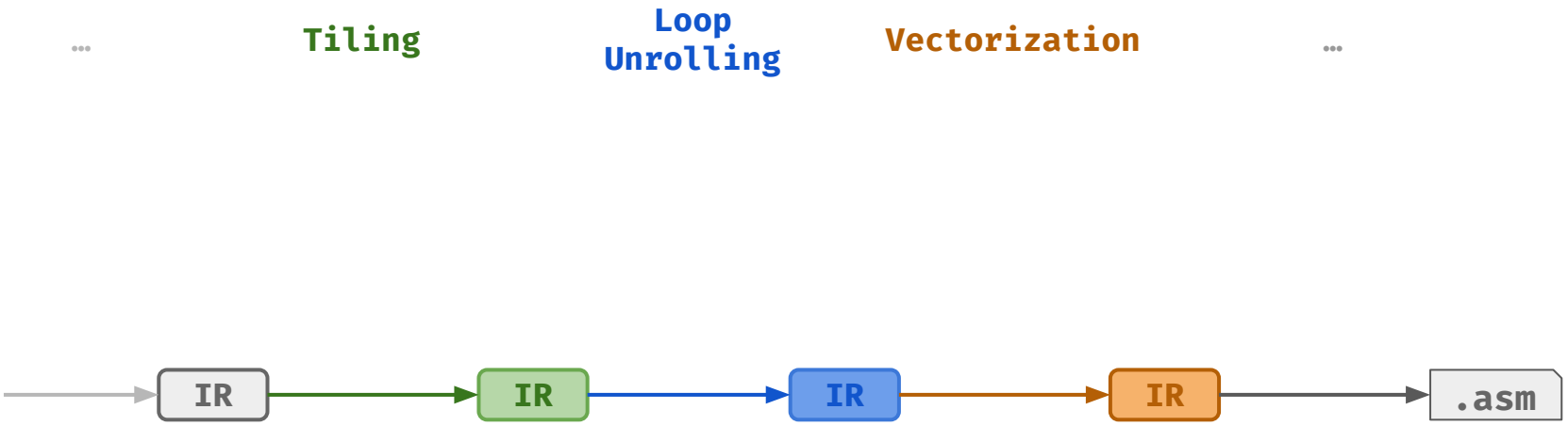
Single Core



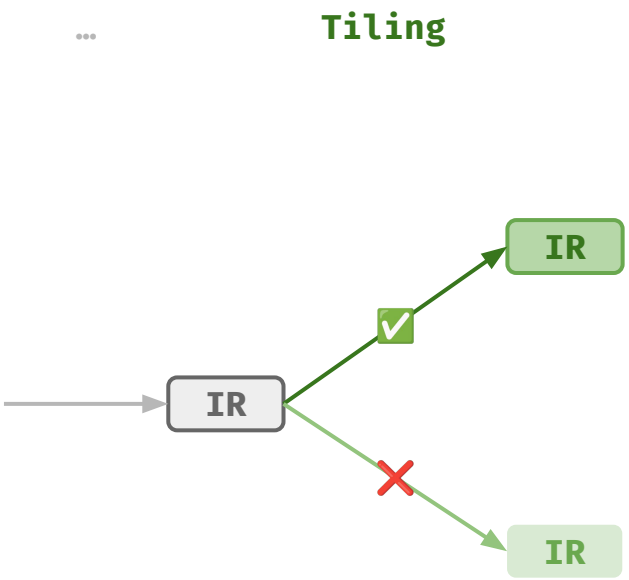
FPU Always Busy

How do we get to peak performance?

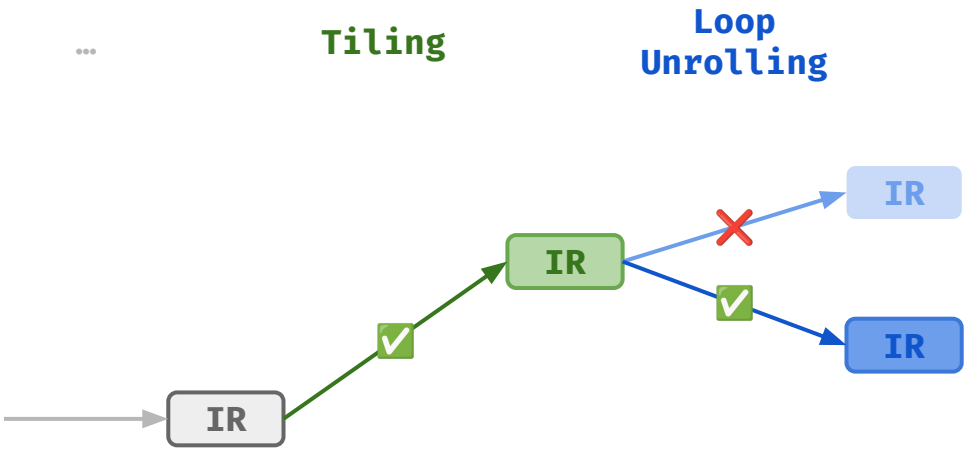
Compilers Lower IR in Phases …



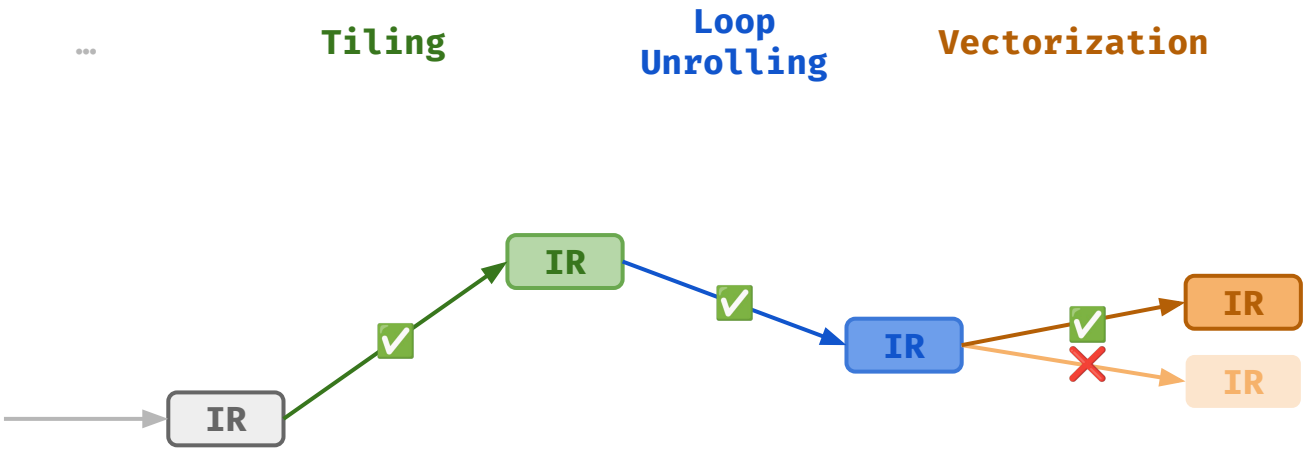
… Making Choices at Multiple Levels …



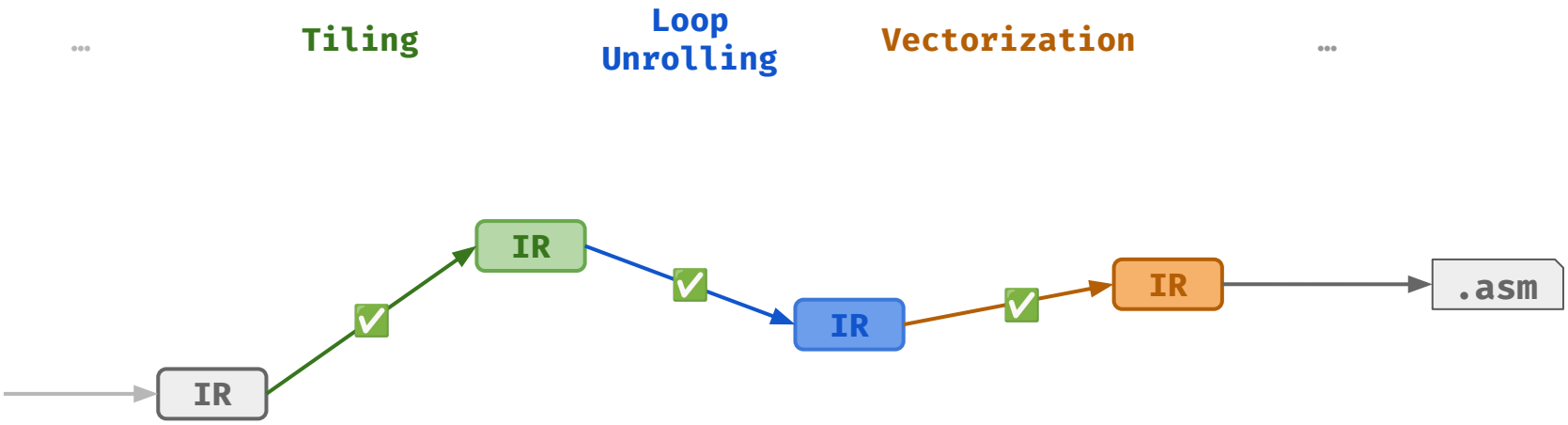
… Making Choices at Multiple Levels …



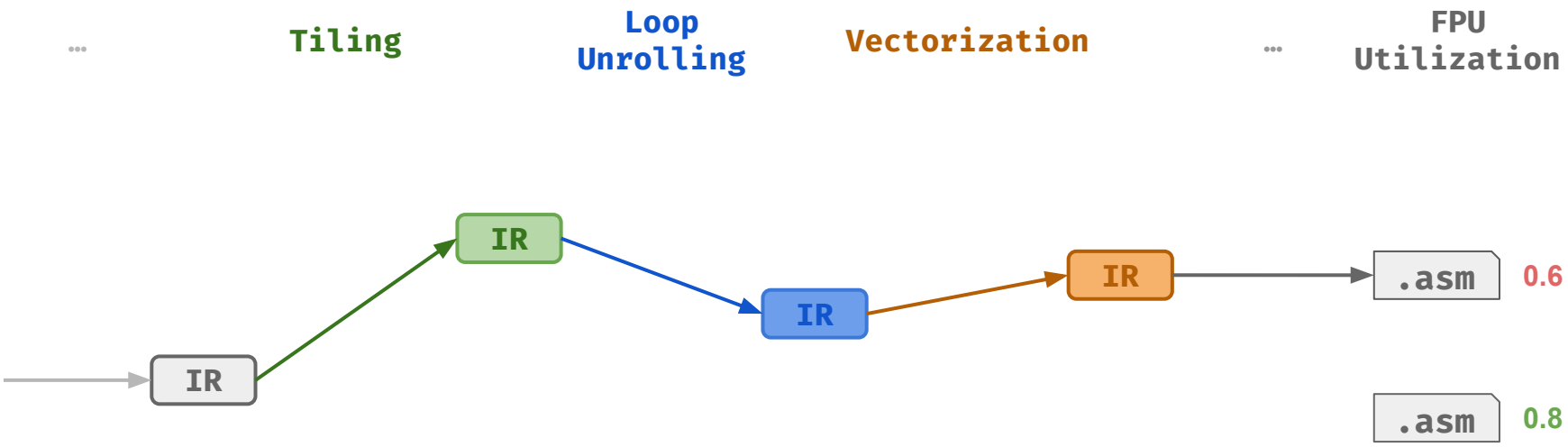
… Making Choices at Multiple Levels …



… Making Choices at Multiple Levels …

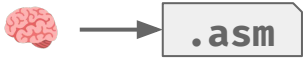


… And Don’t Always Produce Optimal Code



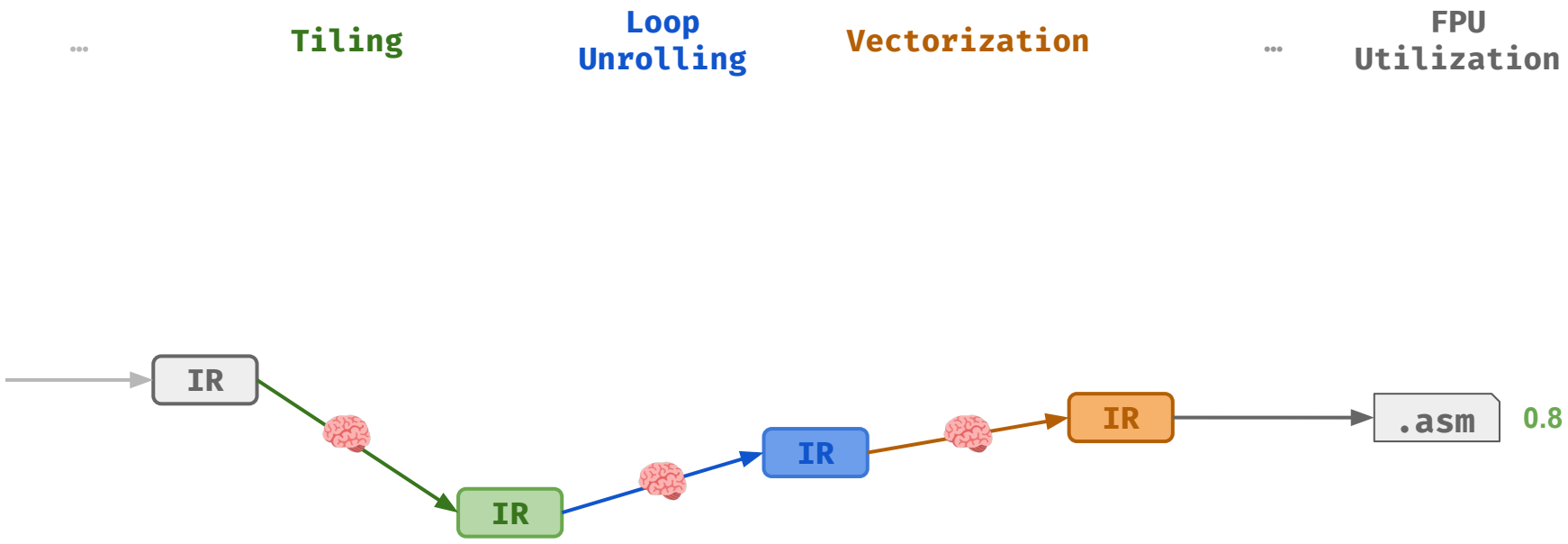
Expert Programmers Bypass The Compiler …

FPU
Utilization

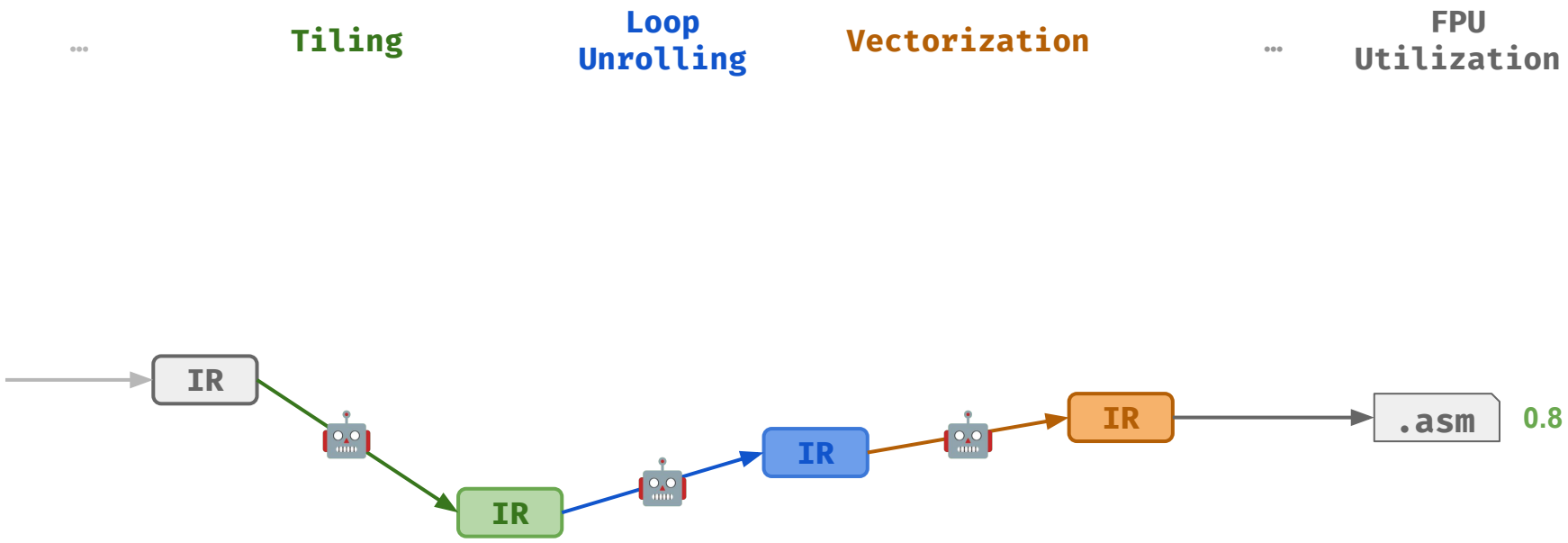


0.8

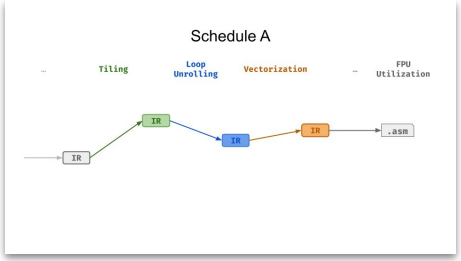
… Or Provide Schedules That Guide It

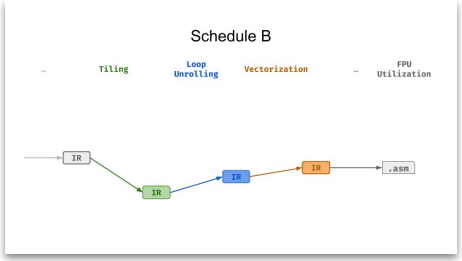


Can We Automate The Schedule Per Kernel?



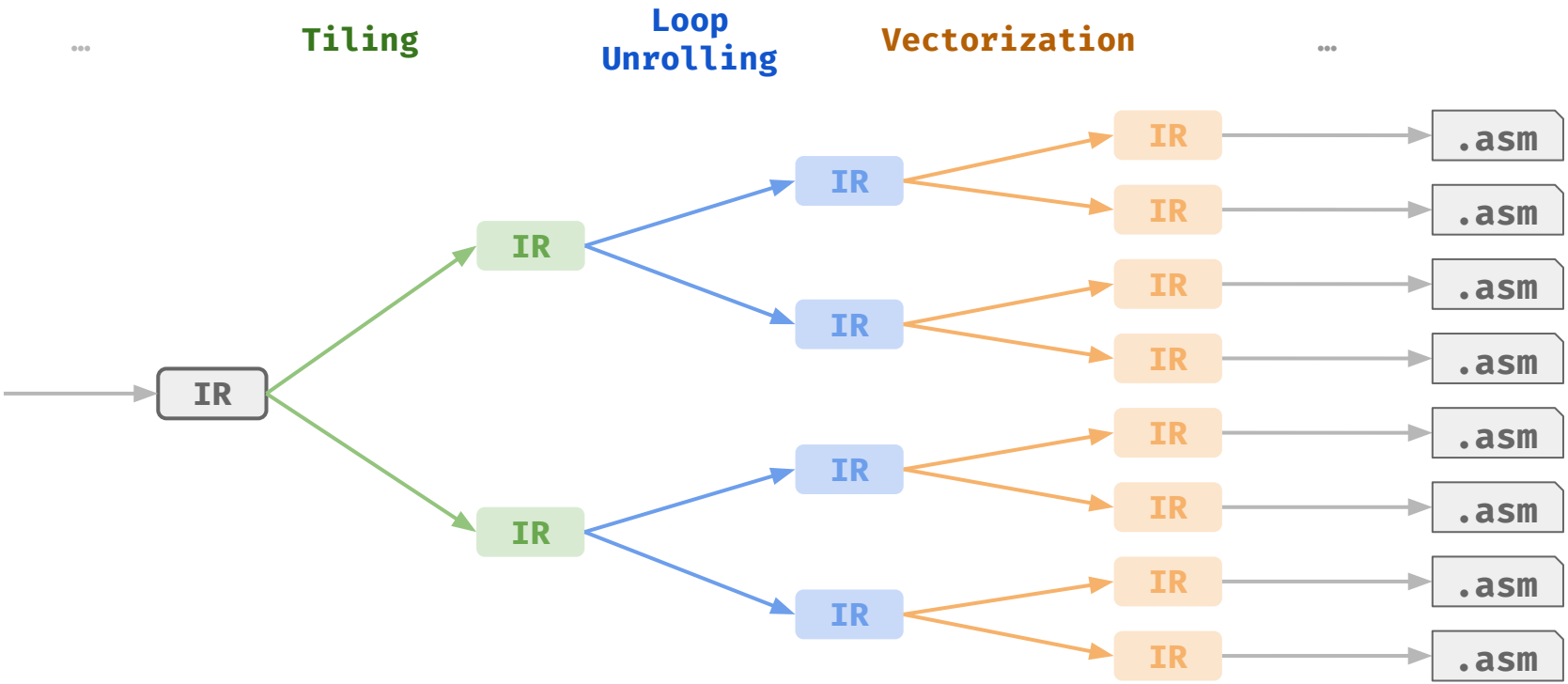
Compilers Optimize for Best-Average-Case Performance





MatMul 16x16x16	0.6	0.8
MatMul 64x64x64	0.7	0.3
MatMul 64x16x128	0.5	0.2
…	…	…

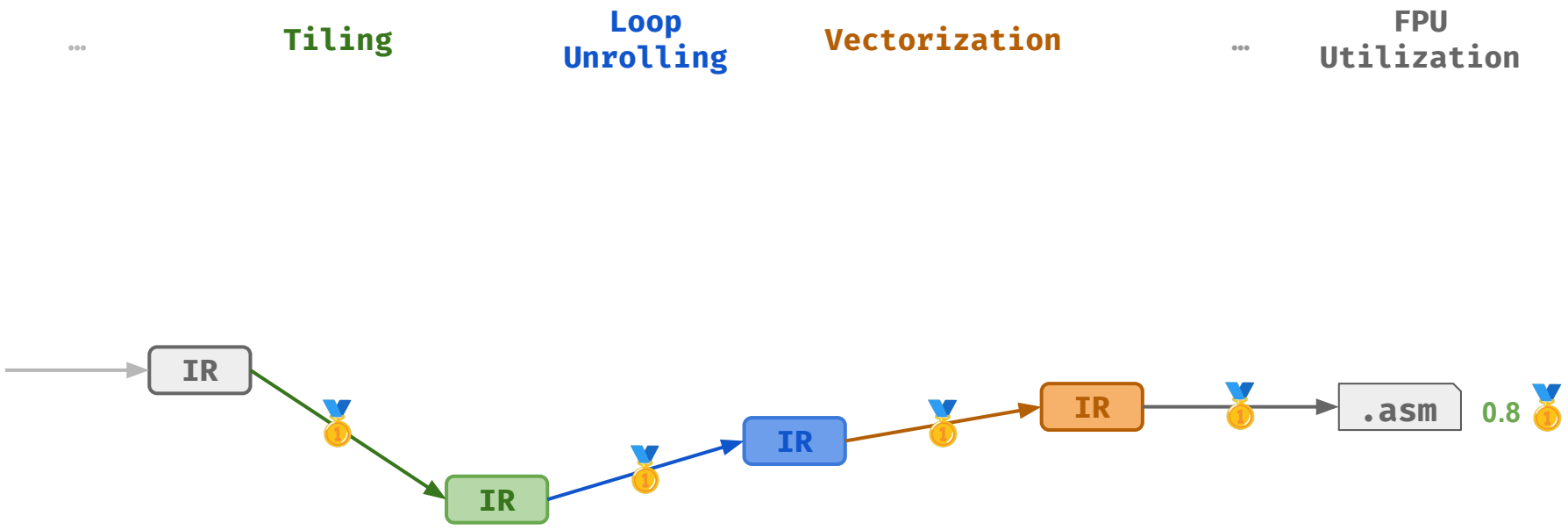
Step 1: Generate Full Range of Lowerings



Step 2: Evaluate Performance

.asm	0.5
.asm	0.4
.asm	0.6
.asm	0.3
.asm	0.8
.asm	0.7
.asm	0.1
.asm	0.2

Step 3: Take The Best



How Do We Do This In Practice?

🎯

Accurate

🚅

Fast

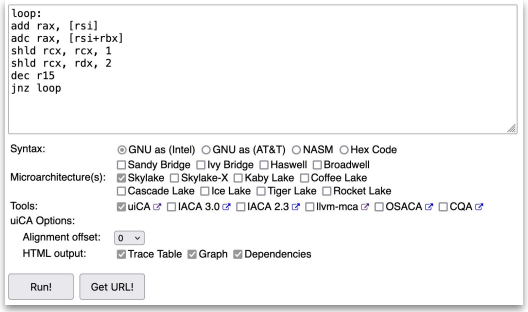
🤸

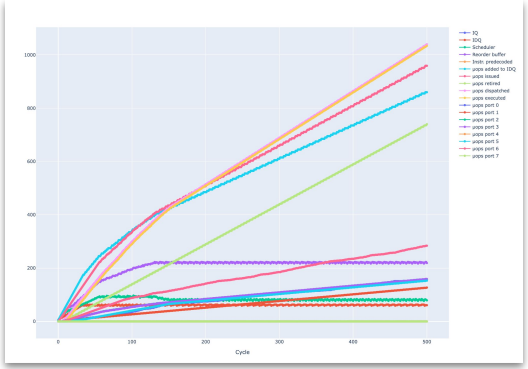
Flexible

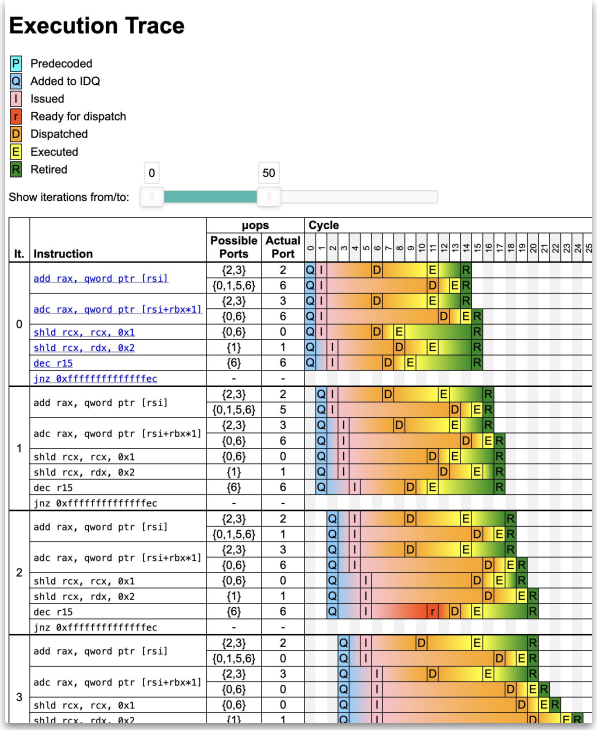
Micro-Architecture-Aware Cost Models 🎯🚅

ARM	x86	RISC-V
llvm-mca	llvm-mca	llvm-mca
	uiCA	GVSoC
	IACA	

uiCA - The uops.info Code Analyzer







Parameterizable Domain-Specific Passes 🤸

Tiling	Loop Unrolling	Vectorization
(4,4,4)	4x	2xf32
(8,8,8)	8x	4xf32
…	…	…

Tiling 128x128x128 MatMul by (16,16,16)

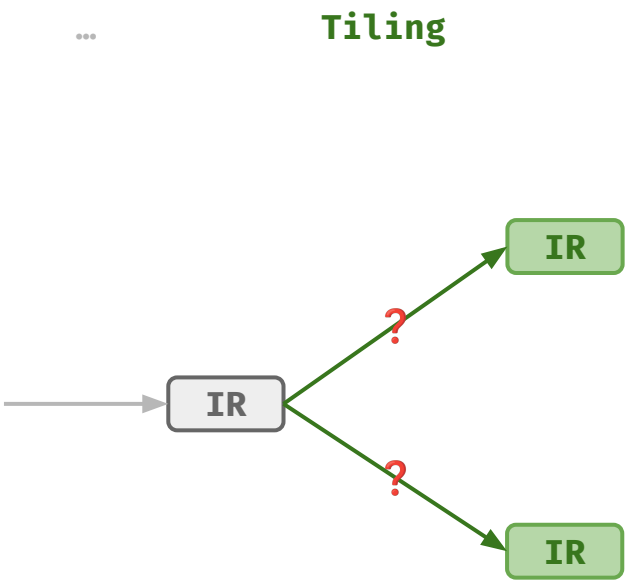
```
def matmul(A, B, C):
  for i in range(128):
    for j in range(128):
      for k in range(128):
        C[i,j] += A[i,k] * B[k,j]
```

```
def matmul(A, B, C):
  for io in range(0, 128, 16):
    for jo in range(0, 128, 16):
      for ko in range(0, 128, 16):
        for ii in range(16):
          for ji in range(16):
            for ki in range(16):
              i = io + ii
              j = jo + ji
              k = ko + ki
              C[i,j] += A[i,k] * B[k,j]
```

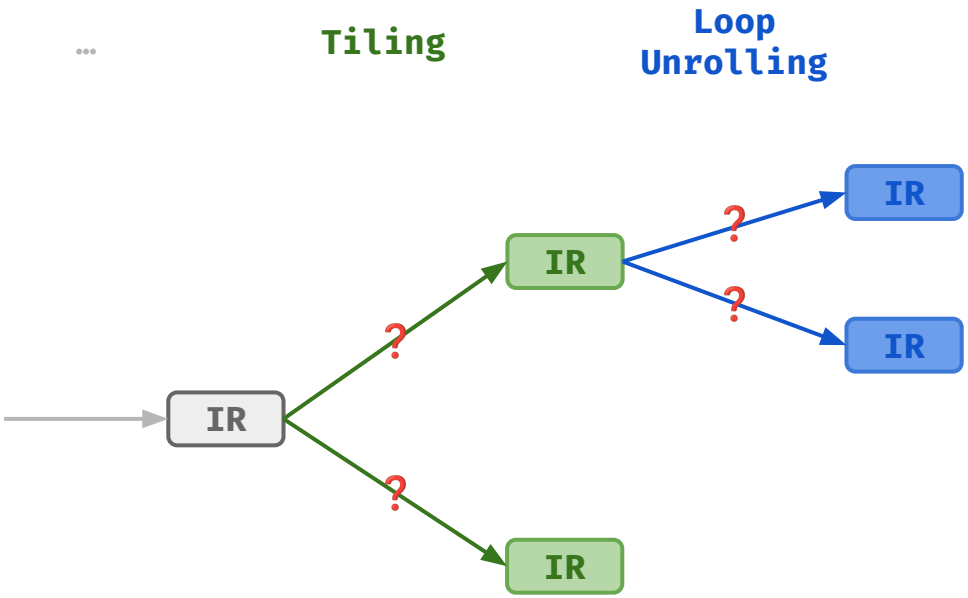
Parameterizable Target-Specific Passes ❓

Instruction Selection	Instruction Scheduling	Register Allocation
??	??	??

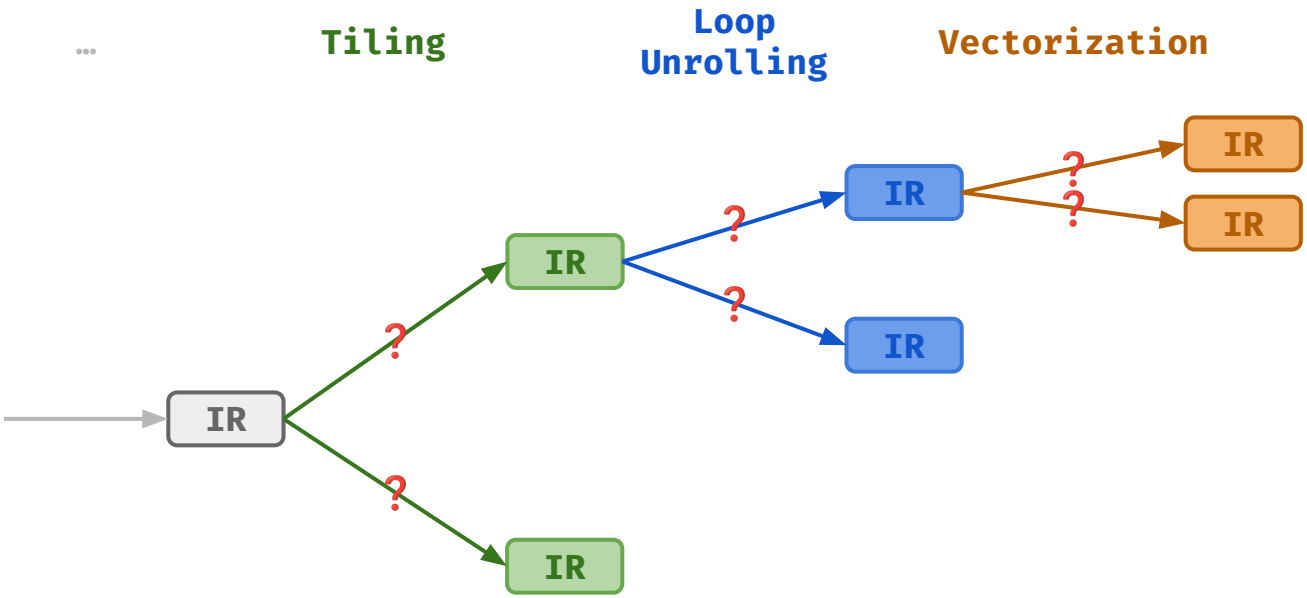
Recursive Exploration



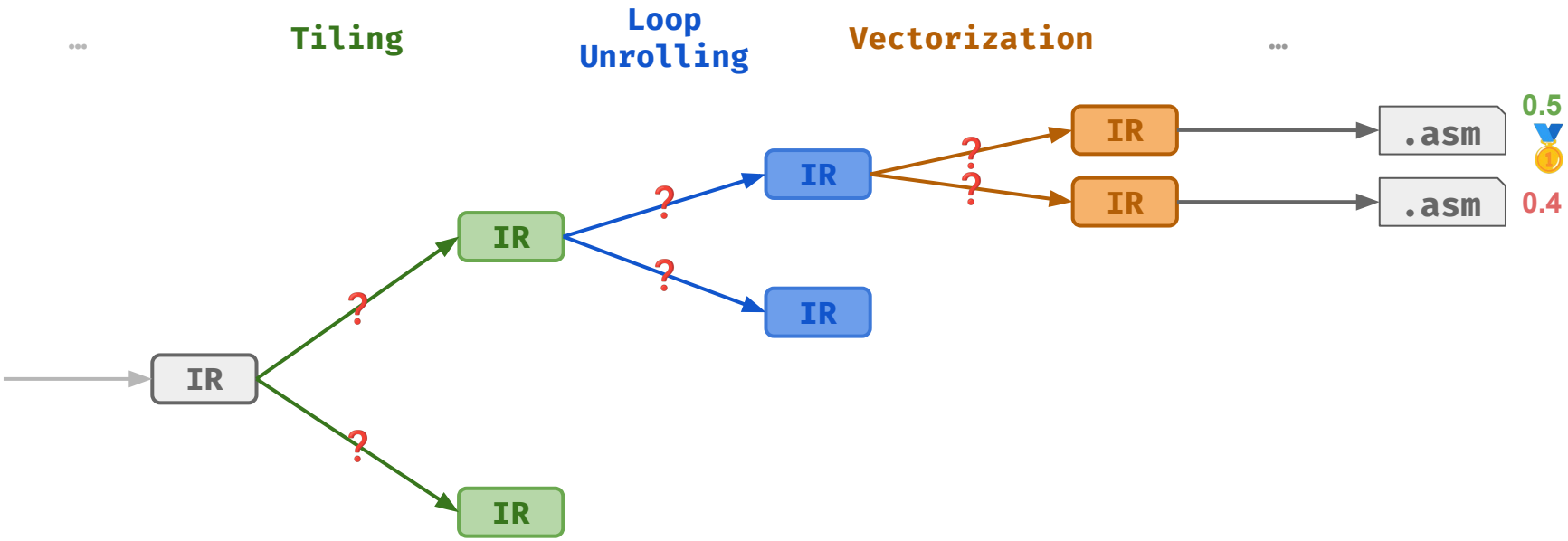
Recursive Exploration



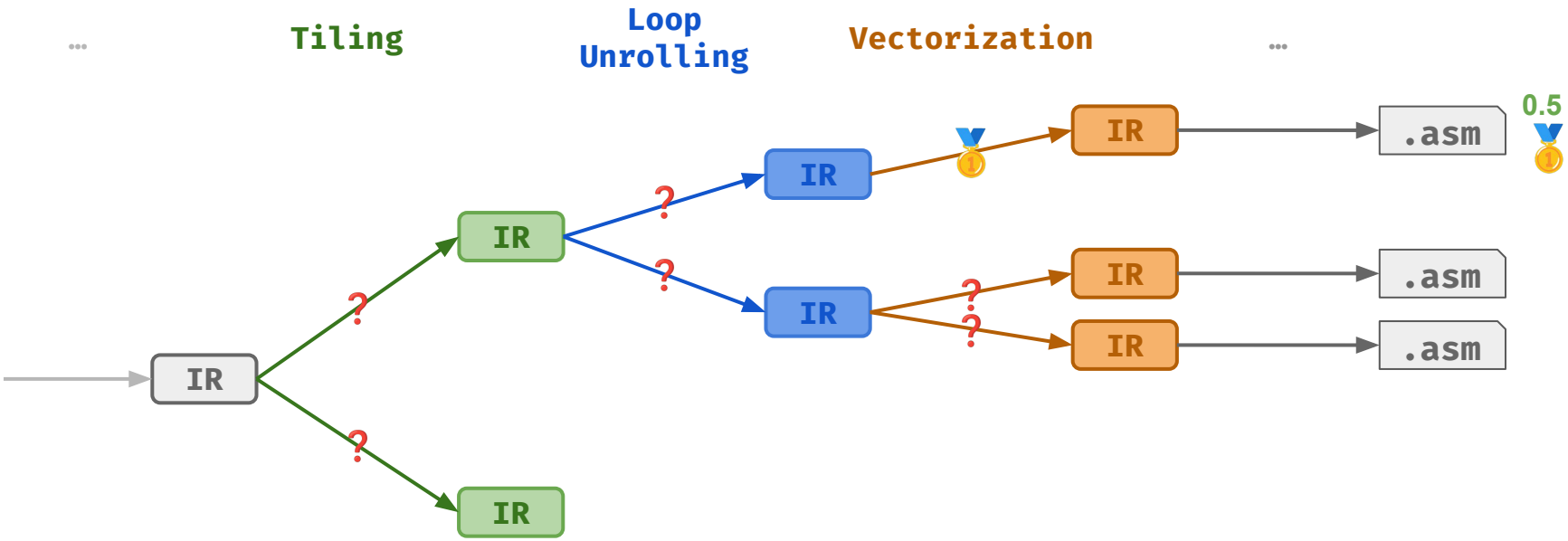
Recursive Exploration



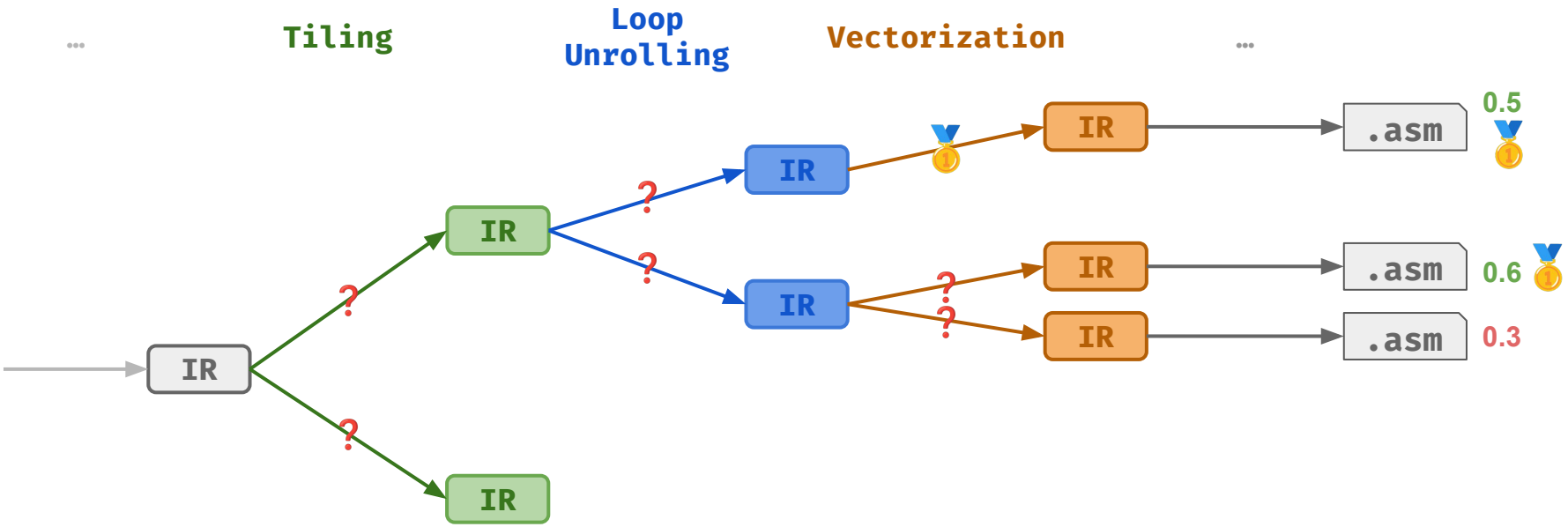
Recursive Exploration



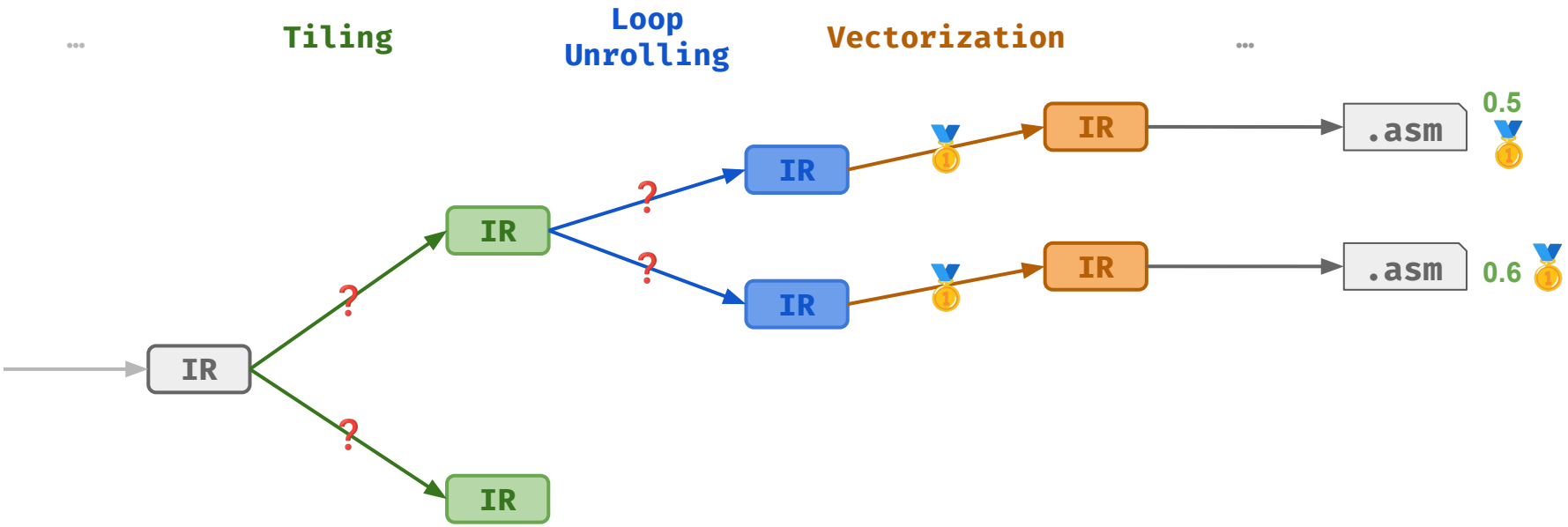
Recursive Exploration



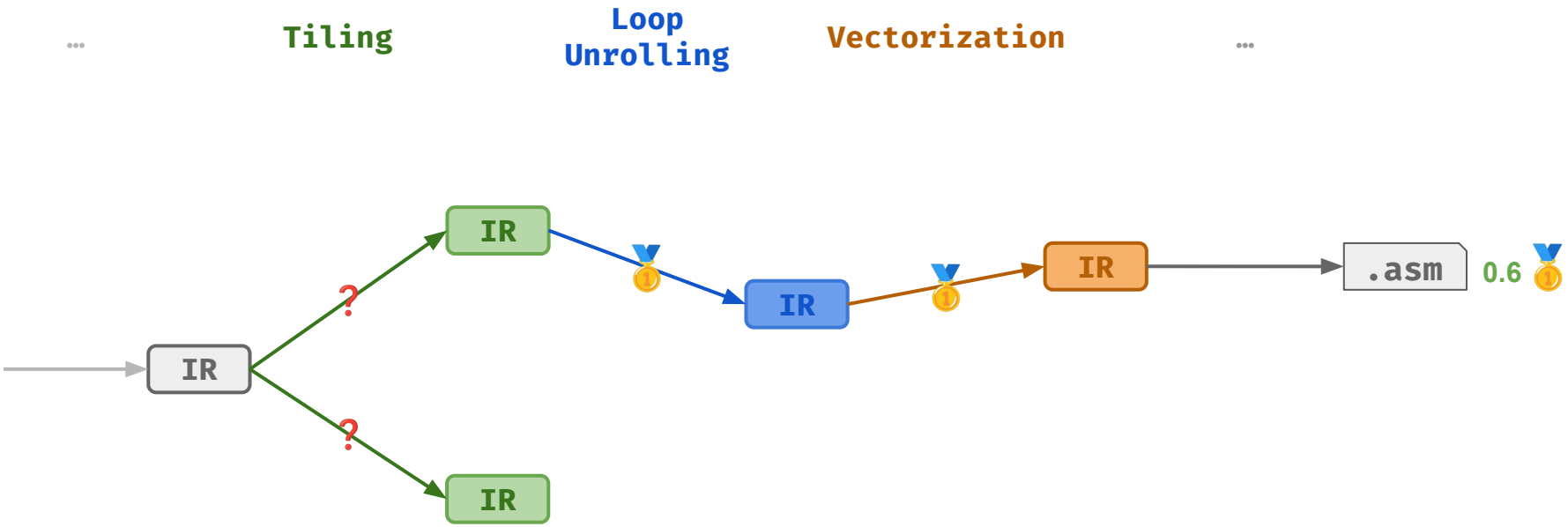
Recursive Exploration



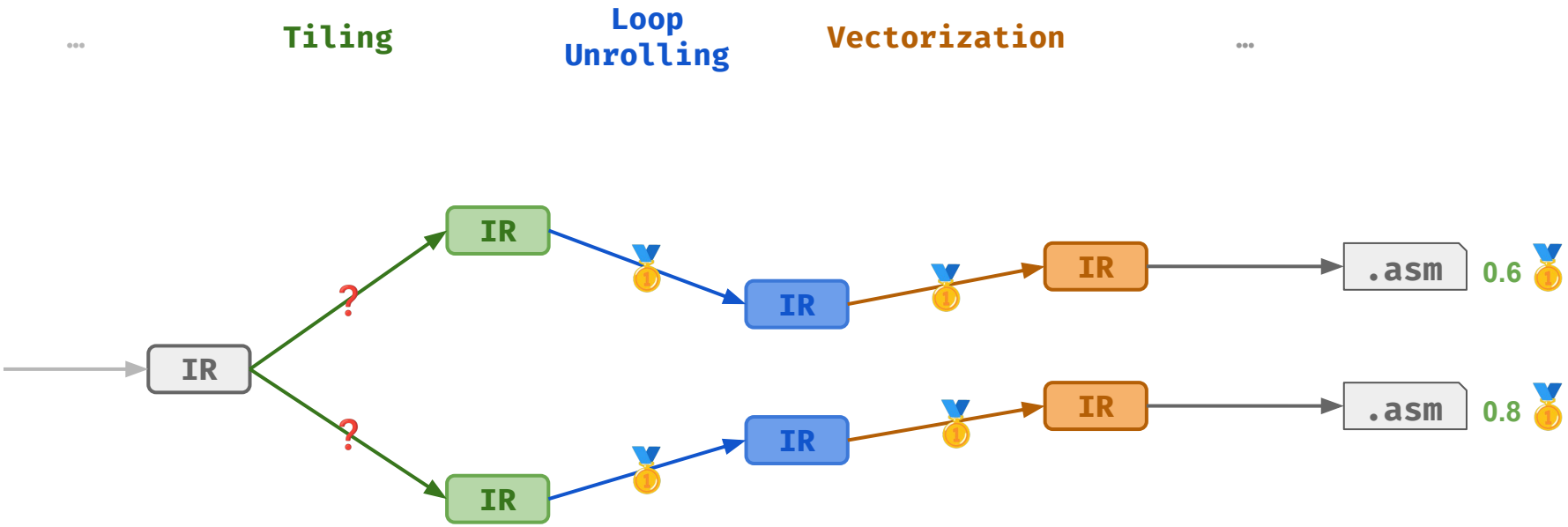
Recursive Exploration



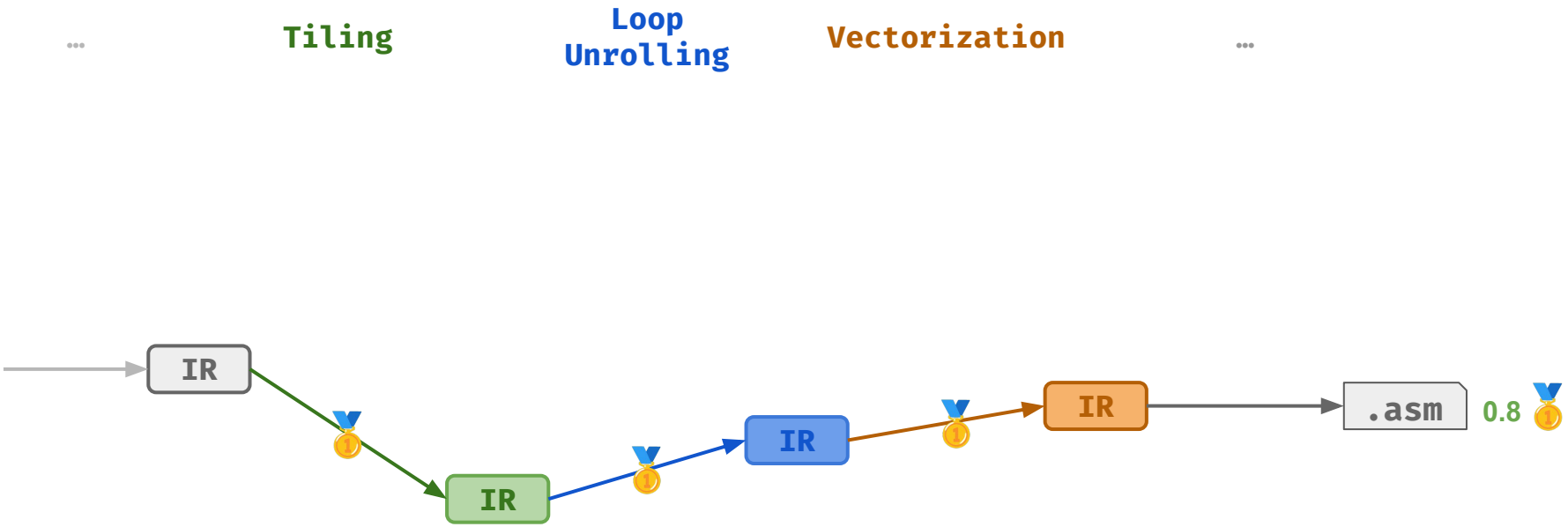
Recursive Exploration



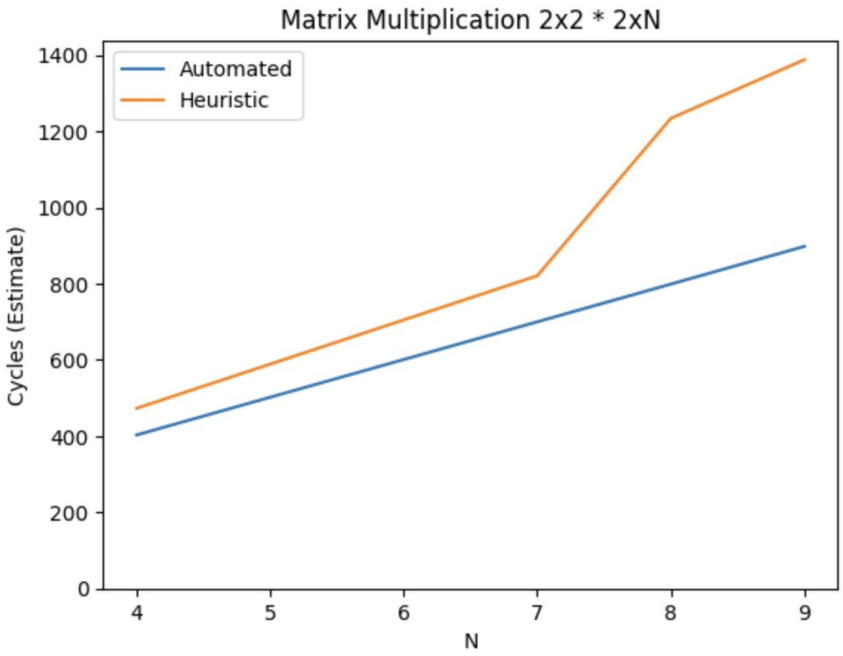
Recursive Exploration



Recursive Exploration



A First Experiment in xDSL



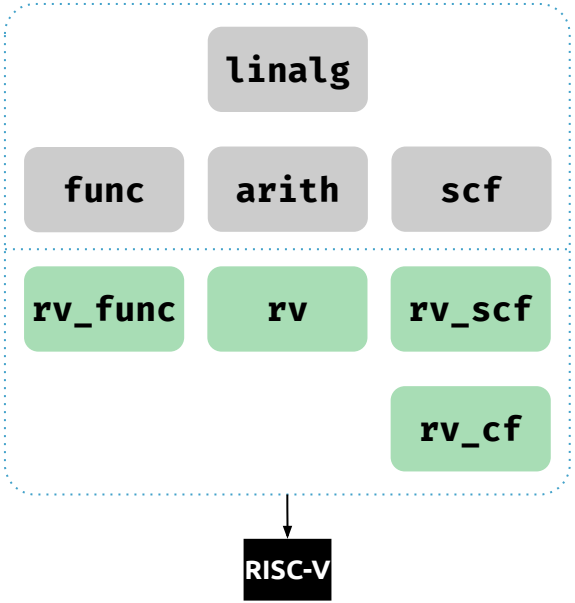
xDSL interpreter-based cost model

Targeting RISC-V-based Snitch Core

Selecting Unroll-And-Jam Parameters

Hardware-Aware Domain-Specific Transformations 🤸🎯🚄

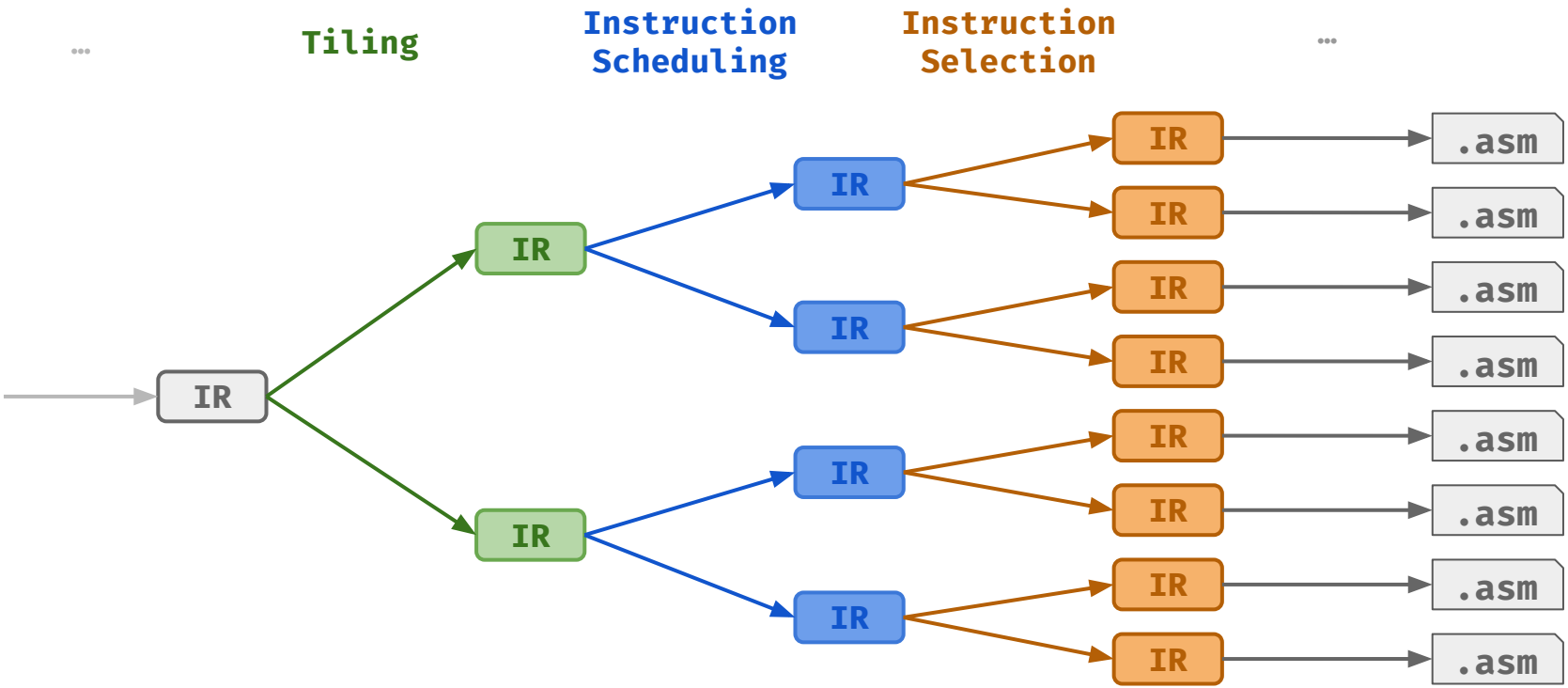
MLIR-Native



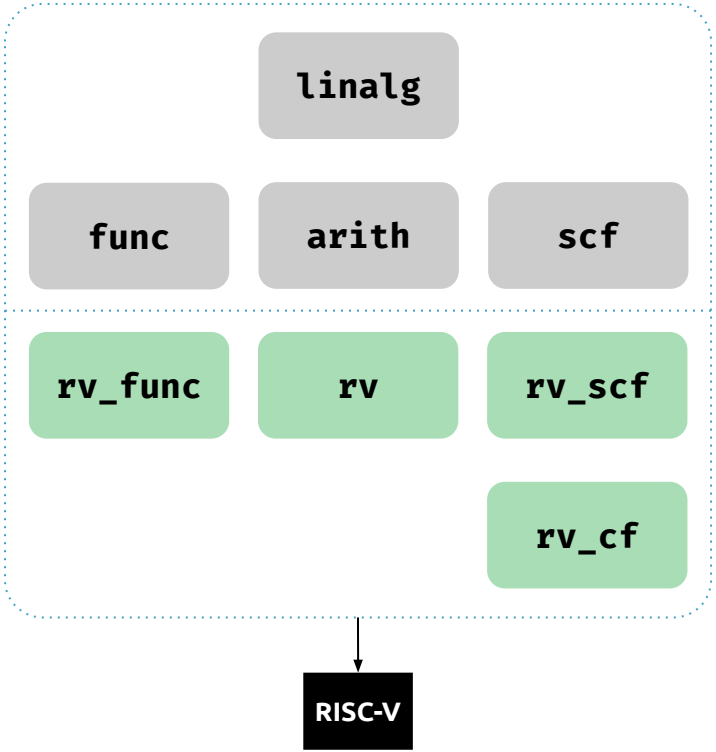
Leveraging Existing Cost Models

x86	ARM	RISC-V
uiCA	llvm-mca	GVSoC

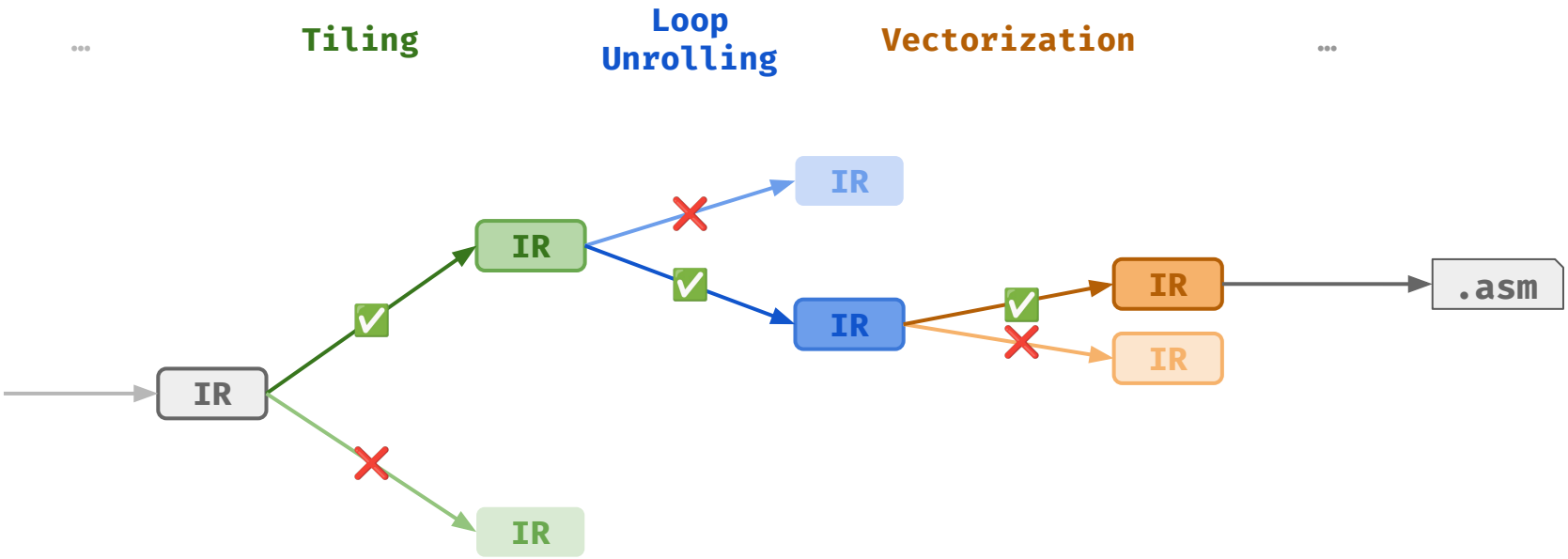
Earlier Compiler Passes Restrict Later Ones



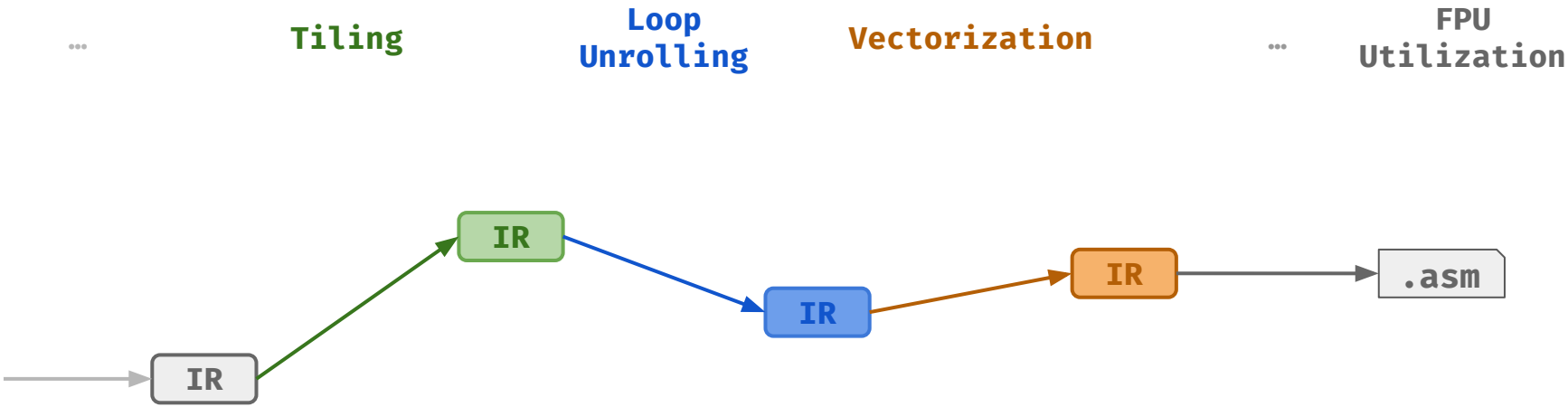
A Multi-Level RISC-V Backend



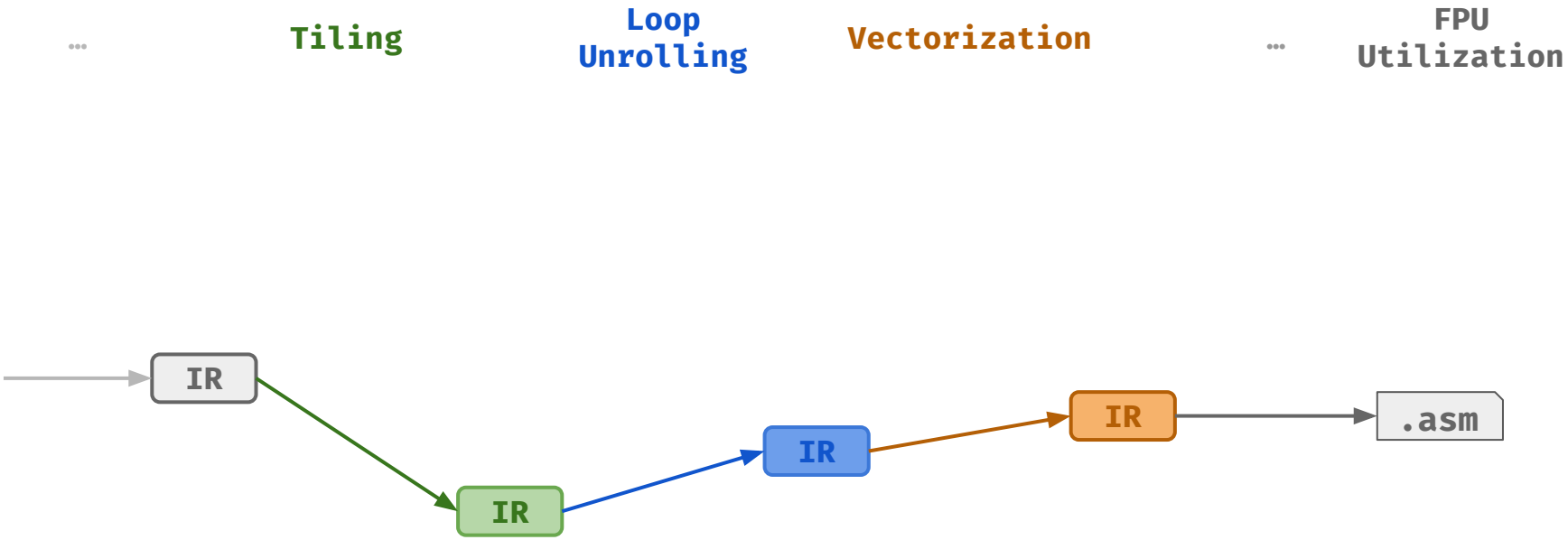
… Making Choices at Multiple Levels …



Schedule A


...
Tiling
Loop Unrolling
Vectorization
...
FPU Utilization
IR
IR
IR
IR
.asm

Schedule B


...
Tiling
Loop Unrolling
Vectorization
...
FPU Utilization
IR
IR
IR
IR
.asm