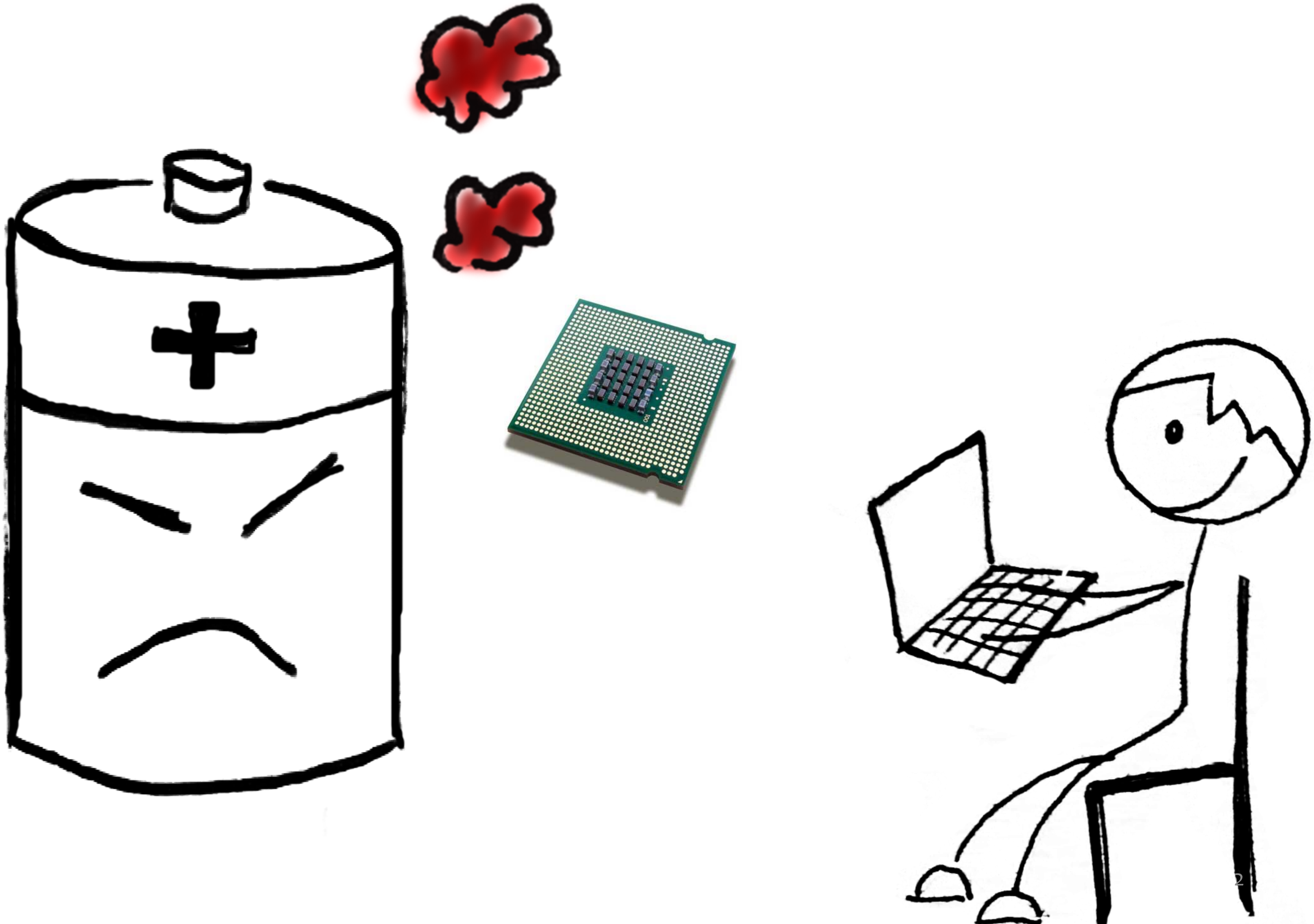
Chlorophyll

Synthesis-Aided Compiler for Low-Power Spatial Architectures

Phitchaya Mangpo Phothilimthana,

Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Ras Bodik

Berkeley
UNIVERSITY OF CALIFORNIA



unusual ISA

small memory

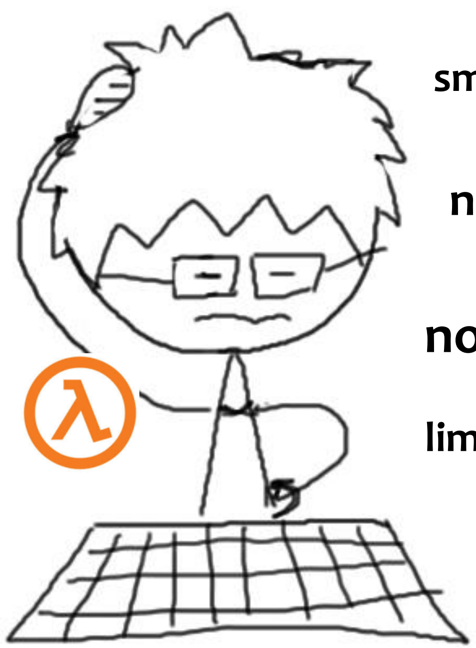
narrow bitwidth

no cache

limited interconnect

spatial & temporal partitioning

unusual ISA



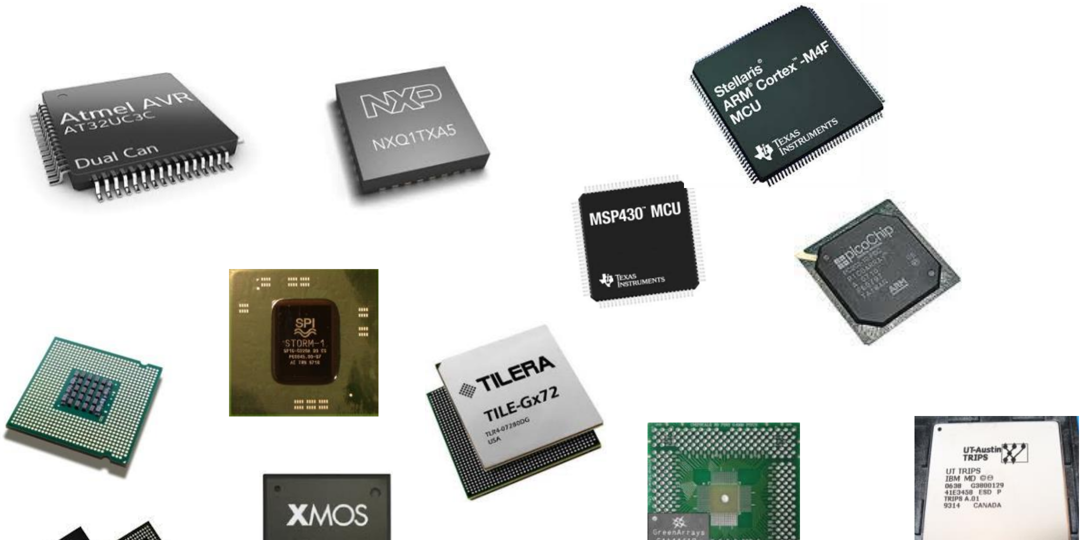
small memory

narrow bitwidth

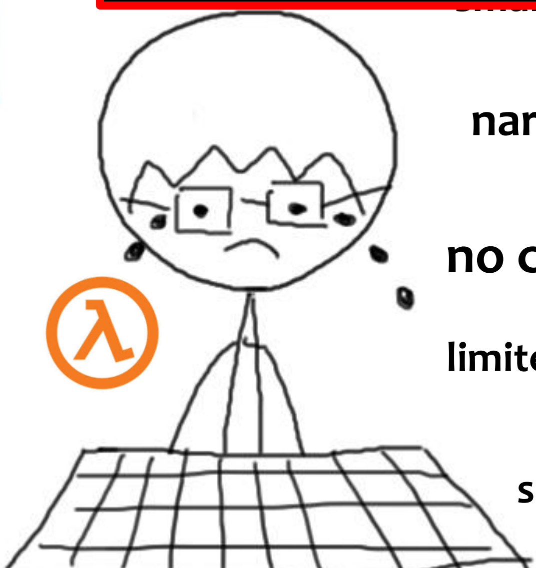
no cache

limited interconnect

spatial & temporal partitioning



Need a new way of building a compiler!



narrow bitwidth

no cache

limited interconnect

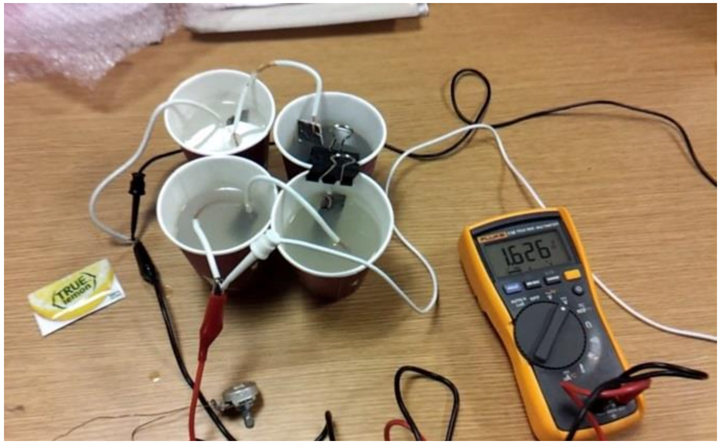
spatial & temporal partitioning

Synthesis-Aided Compiler

Classical vs. Synthesis Compiler

	Classical	Synthesis-Aided
Approach	Apply heuristic transformations	Find best program in defined search space
Required Components	• Transformations • Legality analysis • Heuristics	• Defined search space • Equivalence checker • Abstract cost function
Output's Performance	Depends on heuristic quality	Optimal in defined search space
Building Effort	High	Low

Case study: GreenArrays Spatial Processor



On FIR benchmark, *[Avizienis, Ljung]*

GA144 is 11x faster and simultaneously 9x more energy efficient than TI MSP 430.

Specs

- Stack-based 18-bit architecture
- 144 tiled cores
- Limited communication (neighbors only)
- No cache, no shared memory
- < 300 bytes of memory per core
- 32 instructions

Example challenges of programming spatial architectures like GA144:

- *Bitwidth slicing:* Represent 32-bit numbers by two 18-bit words
- *Function partitioning:* Break functions into a pipeline with just a few operations per core.

Our Contributions

- Spatial programming model
  - Flexible control over partitioning
- Low-effort approach to compiler construction
  - Solved a compilation problem as a synthesis problem
  - To scale synthesis, decomposed it into subproblems
- Empirical evaluation
  - Easy-to-build compiler architecture
  - Performance within 2x of expert-written code

Compiler Workflow

Spatial Program

Partitioner — constraint solving minimizing # of msgs

Program + partition annotation **(logical cores)**

Layout — Quadratic Assignment Problem

Program + location annotation & routing info **(physical cores)**

Code Separator — a traditional transformation

Per-core code with **communication code**

Code Generator — superoptimization

Per-core optimized **machine code**

Spatial programming model

Spatial Program

Partitioner

Program + partition annotation
(logical cores)

Layout

Program + location annotation
& routing info (physical cores)

Code Separator

Per-core code
with communication code

Code Generator

Per-core optimized machine code

Spatial programming model

```
int a, b;
int ans = a * b;
```

How does one want to program a spatial architecture?

1. Write algorithm in high-level language without dealing with low-level details

2. Have control over partitioning of data and computation if desired

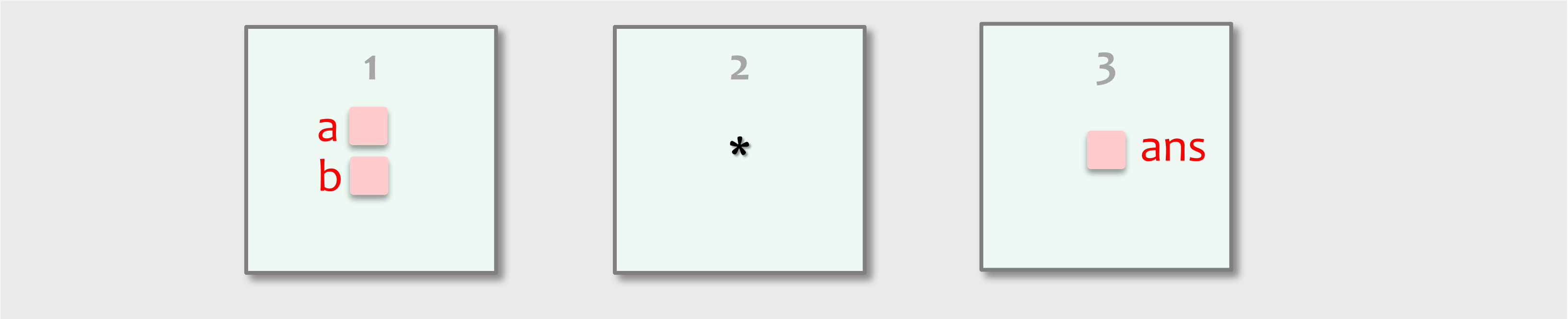
Spatial programming model

```
int@1 a, b;
int@3 ans = a *@2 b;
```

Partition Type

pins data and operators to specific partitions (logical cores)

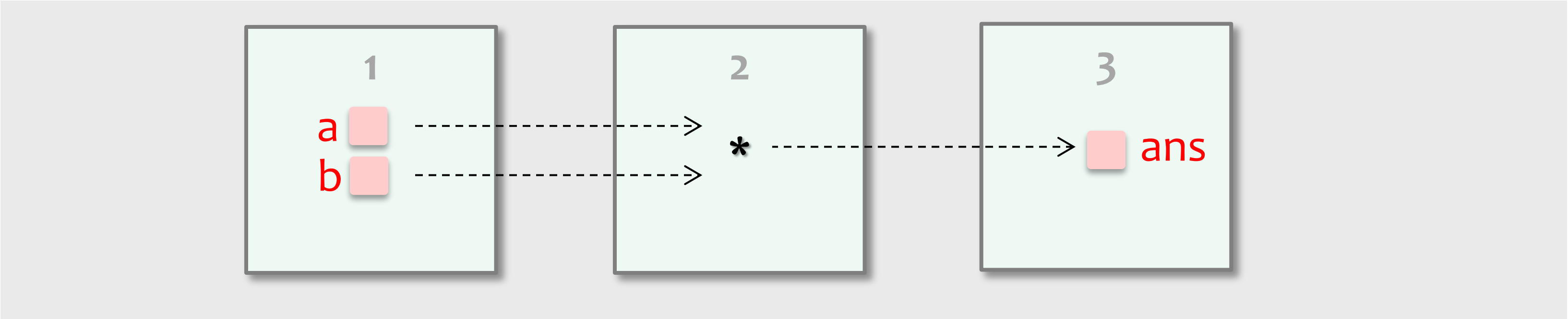
Similar to [Chandra *et al.* PPoPP'08]



Spatial programming model

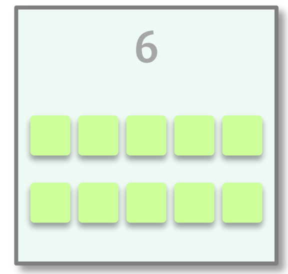
```
int@1 a, b;
int@3 ans = a *@2 b;
```

Do not need to handle data routing and communication code

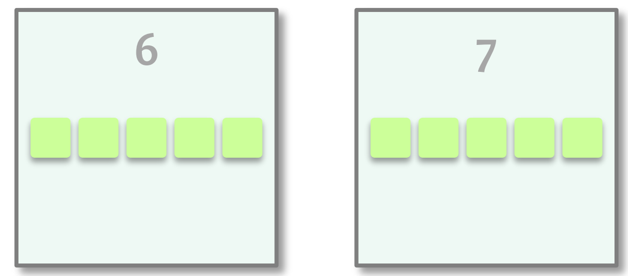


Distributed Partition Type

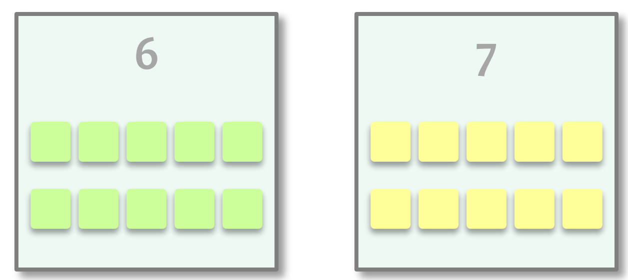
int@6 k[10];



int@{[0:5]=6, [5:10]=7} k[10];



int::2@{[0:10]=(6,7)} k[10];



Unspecified Partitions

How to compile a partially annotated program?

```
int a, b;
int@3 ans = a * b;
```

Unspecified Partitions

How to compile a partially annotated program?

```
int@?? a, b;
int@3 ans = a *@?? b;
```

Partitioning Synthesizer

Program

Partitioner

constraint solving to minimize # of msgs

Program + partition annotation (logical cores)

Layout

Program + location annotation & routing info (physical cores)

Code Separator

Per-core code with communication code

Code Generator

Per-core optimized machine code

How Does Partitioning Synthesizer Work?

Idea: infer partition types subject to

- Communication count constraint

  # of messages is minimized

- Space constraint

  code and data fit in each core

How: use Rosette (by Emina Torlak, Session 9A)

- Implement a type checker
- Get type inference for free

Layout Synthesizer

Program

Partitioner

Program + partition annotation
(logical cores)

Layout

Quadratic Assignment Problem

- Simulated annealing

Program + location annotation
& routing info (physical cores)

Code Separator

Per-core code
with communication code

Code Generator

Per-core optimized machine code

Code Separator

Program

Partitioner

Program + partition annotation
(logical cores)

Layout

Program + location annotation
& routing info (physical cores)

Code Separator a traditional transformation

Per-core code
with communication code

Code Generator

Per-core optimized machine code

Code Generator

Program

Partitioner

Program + partition annotation
(logical cores)

Layout

Program + location annotation
& routing info (physical cores)

Code Separator

Per-core code
with communication code

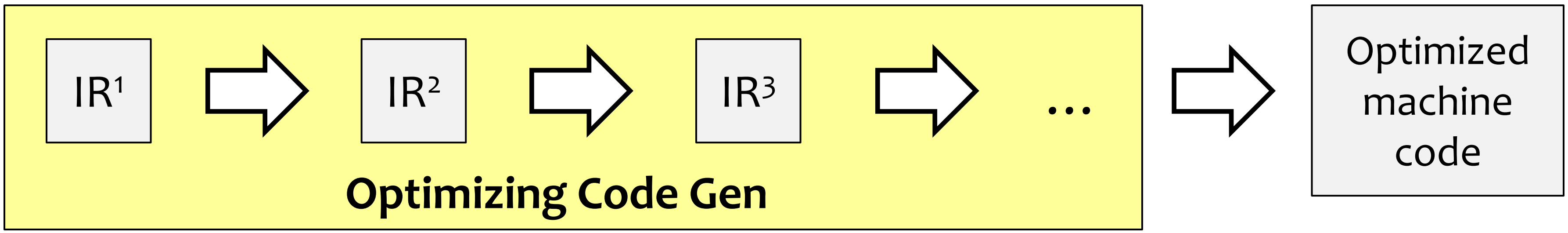
Code Generator

superoptimization

Per-core optimized machine code

Code Generator

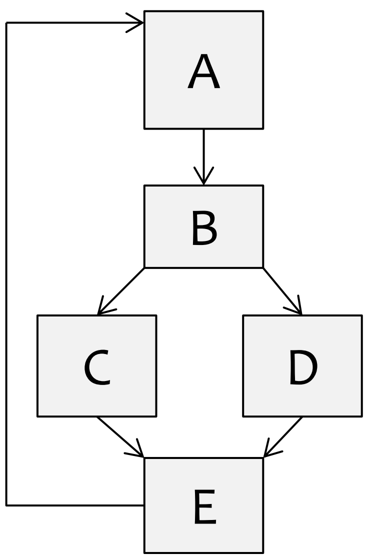
Classical compiler backend



Our compiler backend

Optimal program = minimum cost

Naïve Code Gen



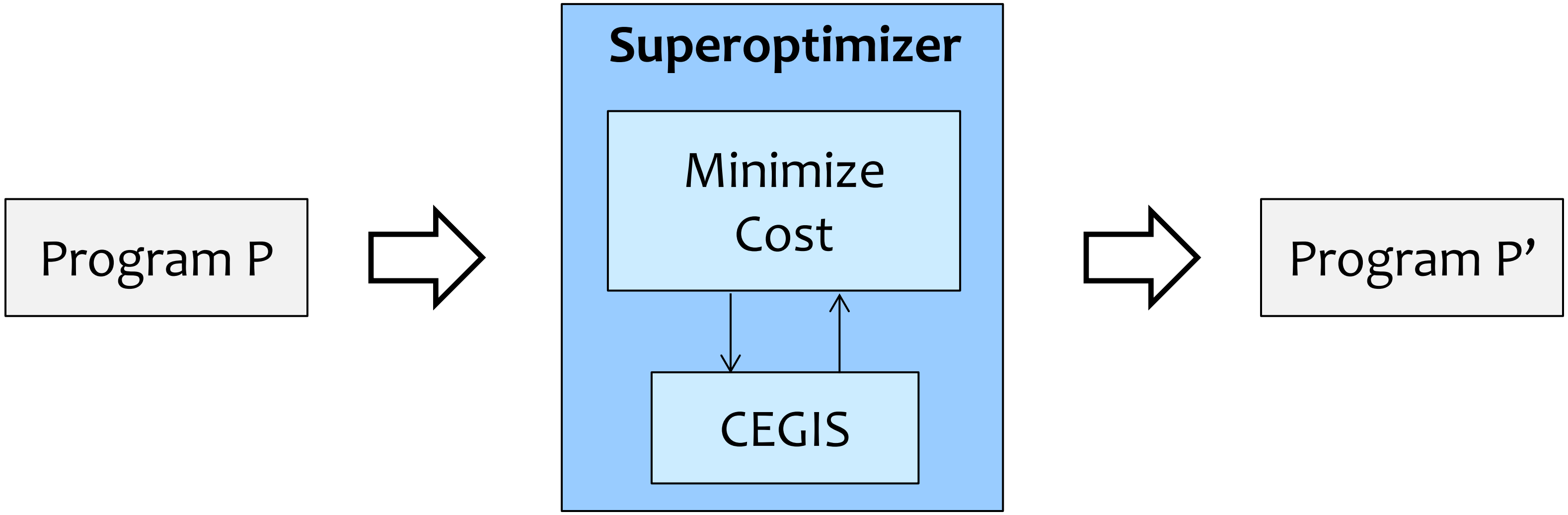
Machine code

Super-optimizer

[Massalin *et al*. ASPLOS'87, Bansal *et al*. ASPLOS'06, Gulwani *et al*. PLDI'11, …]

Super-optimized machine code

Superoptimizer

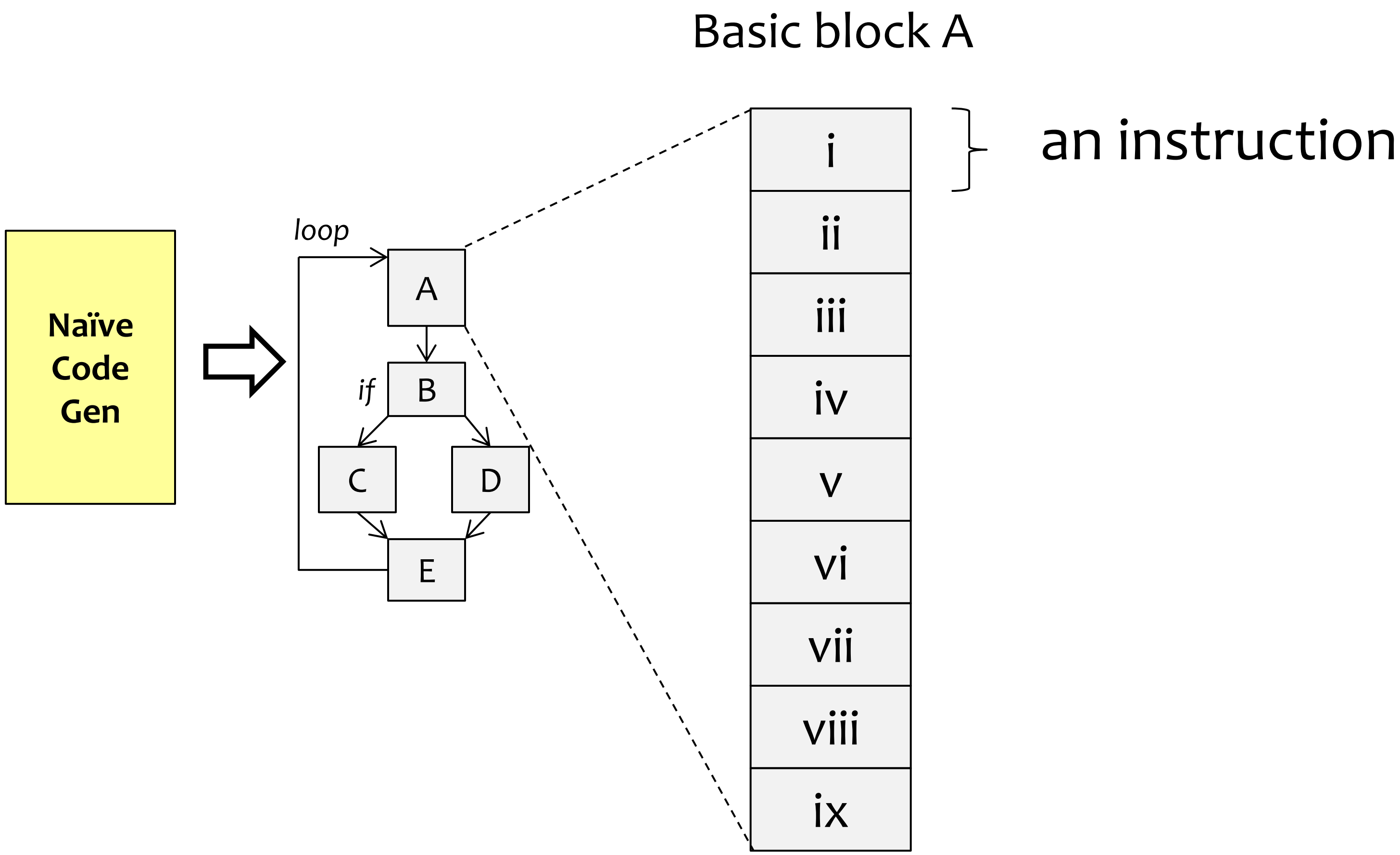


- Counter-Example-Guided Inductive Synthesis (CEGIS)
  - encode program as SMT formula
  - solve using Z3
- Minimizing one of:
  - Execution time
  - Energy consumption
  - Program length

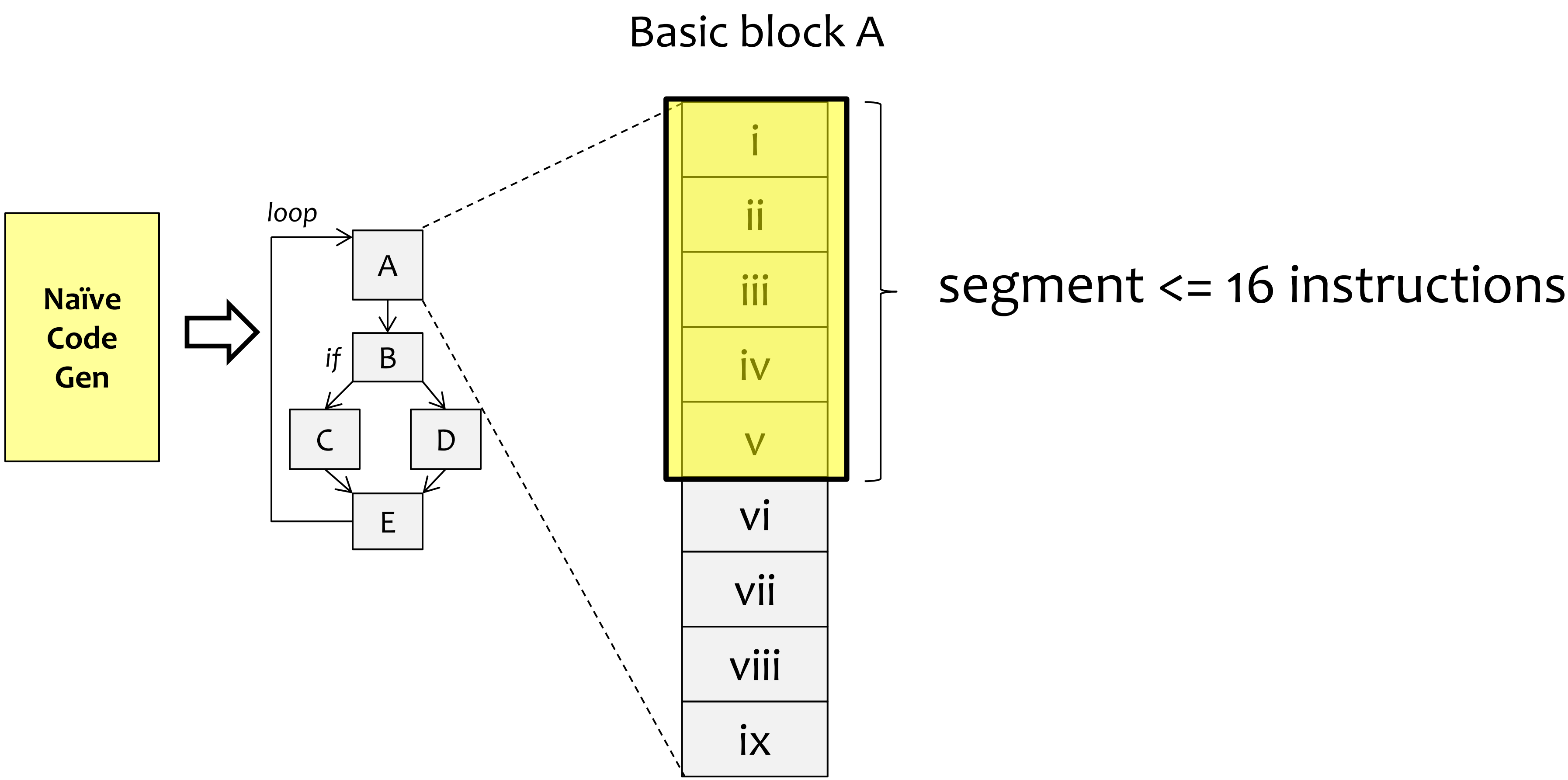
Problem with Superoptimizers

- Synthesizing the entire program is not scalable.
  - Start-of-the-art synthesizers can generate up to 25 instructions [Schkufza *et al*. ASPLOS13, Gulwani *et al*. PLDI'11].
- Must decompose the superoptimization.

Modular Superoptimizer

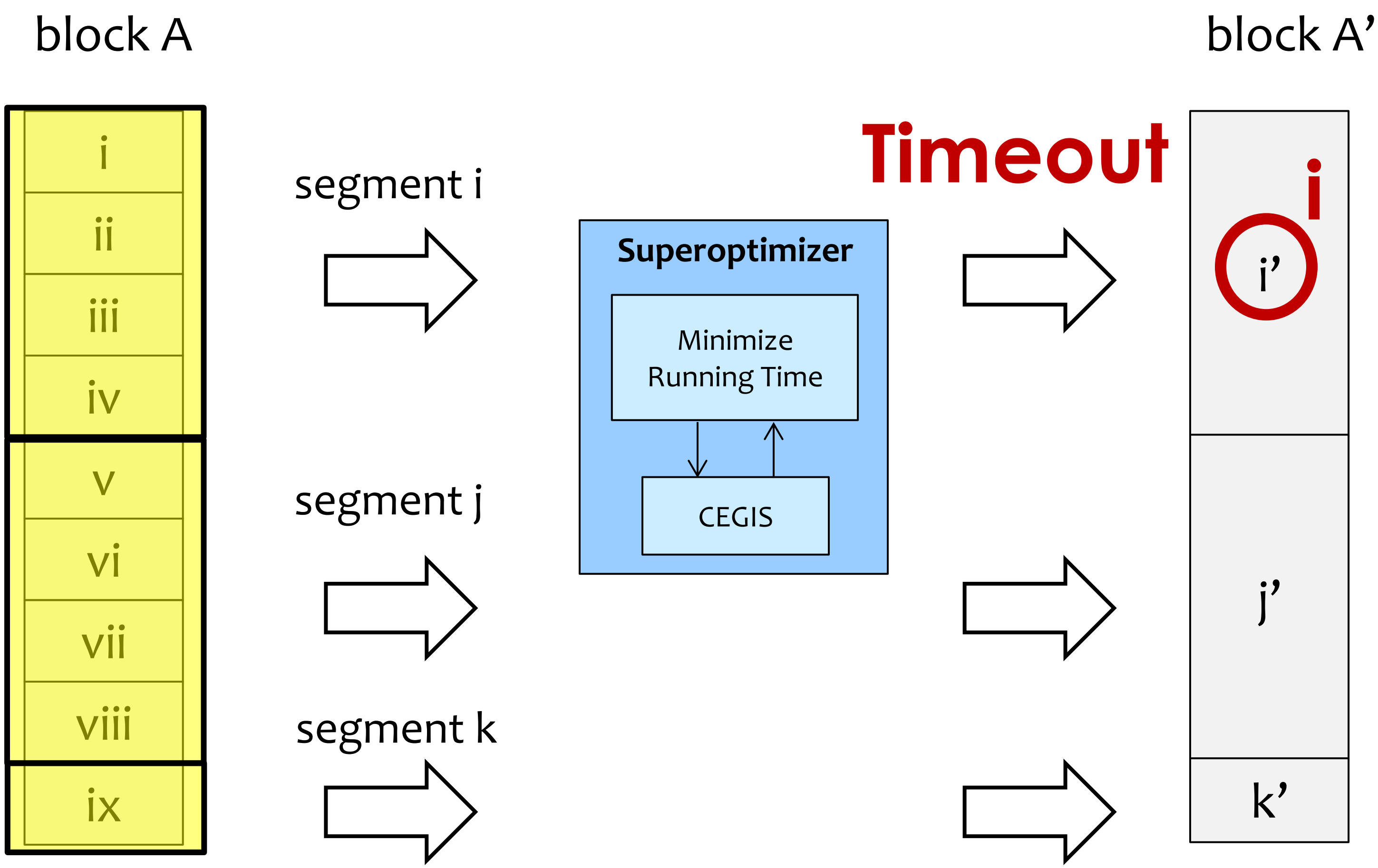


Modular Superoptimizer



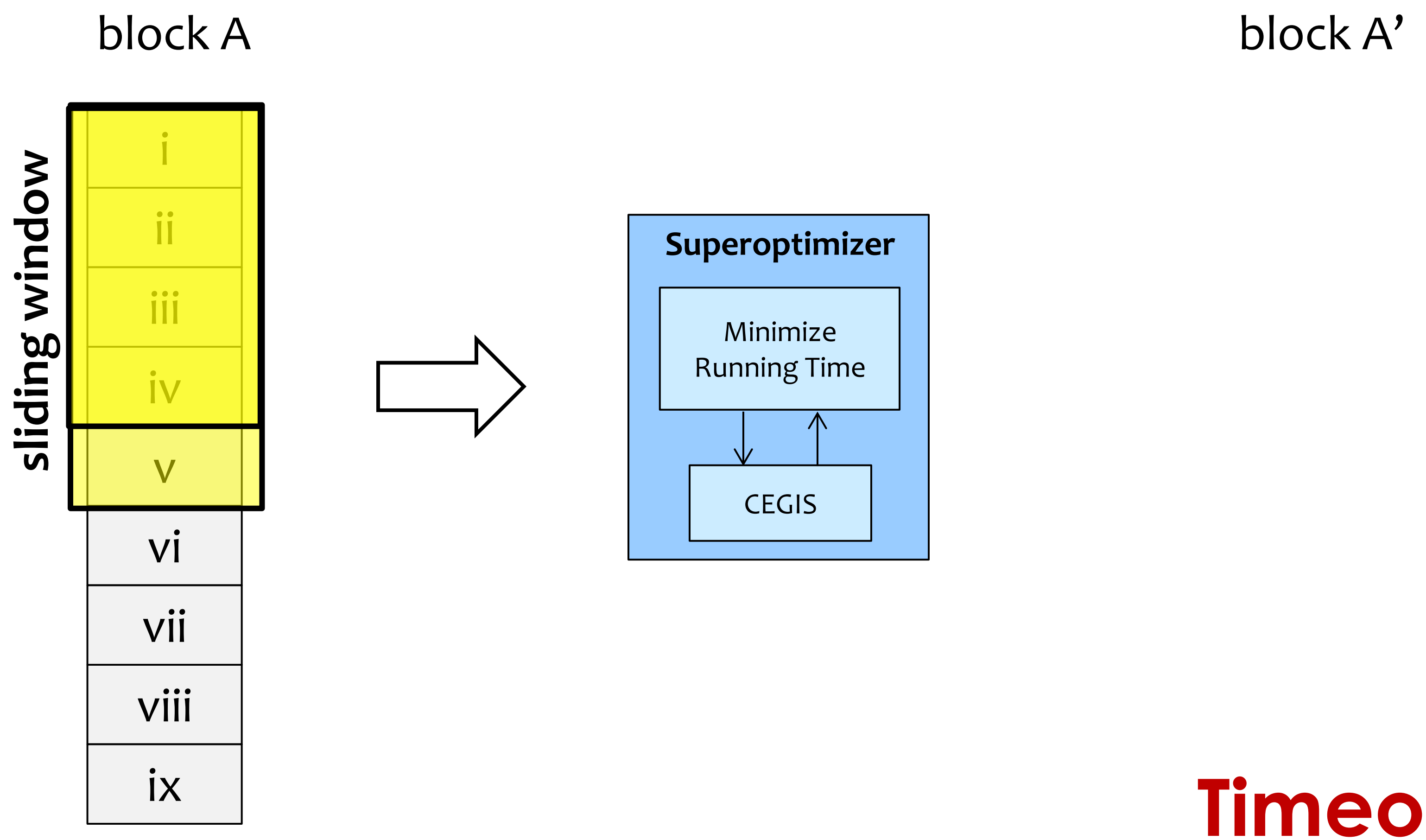
Naïve Way to Decompose

Fixed Windows



A Better Way

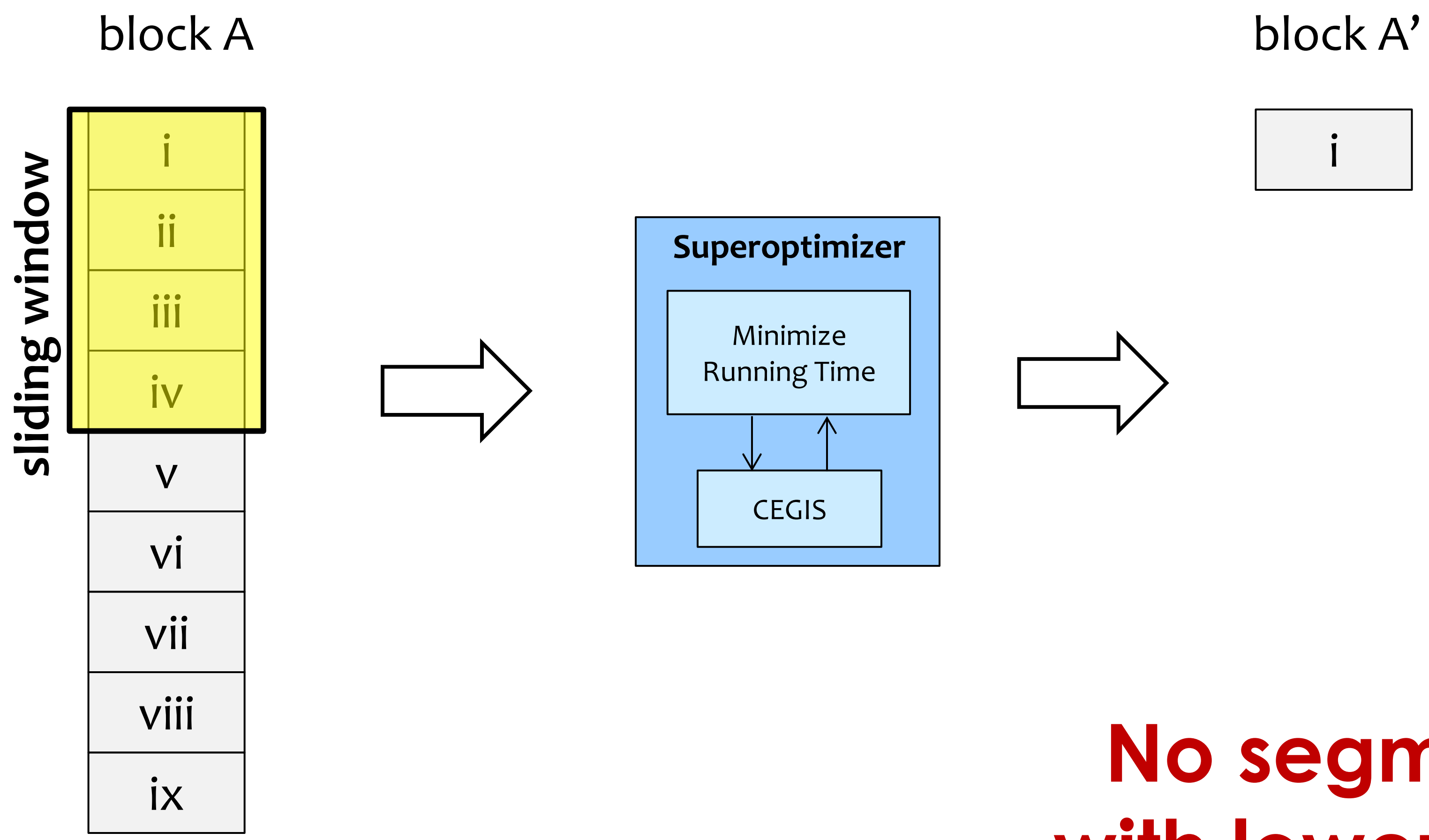
Sliding Window



Timeout

A Better Way

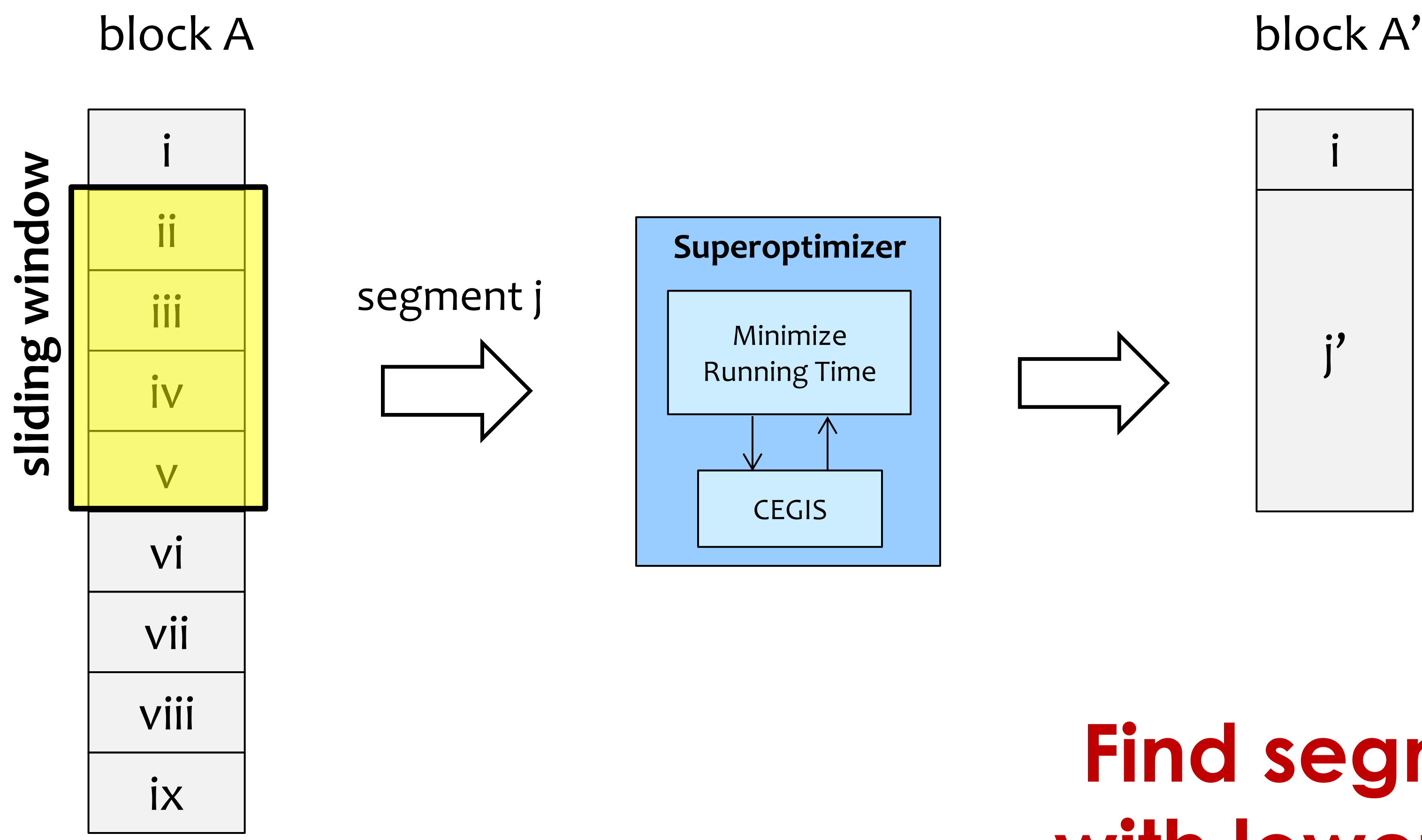
Sliding Window



No segment with lower cost

A Better Way

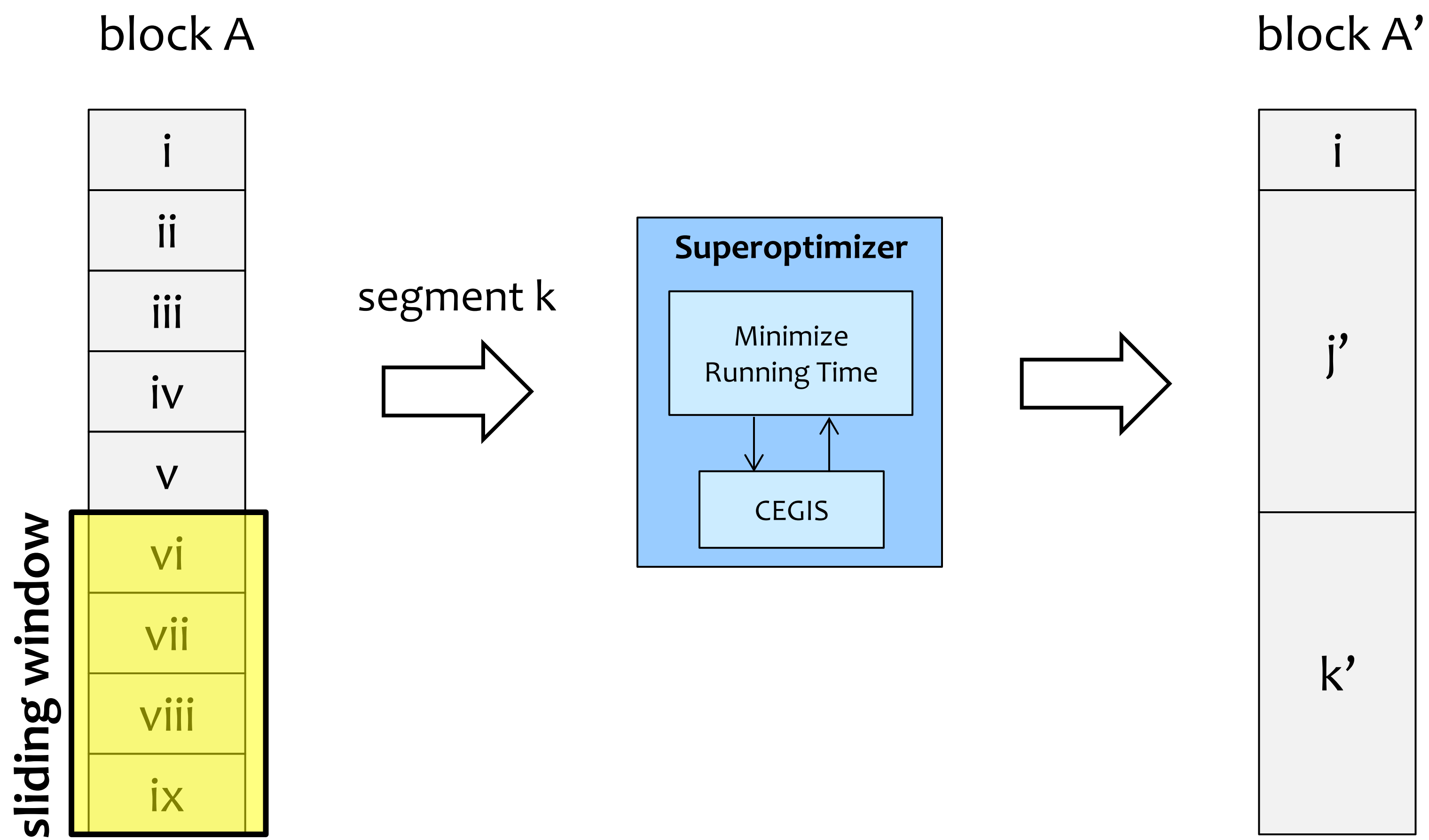
Sliding Window



Find segment with lower cost

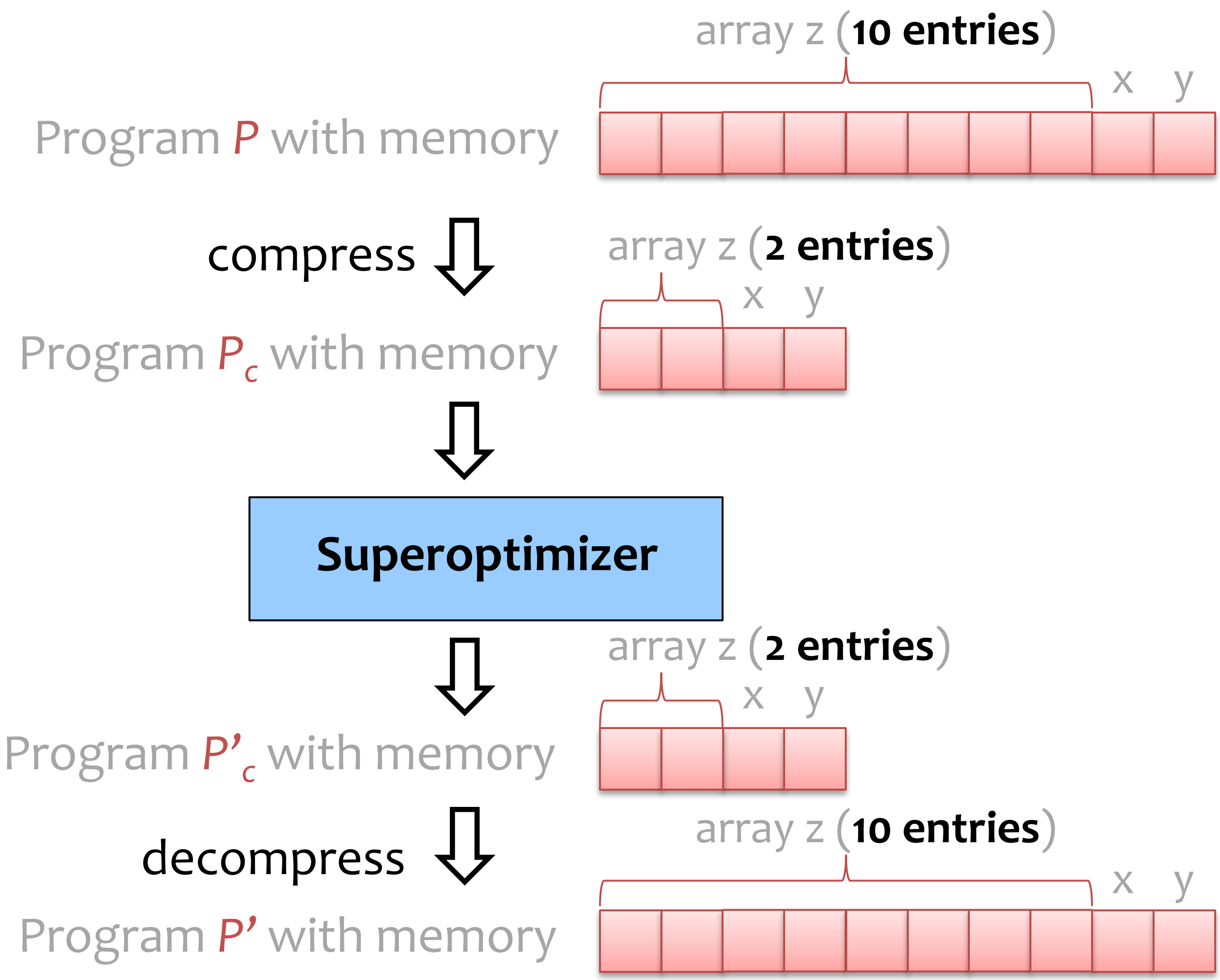
A Better Way

Sliding Window



Address Space Compression

Trick to speed up synthesis time



Then verify if $P \equiv P'$

Empirical Evaluation

Hypothesis 1

Synthesis generates faster code than a heuristic compiler.

Synthesizing partitioner	vs.	Heuristic partitioner
		a greedy algorithm

Empirical Evaluation

Hypothesis 1

Synthesis generates faster code than a heuristic compiler.

Synthesizing partitioner	vs.	Heuristic partitioner
Precise layout	vs.	Less precise layout assumes each message is sent once

Empirical Evaluation

Hypothesis 1

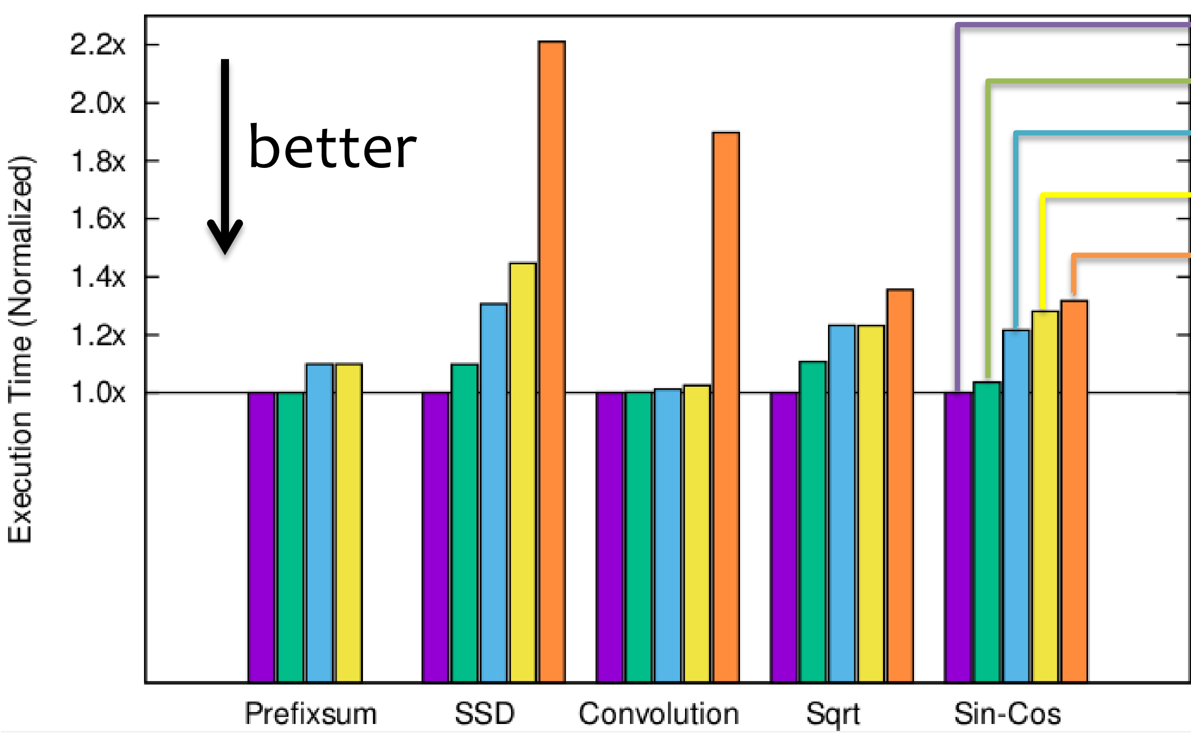
Synthesis generates faster code than a heuristic compiler.

Synthesizing partitioner	vs.	Heuristic partitioner
Precise layout	vs.	Less precise layout
Superoptimizer	vs.	No superoptimizer
Sliding window	vs.	Fixed window

Empirical Evaluation

Hypothesis 1

Synthesis generates faster code than a heuristic compiler.



+ precise layout
→ up to 1.8x speedup

+ synthesizing partitioner
→ 5% speedup

+ superoptimizer
→ 15% speedup

(4% from sliding window)

Empirical Evaluation

Hypothesis 2

Our compiler produces code comparable to the expert's code.

On MD5 benchmark, the expert uses many advanced tricks:

- 10 cores
- Self-modifying code
- Circular array data structure
- Different modes of operations for different cores
  - Instruction fetch from local memory
  - Instruction fetch from neighbors

We define success to be within 2x of the expert's code.

Empirical Evaluation

Hypothesis 2

Our compiler produces code comparable to the expert's code.

On 4 smaller benchmarks, Chlorophyll was on average

- 46% slower
- 44% less energy-efficient

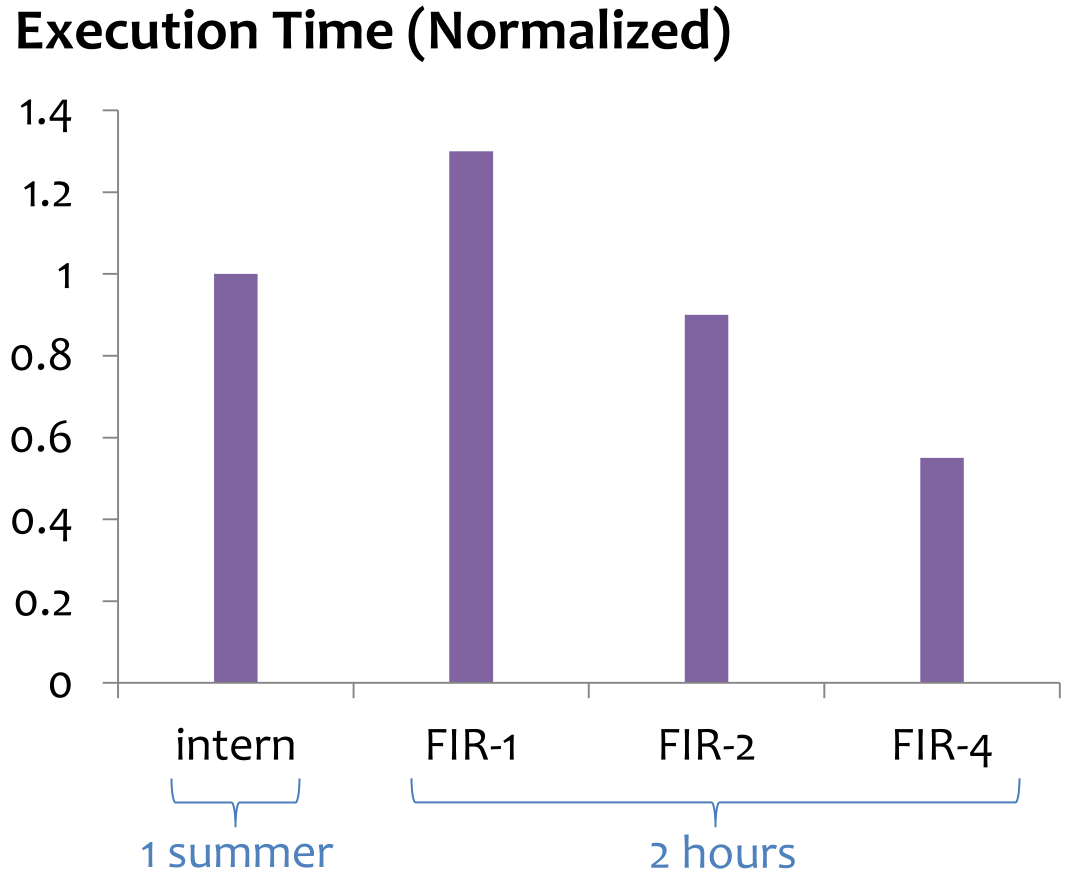
On a larger benchmark (MD5), Chlorophyll was

- 65% slower
- 69% less energy-efficient

Empirical Evaluation

Hypothesis 3

Chlorophyll increases programmer productivity
and offers the ability to explore different implementations quickly.



Using program synthesis as a core compiler building block enables us to build a new compiler with low effort that still produces efficient code.

Thank you