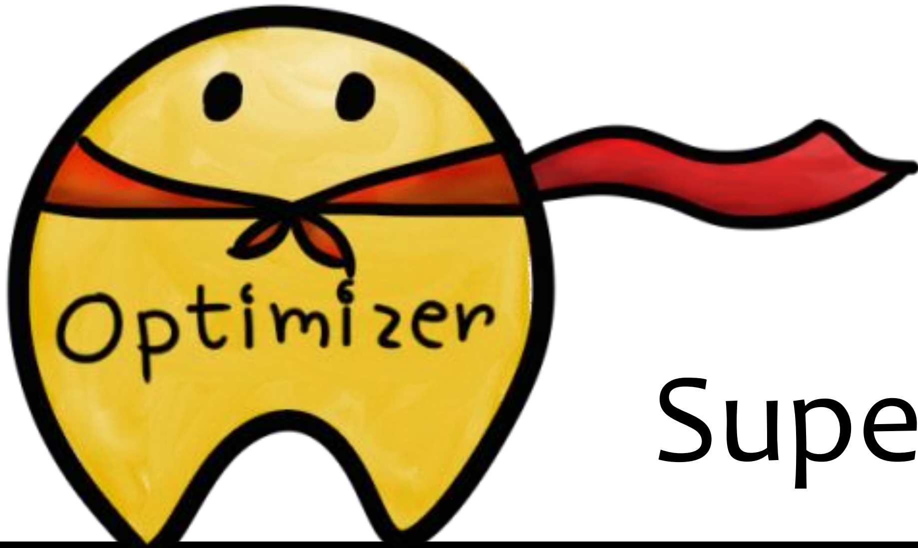
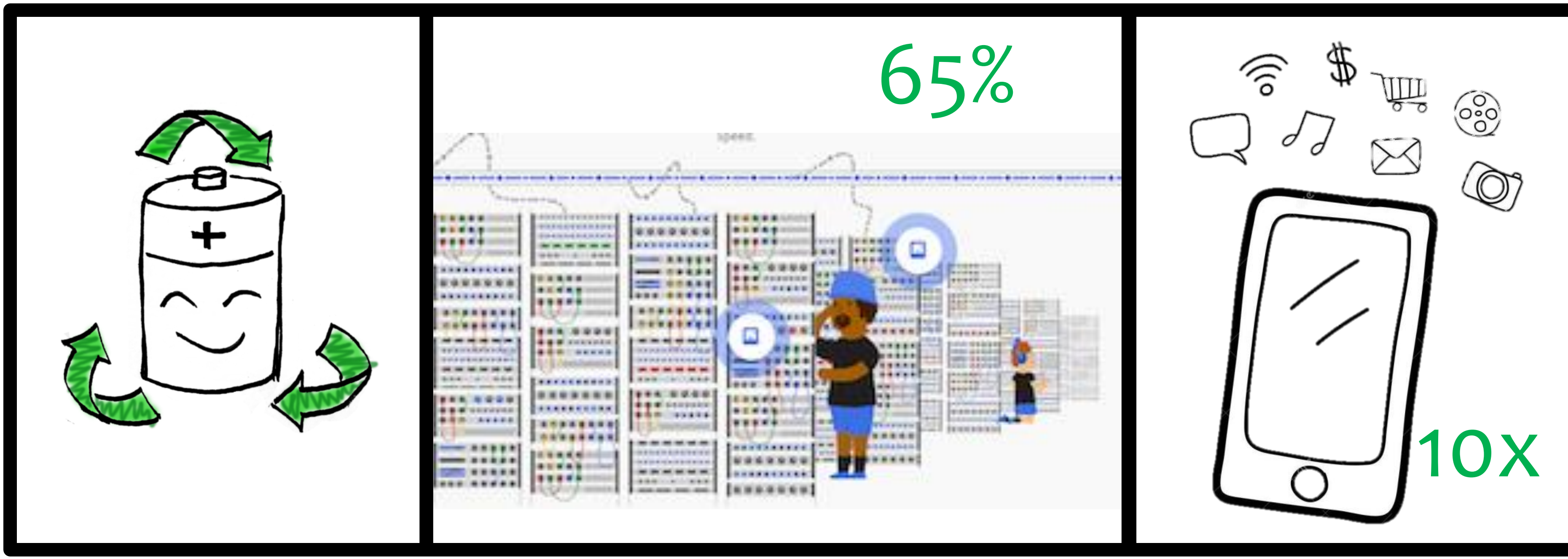


Scaling Up Superoptimization

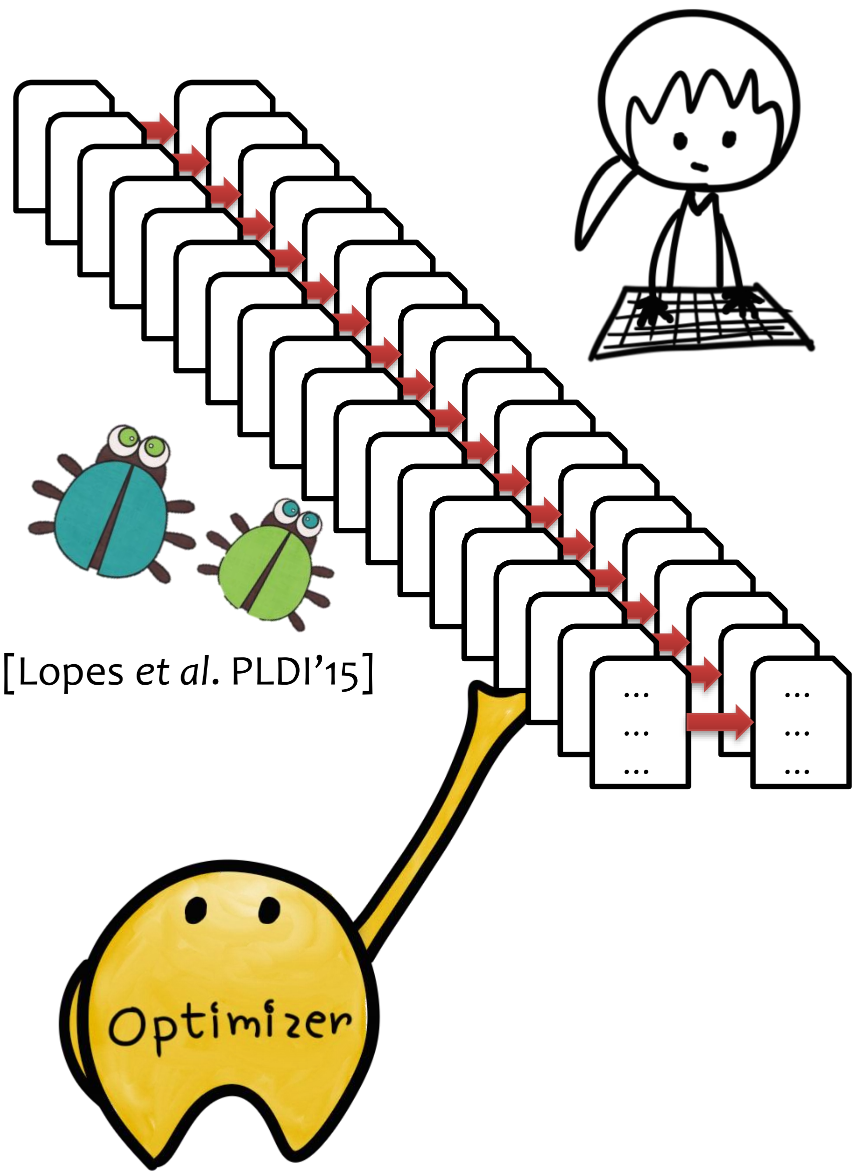
Phitchaya Mangpo Phothilimthana (UC Berkeley)
Aditya Thakur (Google)
Rastislav Bodik (University of Washington)
Dinakar Dhurjati (Qualcomm Research)




```
inst1 ...
inst2 ...
inst3 ...
inst4 ...
inst5 ...
```

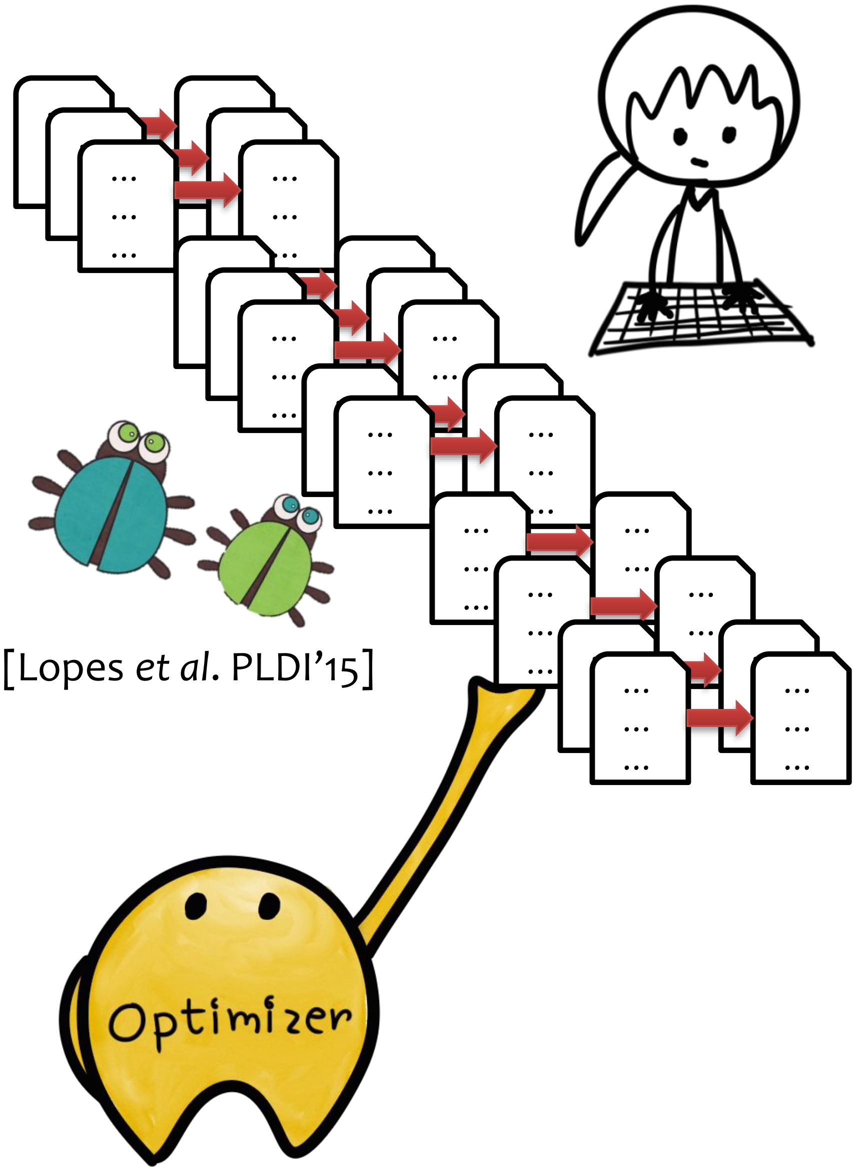

```
inst1' ...
inst2' ...
inst3' ...
inst4' ...
```

Rewrite Rules



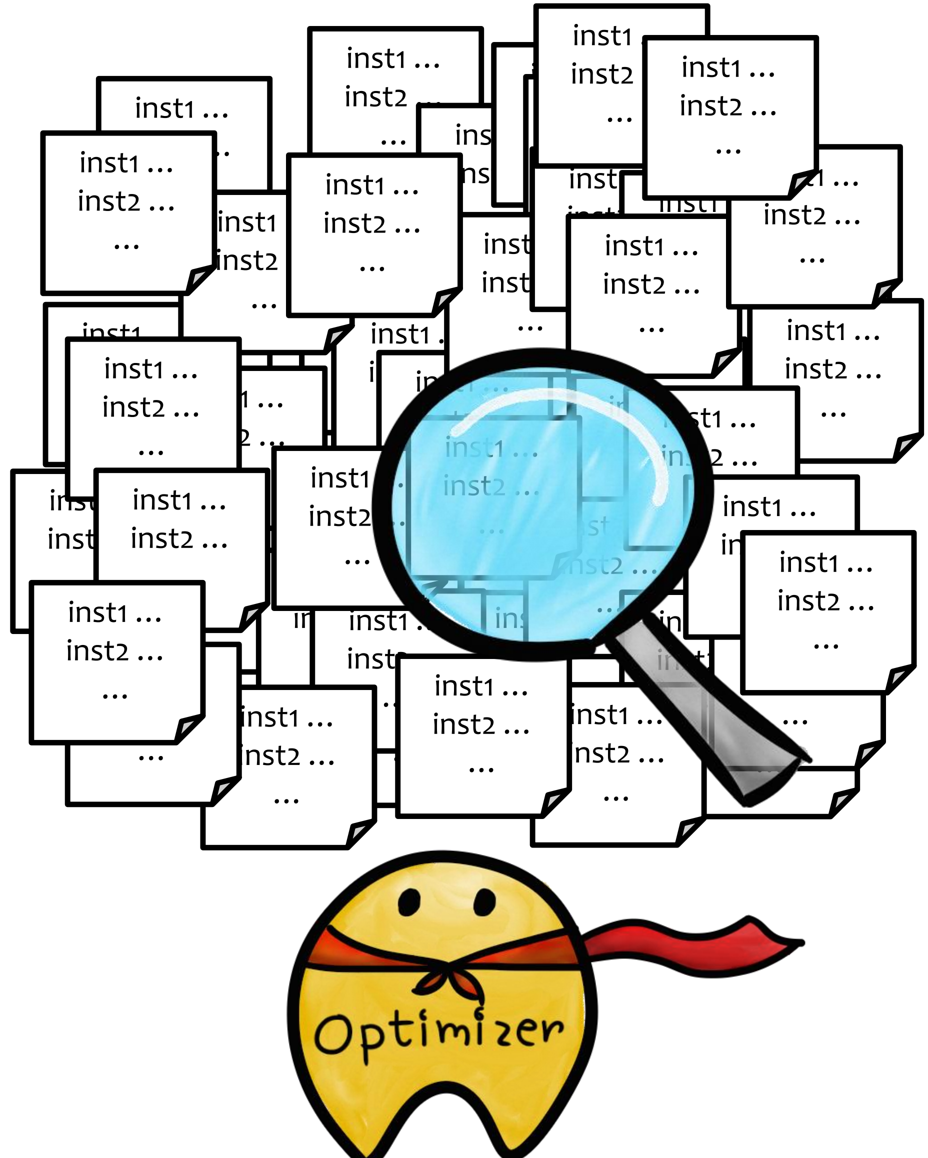
[Lopes *et al.* PLDI'15]

Rewrite Rules



[Lopes *et al.* PLDI'15]

Search across all possible programs

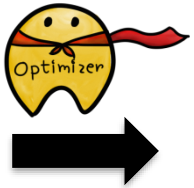


ARM

register-based ISA

gcc –O3

```
cmp   r1, #0
mov   r3, r1, asr #31
add   r2, r1, #7
mov   r3, r3, lsr #29
movge r2, r1
ldrb  r0, [r0, r2, asr #3]
add   r1, r1, r3
and   r1, r1, #7
sub   r3, r1, r3
asr   r1, r0, r3
and   r0, r0, #1
```



82% speedup

```
asr   r3, r1, #2
add   r2, r1, r3, lsr #29
ldrb  r0, [r0, r2, asr #3]
and   r3, r2, #248
sub   r3, r1, r3
asr   r1, r0, r3
and   r0, r1, #1
```

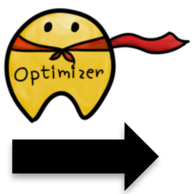
GreenArrays

stack-based ISA

Expert's

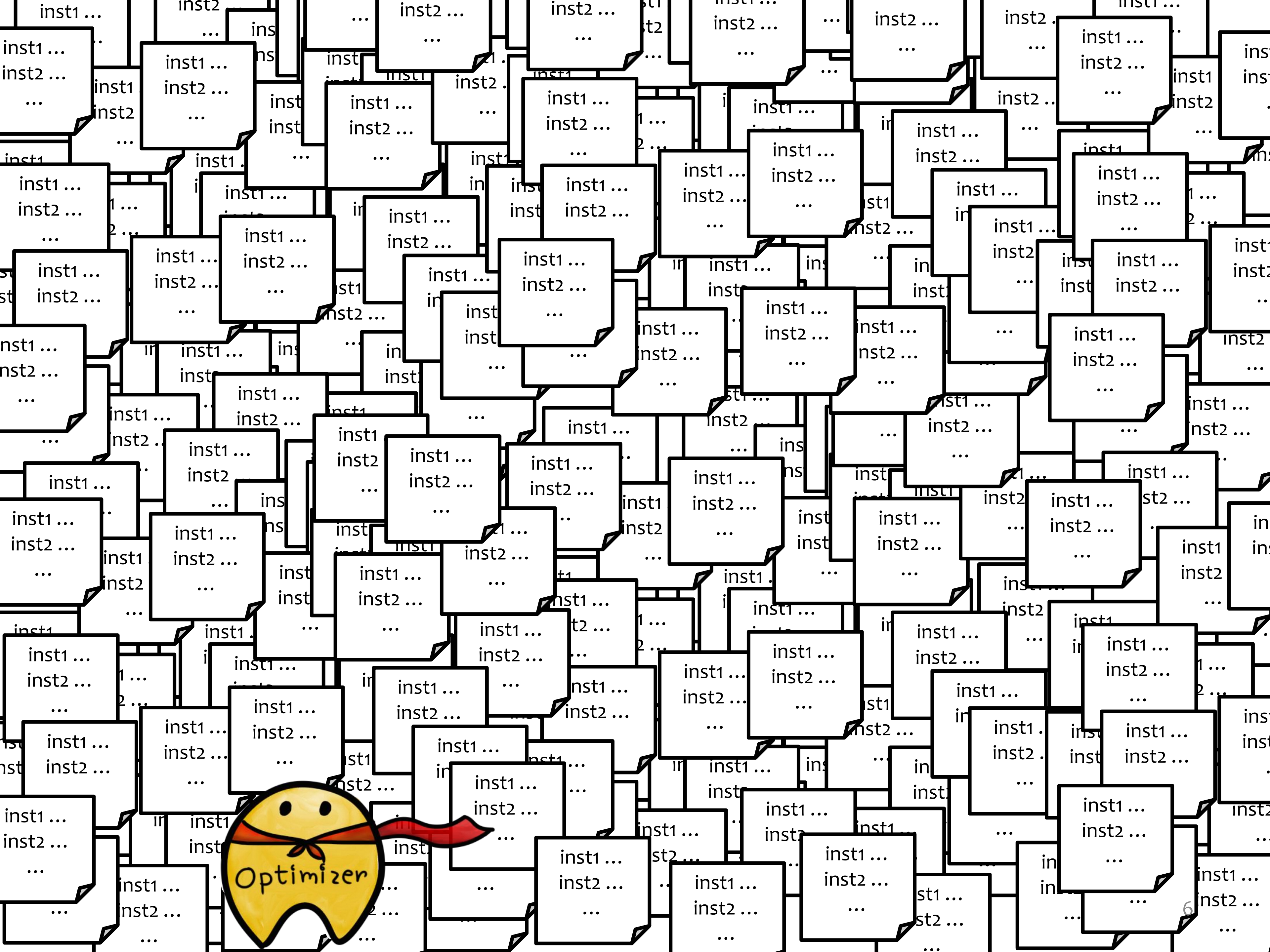
```
push over - push and
pop pop and over
0xffff or and or
```

Precondition: top 3 elements in the stack are <= 0xffff



2.5X speedup

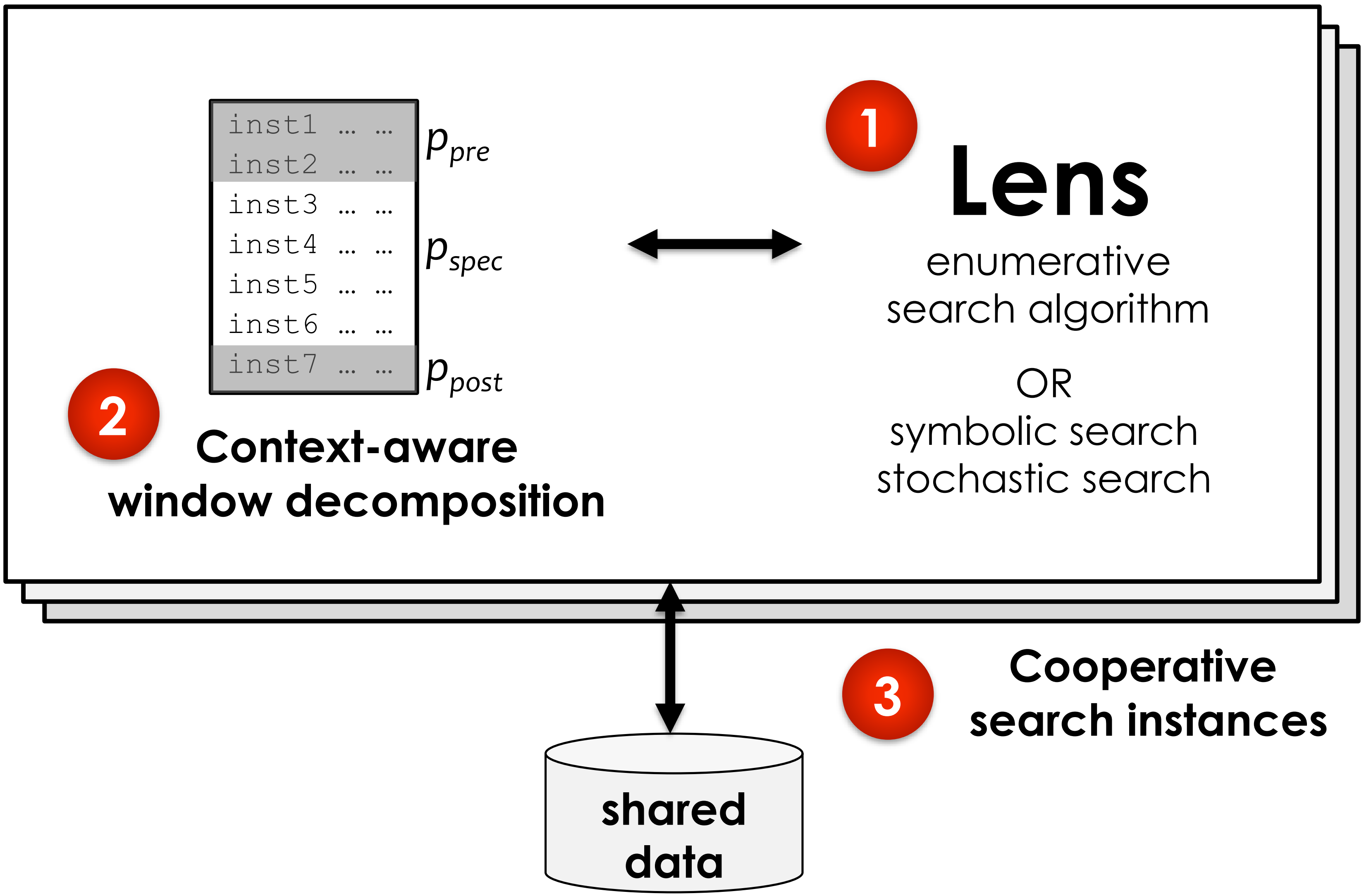
```
dup push or and pop or
```


inst1 ...
inst2 ...
...
Optimizer

Goal

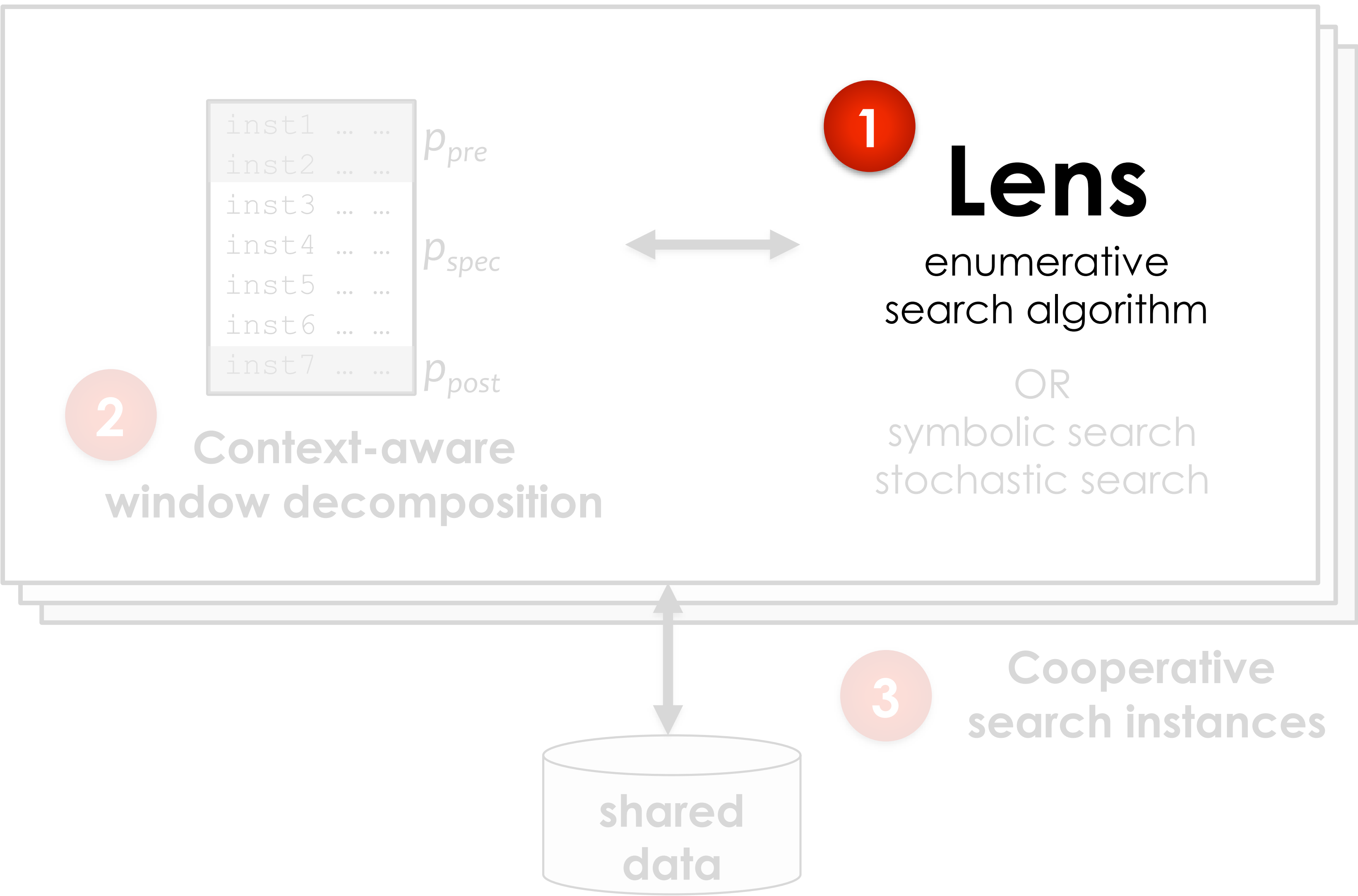
Develop a **search technique** that can synthesize optimal programs **faster and** more **consistently**.

We Develop...



inst1 … …
inst2 … …
inst3 … …
inst4 … …
inst5 … …
inst6 … …
inst7 … …
p_{pre}
p_{spec}
p_{post}
1
Lens
enumerative search algorithm
OR
symbolic search
stochastic search
2
Context-aware window decomposition
3
Cooperative search instances
shared data

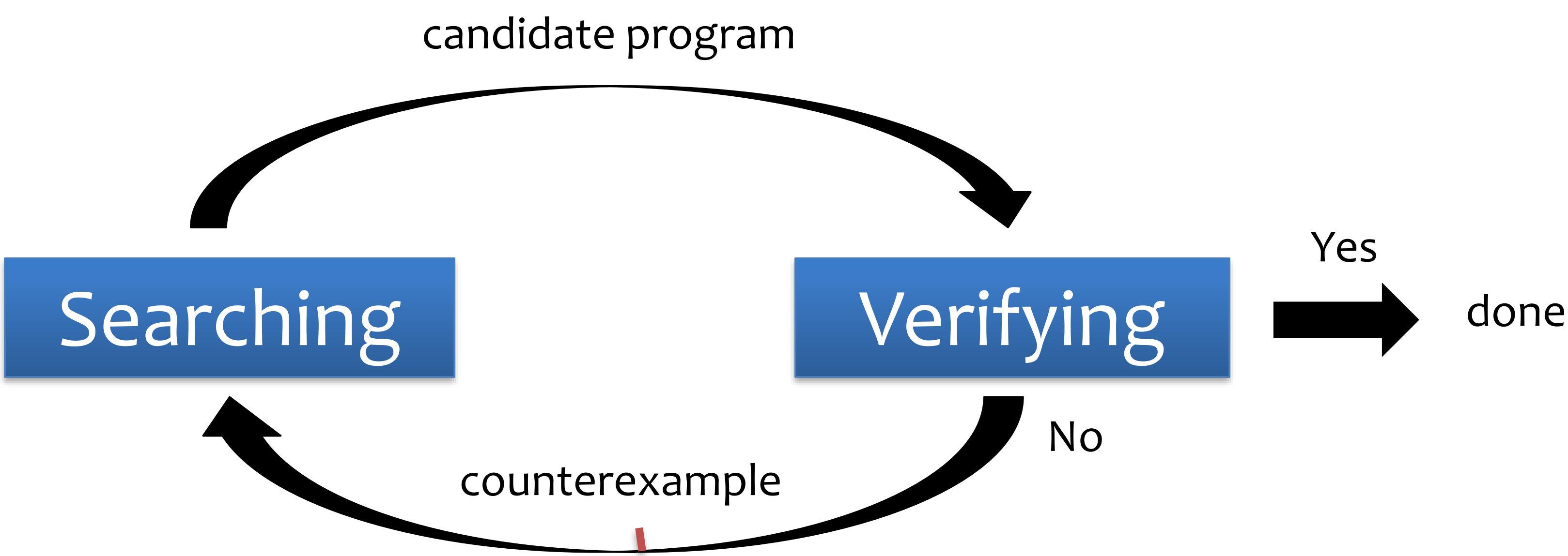
Lens



inst1 ... ...
inst2 ... ...
inst3 ... ...
inst4 ... ...
inst5 ... ...
inst6 ... ...
inst7 ... ...
p_{pre}
p_{spec}
p_{post}
1
Lens
enumerative
search algorithm
OR
symbolic search
stochastic search
2
Context-aware
window decomposition
3
Cooperative
search instances
shared
data

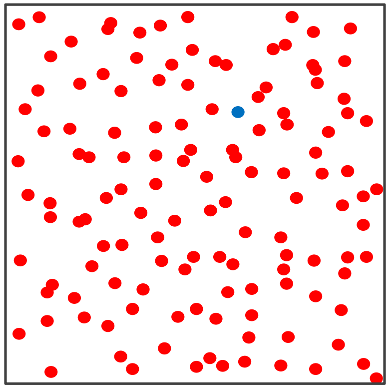
Inductive Synthesis

Find program $p \equiv p_{spec}$



Finds a program that passes all test cases

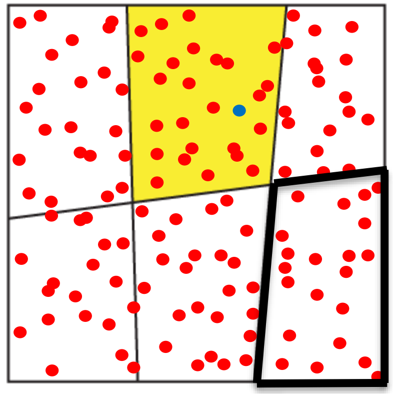
Checks equivalence on all inputs using a constraint solver



Search space of *k*-instruction long programs

- program $\boldsymbol{p} \equiv \boldsymbol{p}_{spec}$ (on all inputs)
- program $\boldsymbol{p} \not\equiv \boldsymbol{p}_{spec}$

n test cases

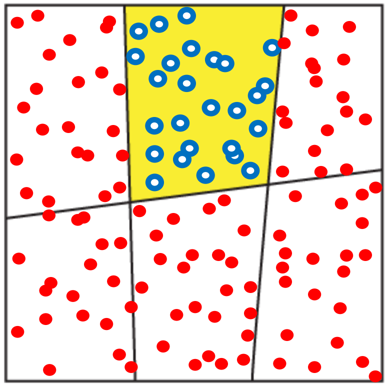


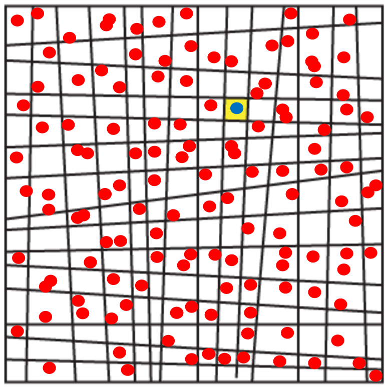
Equivalence class

Search space of *k*-instruction long programs

program $\boldsymbol{p} \equiv \boldsymbol{p_{spec}}$ (on all inputs)

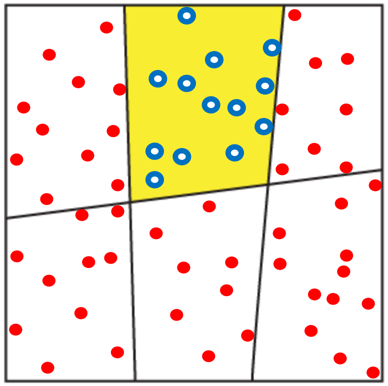
program $\boldsymbol{p} \not\equiv \boldsymbol{p_{spec}}$





Search space of *k*-instruction long programs

- ○ program $\boldsymbol{p}$ possibly $\equiv \boldsymbol{p}_{spec}$
- ● program $\boldsymbol{p} \equiv \boldsymbol{p}_{spec}$
- ● program $\boldsymbol{p} \not\equiv \boldsymbol{p}_{spec}$

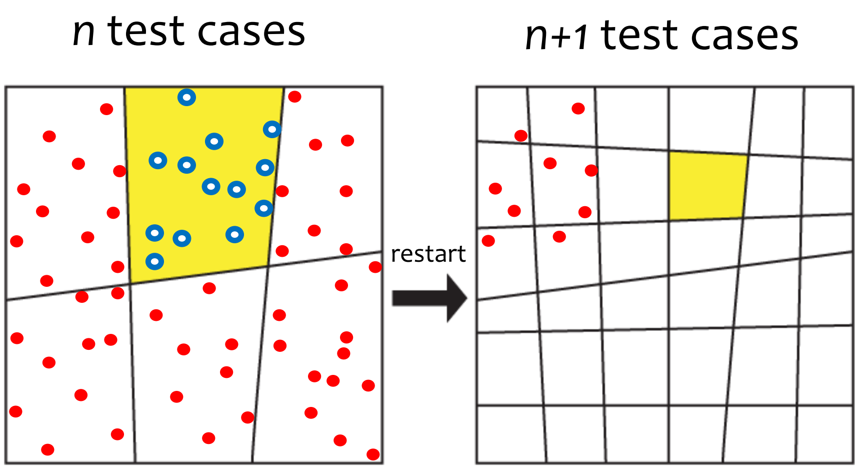


Existing techniques

[Barthe *et al.* PPoPP'13]

[Udupa *et al.* PLDI'13]

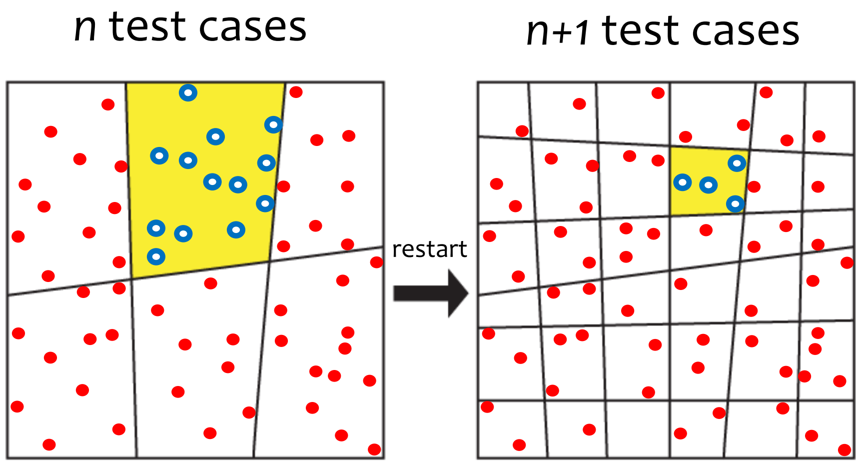
Existing techniques
[Barthe *et al.* PPoPP'13]
[Udupa *et al.* PLDI'13]



Inefficiency 1

Revisit programs that have been pruned away previously.

Existing techniques
[Barthe *et al.* PPoPP'13]
[Udupa *et al.* PLDI'13]
[Bansal *et al.* ASPLOS'06]



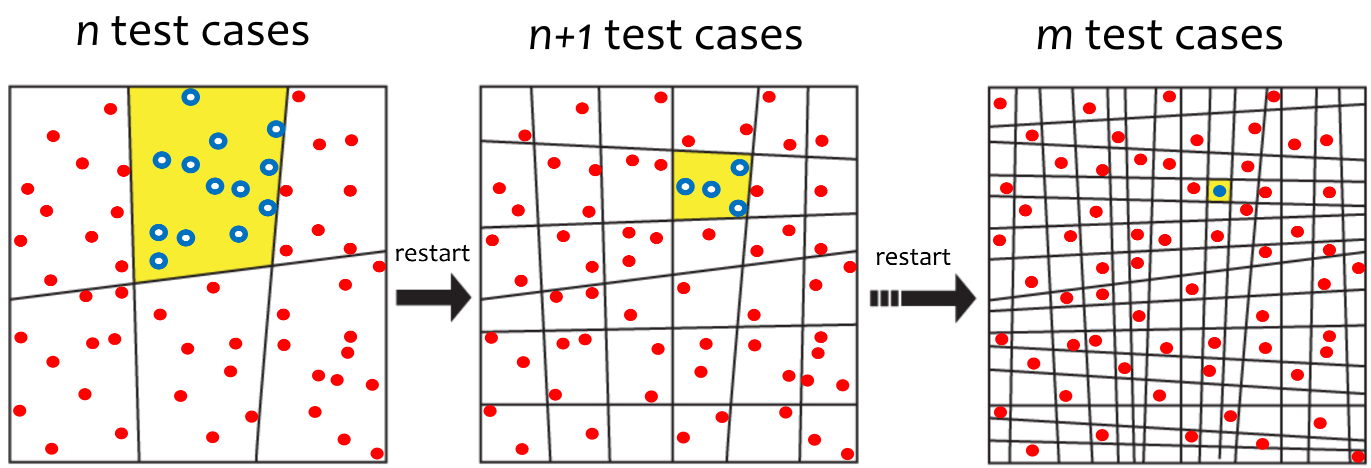
Inefficiency 1

Revisit programs that have been pruned away previously.

Inefficiency 2

Use more test cases than necessary.

Existing techniques
[Barthe *et al.* PPoPP'13]
[Udupa *et al.* PLDI'13]
[Bansal *et al.* ASPLOS'06]



Inefficiency 1

Revisit programs that have been pruned away previously.

Inefficiency 2

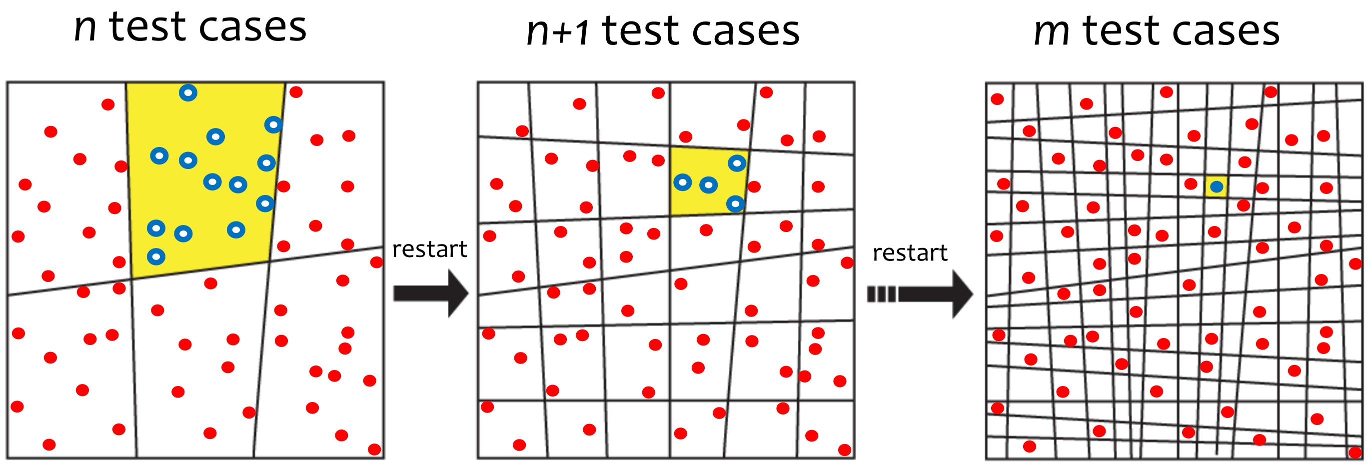
Use more test cases than necessary.

Existing techniques

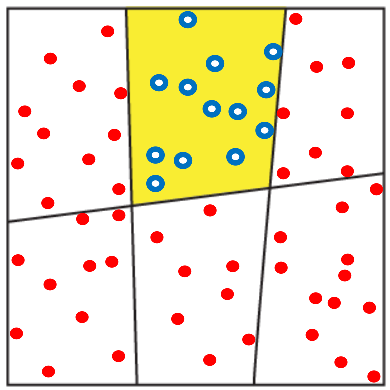
[Barthe *et al.* PPoPP'13]

[Udupa *et al.* PLDI'13]

[Bansal et al. ASPLOS'06]



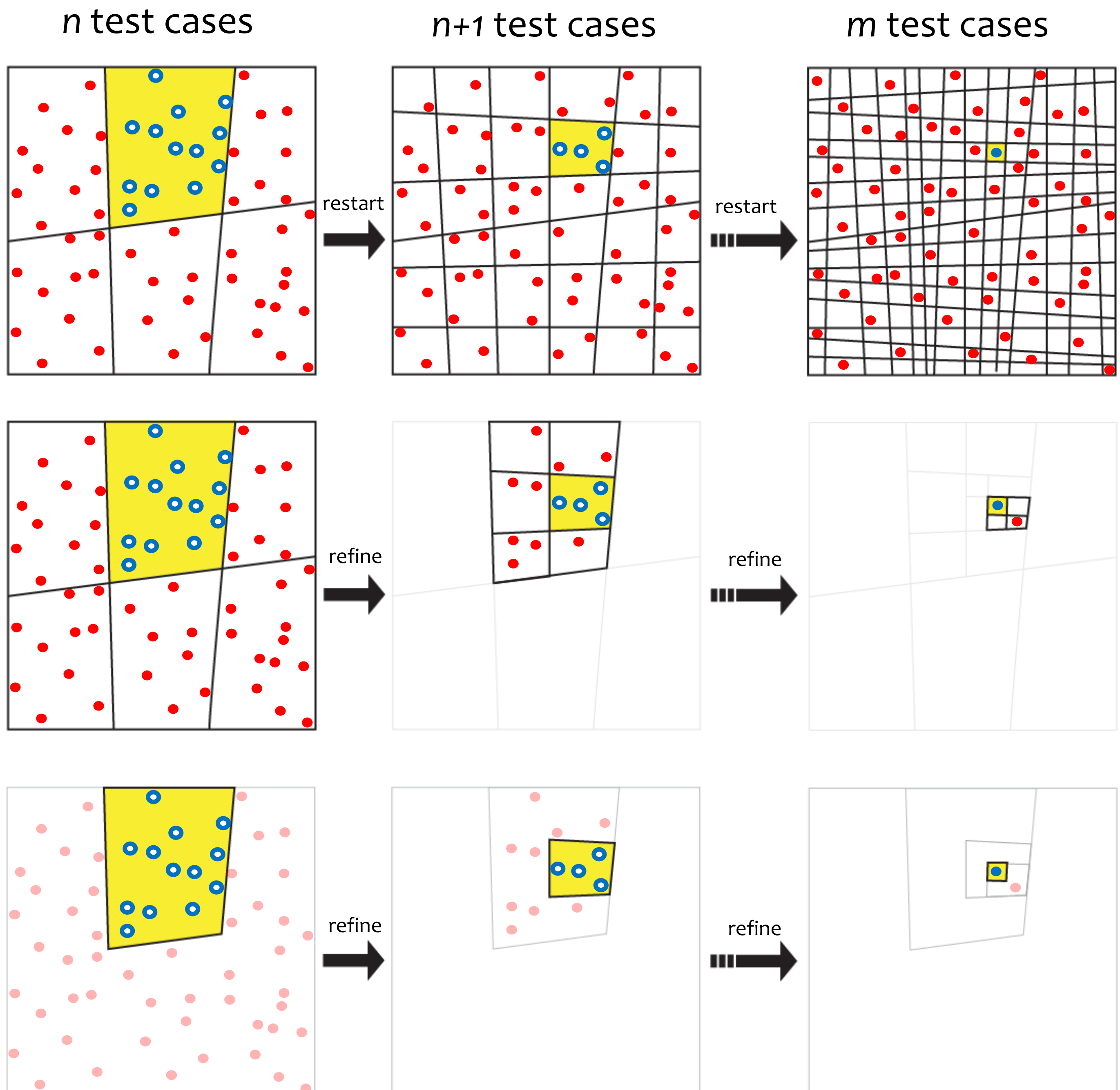
+ Selective refinement



Existing techniques
[Barthe *et al.* PPoPP'13]
[Udupa *et al.* PLDI'13]
[Bansal et al. ASPLOS'06]

+ Selective refinement

+ Bidirectional search
related work
[Bansal, Thesis'08]

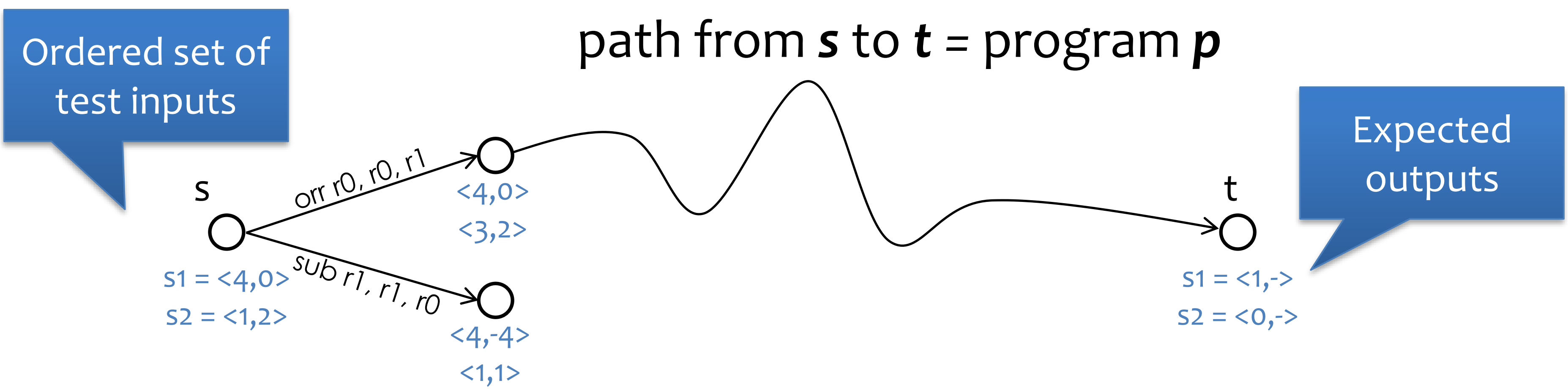


Problem Formulation

Superoptimization = graph search problem

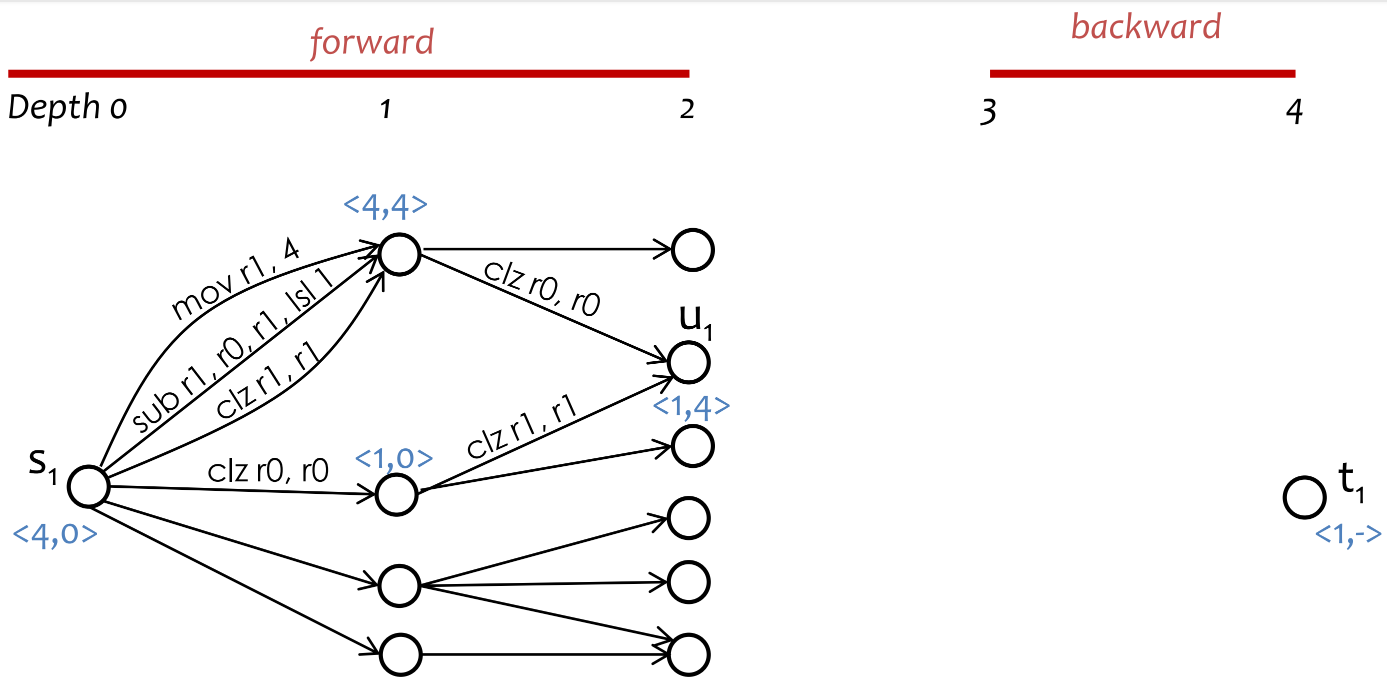
Problem: find program $p \equiv p_{spec}$ with respect to a set of test cases

Example: program state <r0, r1>

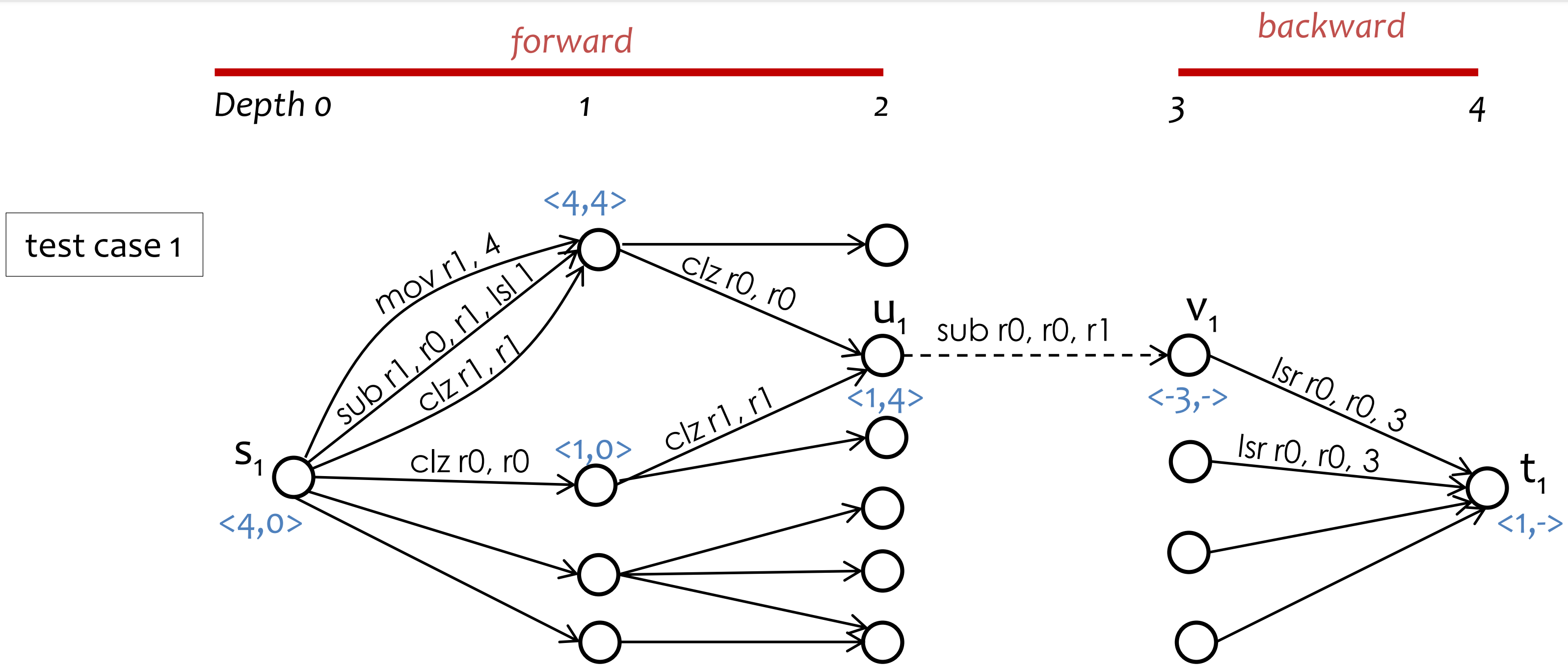


Lens Algorithm

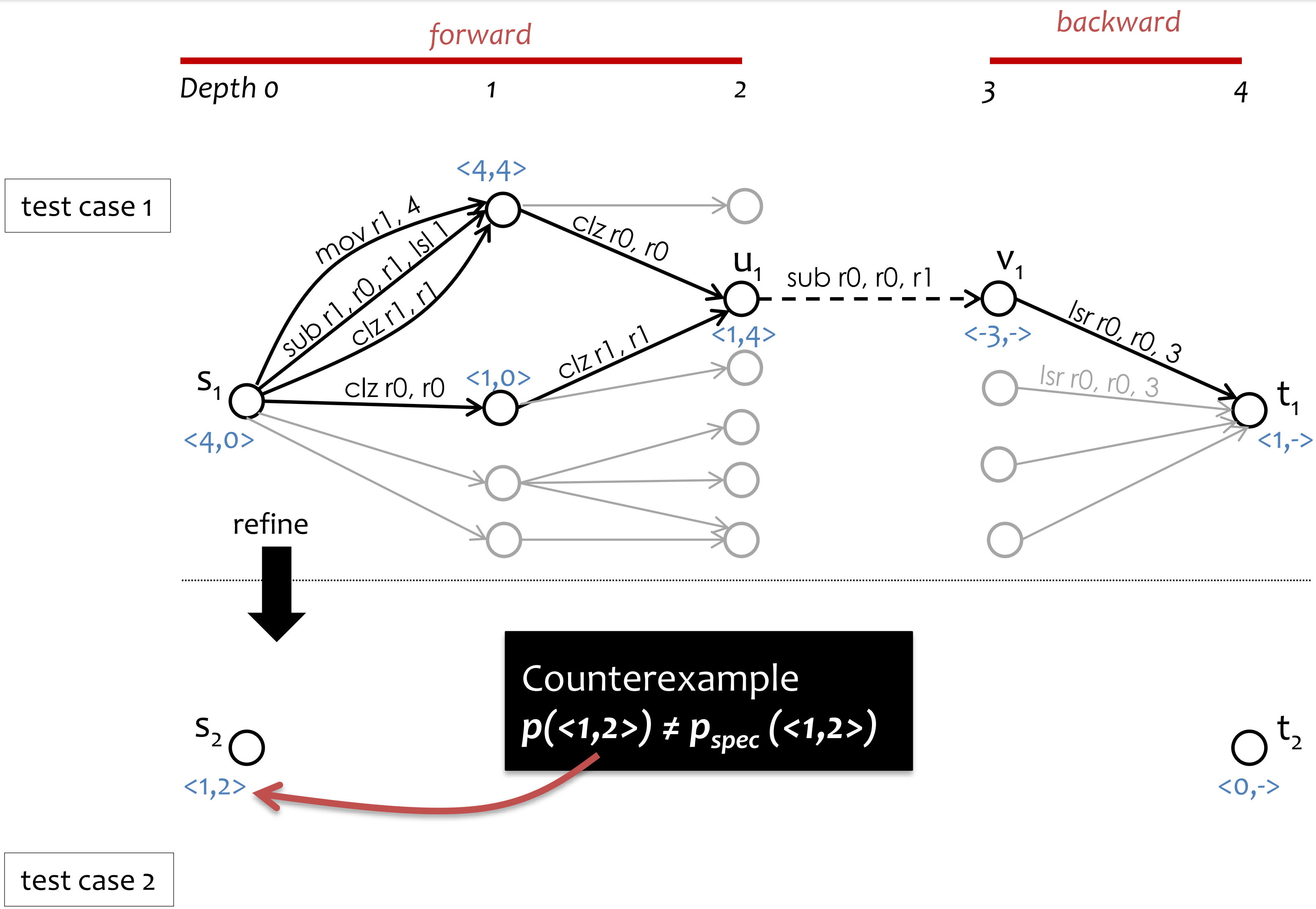
test case 1



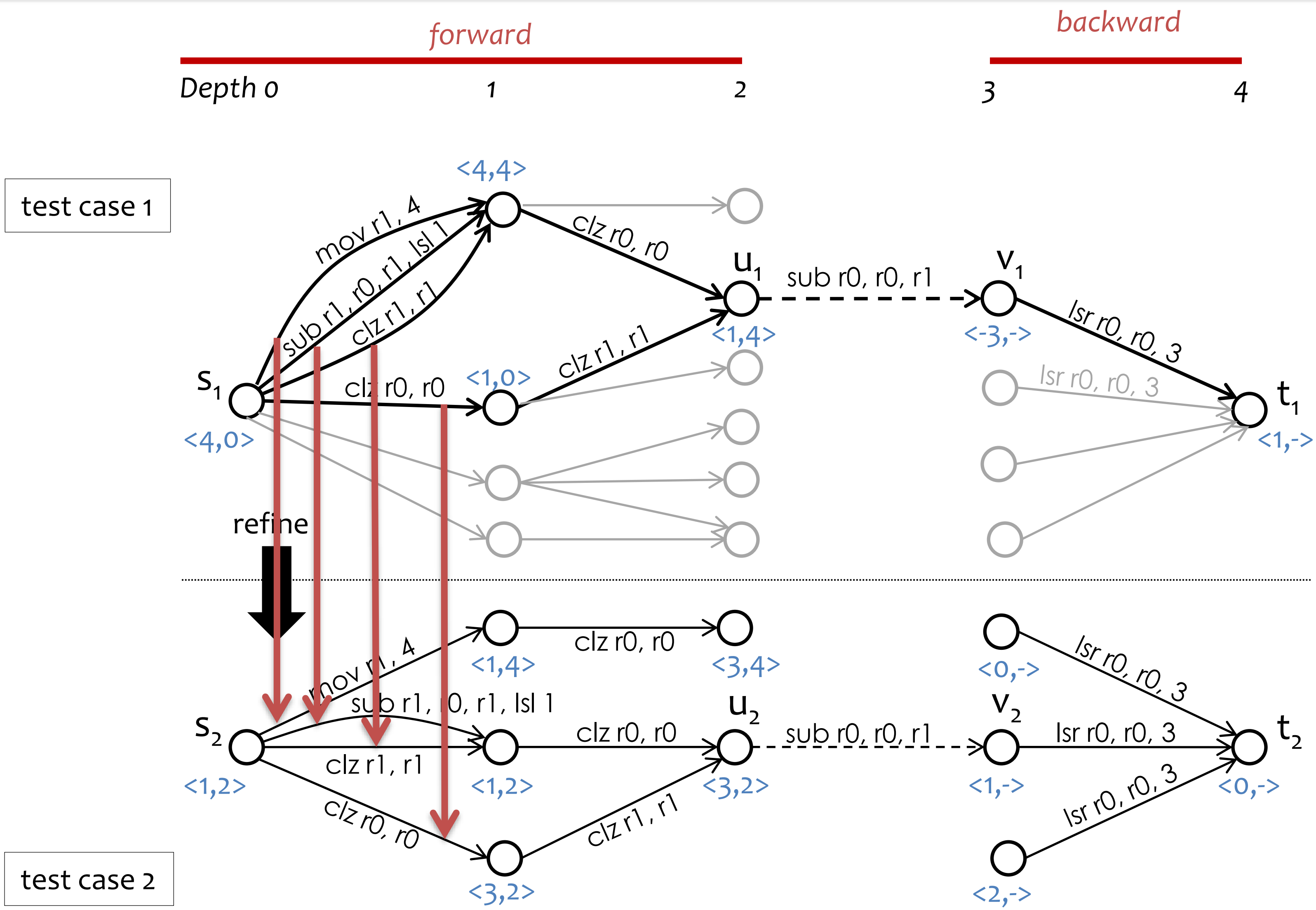
Lens Algorithm



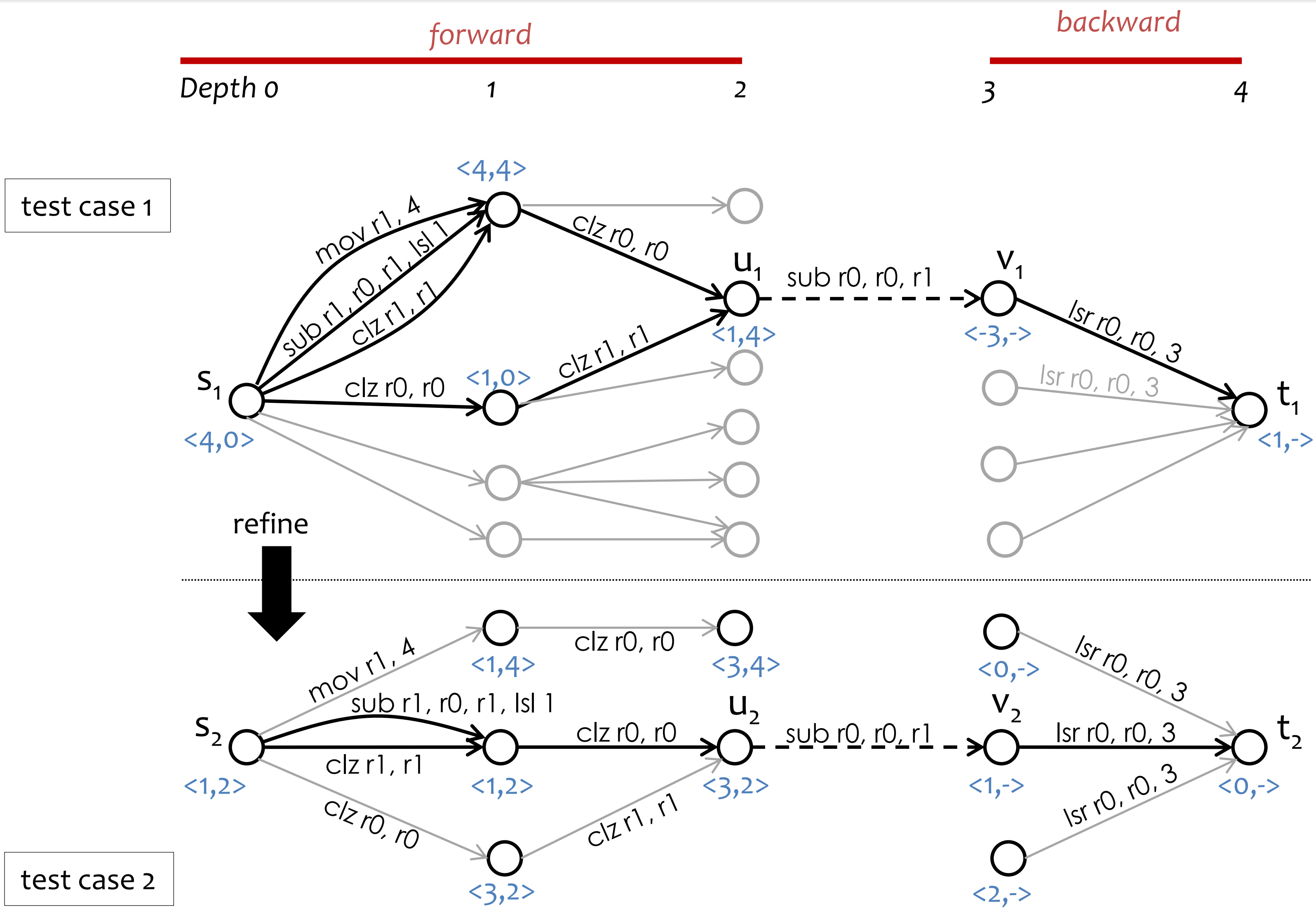
Lens Algorithm



Lens Algorithm



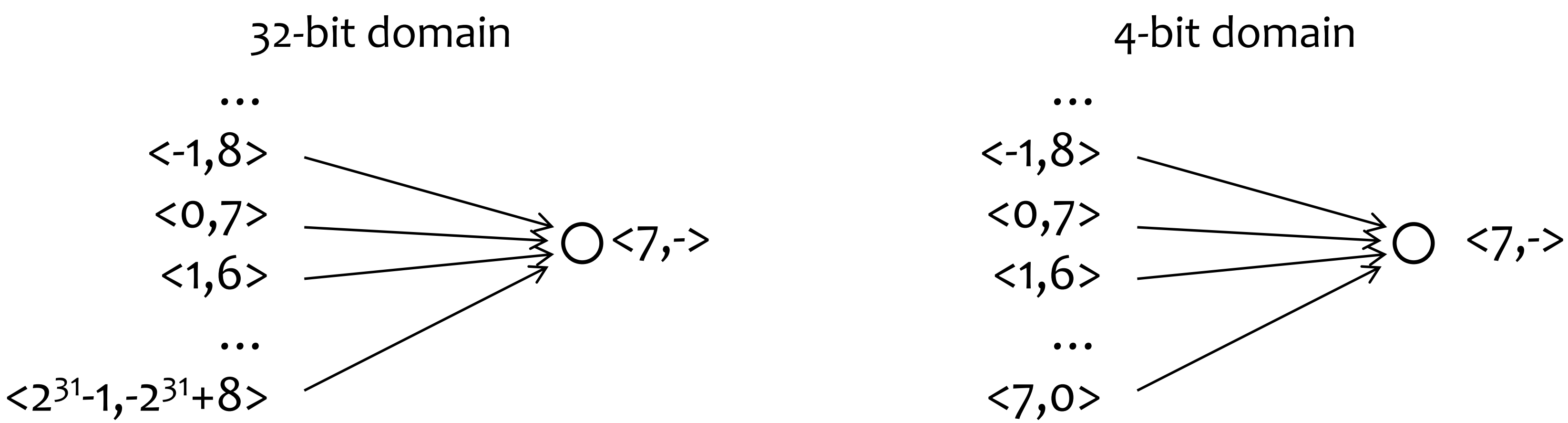
Lens Algorithm



Lens: Reduced Bitwidth

Challenge by backward traversal:

add r0, r0, r1
(r0 = r0 + r1)



2^{32} edges!

16 edges!

Solution:

- Search in reduced-bitwidth domain
- Verify in the original domain

Lens: Evaluation

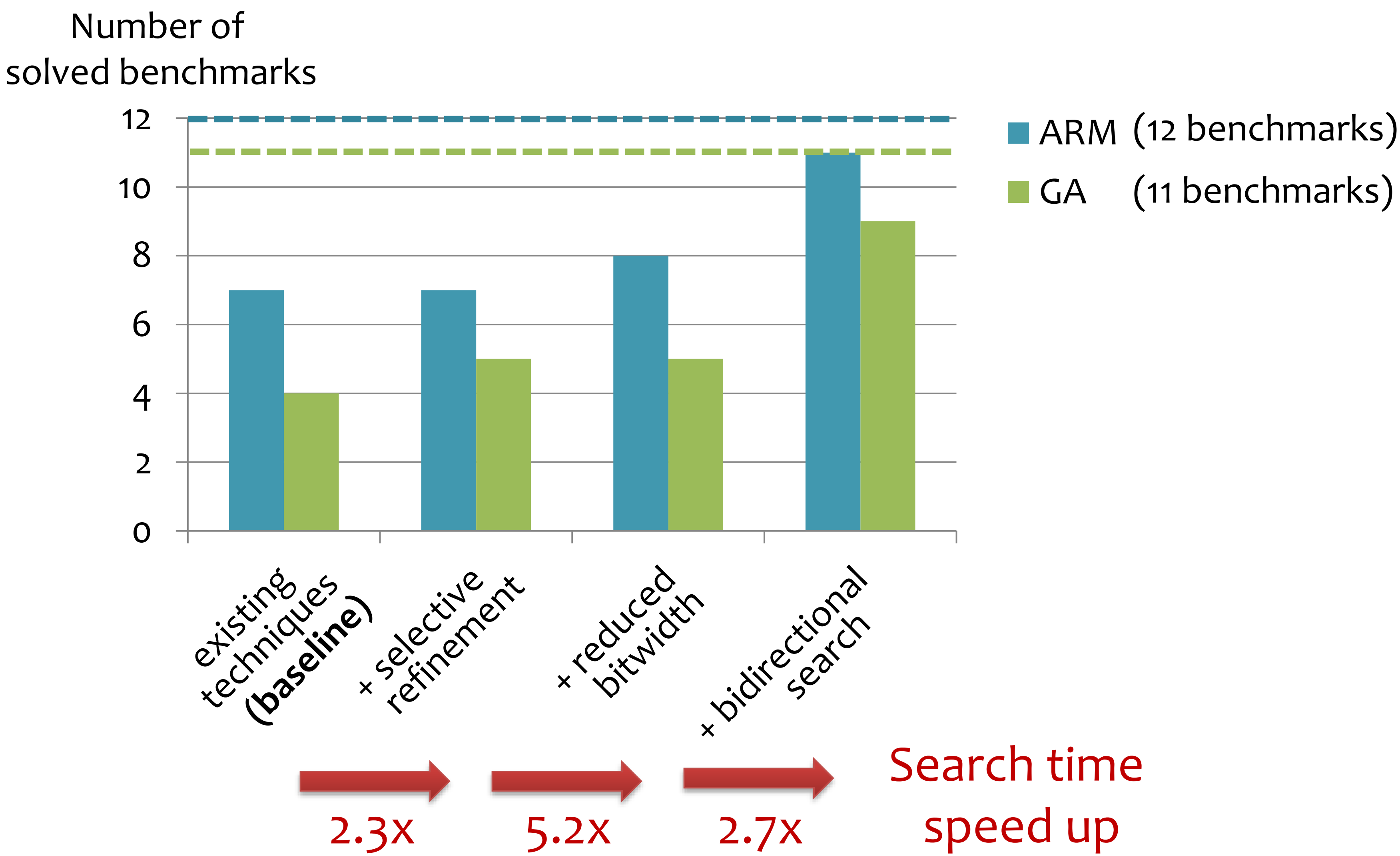
ARM

- Bit-twiddling benchmarks from *Hacker's Delight*
- Input = code generated from `gcc -O0`
- Timeout = 1 hour

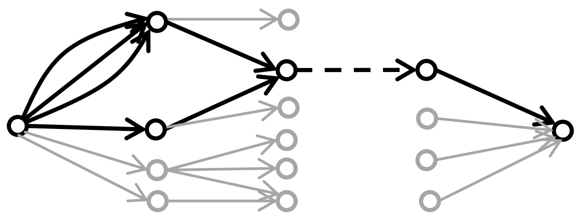
GreenArrays (GA) 18-bit stack-based architecture

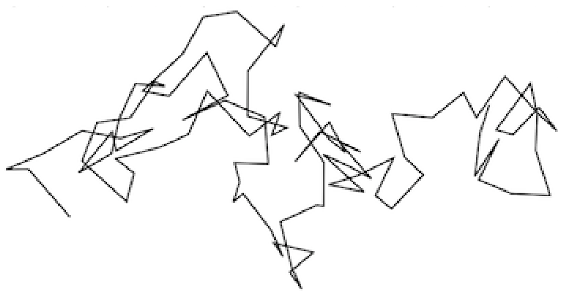
- Frequently-executed basic blocks from MD5, SHA-256, FIR, sine, and cosine functions
- Input = code generated from Chlorophyll compiler without optimizations [Phothilimthana *et al.* PLDI'14]
- Timeout = 20 min

Lens vs. Existing Techniques



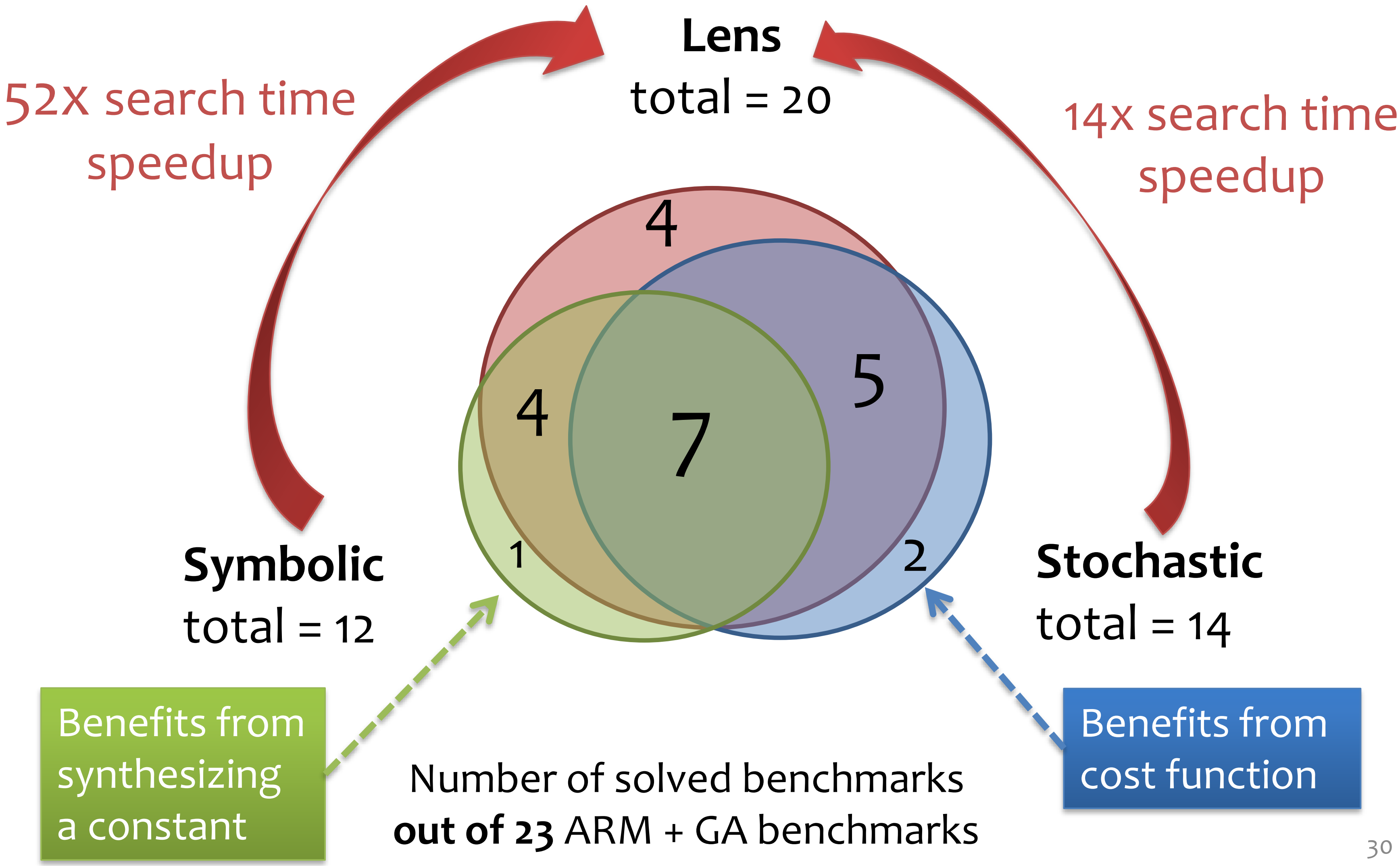
Other Search Techniques

Technique	Description	Pros	Cons
Enumerative		Can apply many pruning strategies specific to program synthesis problem.	Takes a long time to get to big programs. Require a lot of memory.

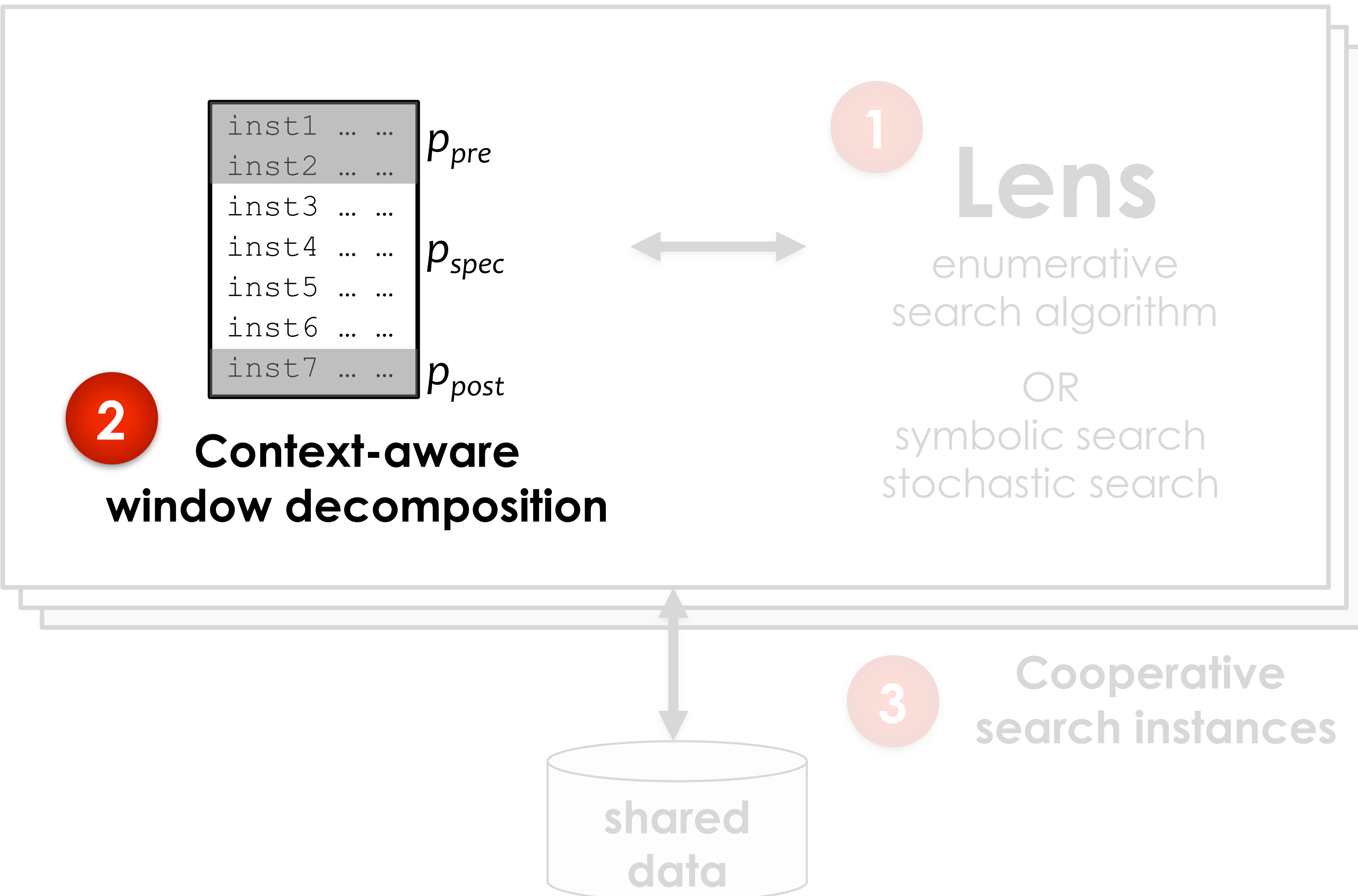
Stochastic	
	Example:
	`cmp r0, r1`
	`movls r0, #0`
	➡
	`cmp r0, r1`
	`movls r1, #0`
	[Schkufza *et al.* ASPLOS'13]
		Can jump to anywhere in the search space.
		
		Guided by cost (correctness + performance).
			Stuck at local minima, esp. at an incorrect program.

Symbolic	Program -> Logical formula Use a constraint solver to perform the search. [Solar-Lezama *et al.* ASPLOS'06]	Can synthesize arbitrary constants.	Slow.

Lens vs. Other Techniques

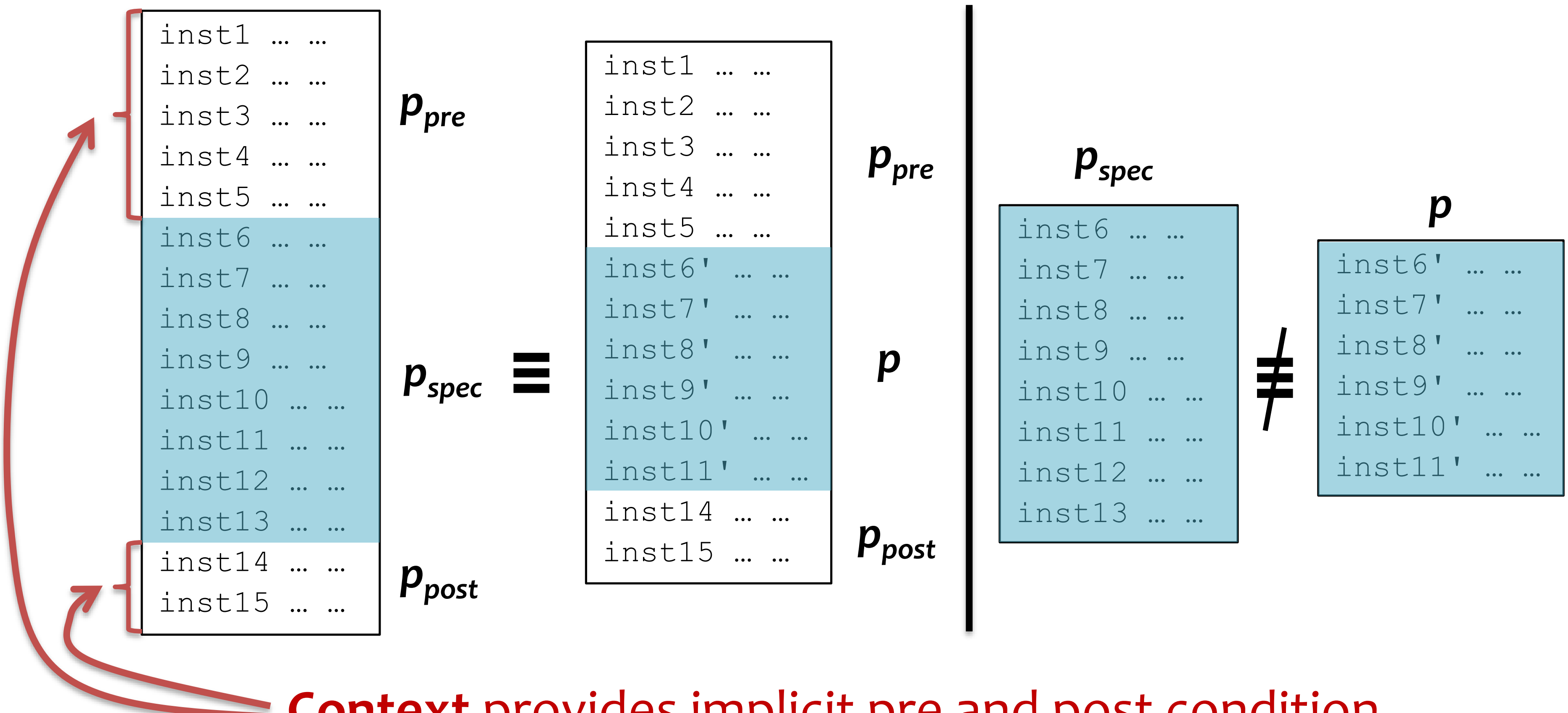


Context-Aware Window Decomposition



Context-Aware Window Decomposition

Find program ***p*** such that $p_{pre} + p + p_{post} \equiv p_{pre} + p_{spec} + p_{post}$



Context provides implicit pre and post condition.

Related work: Conditionally Correct Superoptimization [Sharma *et al.* OOPSLA'15]

Context-Aware Window Decomposition

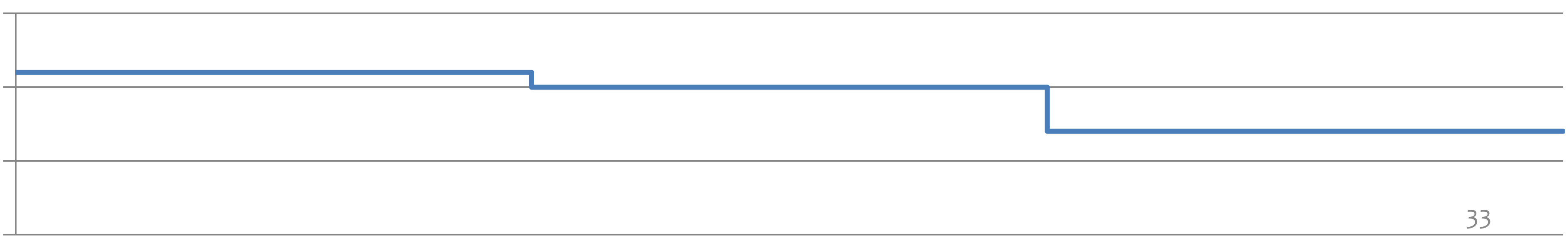
Optimize bitarray benchmark from MiBench (embedded system benchmark suite)

```
cmp    r1, #0
mov    r3, r1, asr #31
add    r2, r1, #7
mov    r3, r3, lsr #29
movge  r2, r1
ldrb   r0, [r0, r2, asr #3]
add    r1, r1, r3
and    r1, r1, #7
sub    r3, r1, r3
asr    r1, r0, r3
and    r0, r0, #1
```

```
cmp    r1, #0
mov    r3, r1, asr #31
add    r2, r1, #7
mov    r3, r3, lsr #29
movge  r2, r1
ldrb   r0, [r0, r2, asr #3]
bic    r1, r2, #248
sub    r3, r1, r3
asr    r1, r0, r3
and    r0, r1, #1
```

```
asr    r3, r1, #2
add    r2, r1, r3, lsr #29
ldrb   r0, [r0, r2, asr #3]
and    r3, r2, #248
sub    r3, r1, r3
asr    r1, r0, r3
and    r0, r1, #1
```

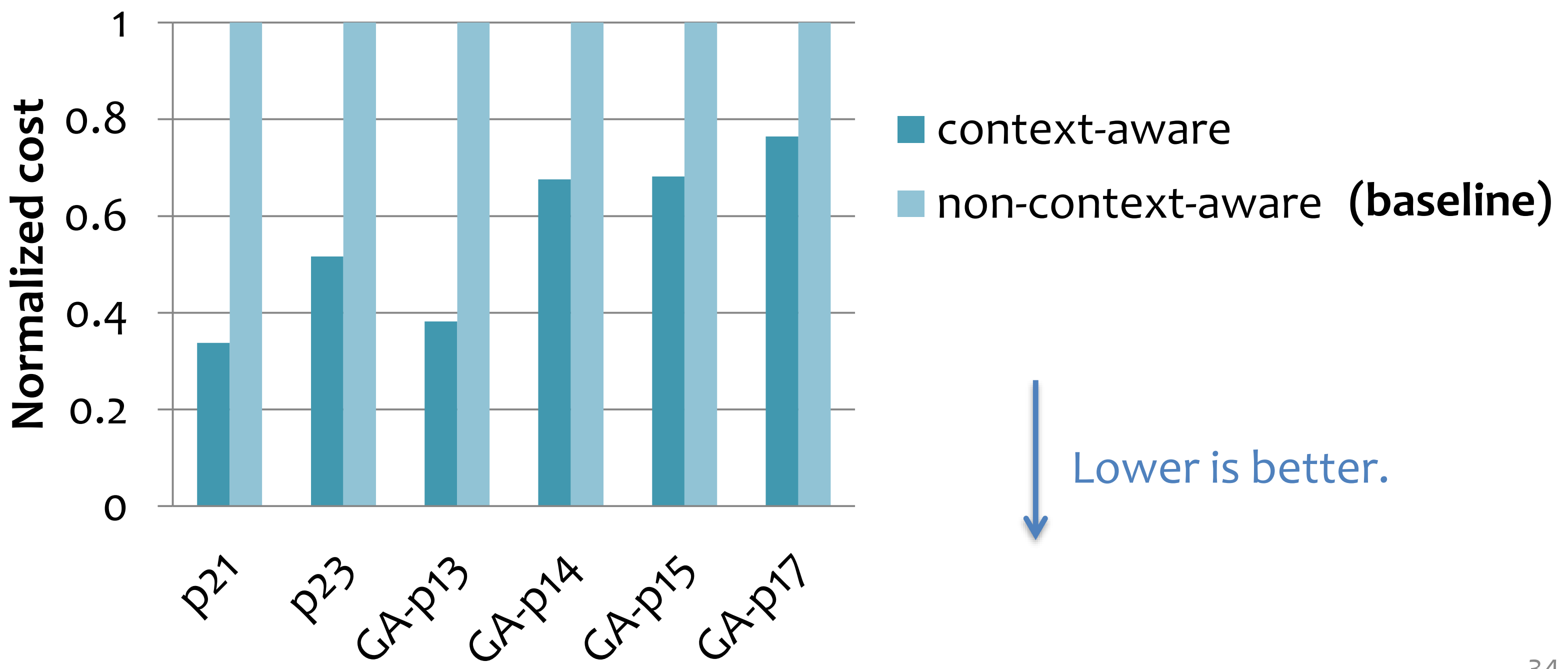
Performance cost



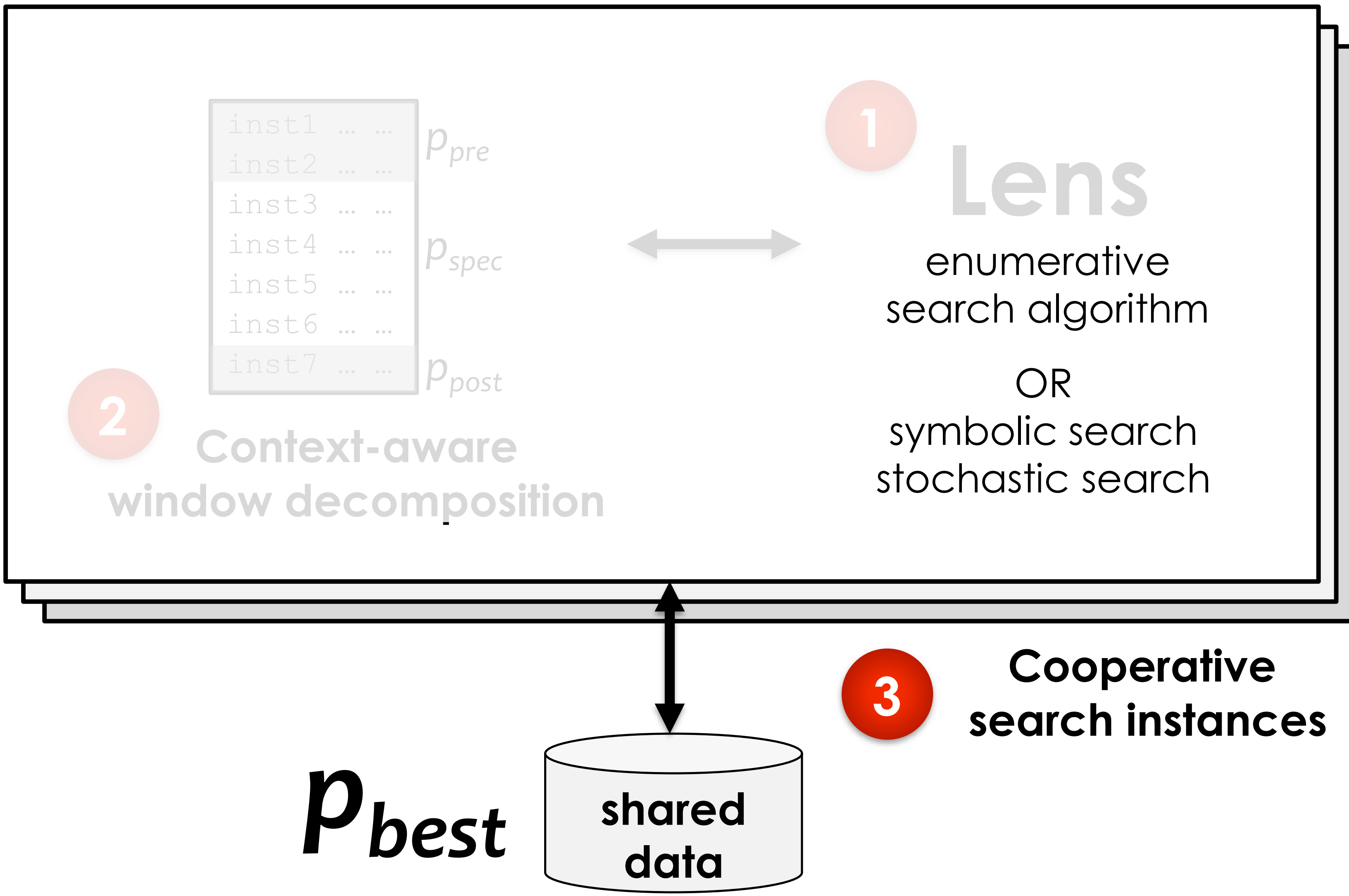
Context-Aware: Evaluation

Context-aware vs. Non-context-aware decomposition

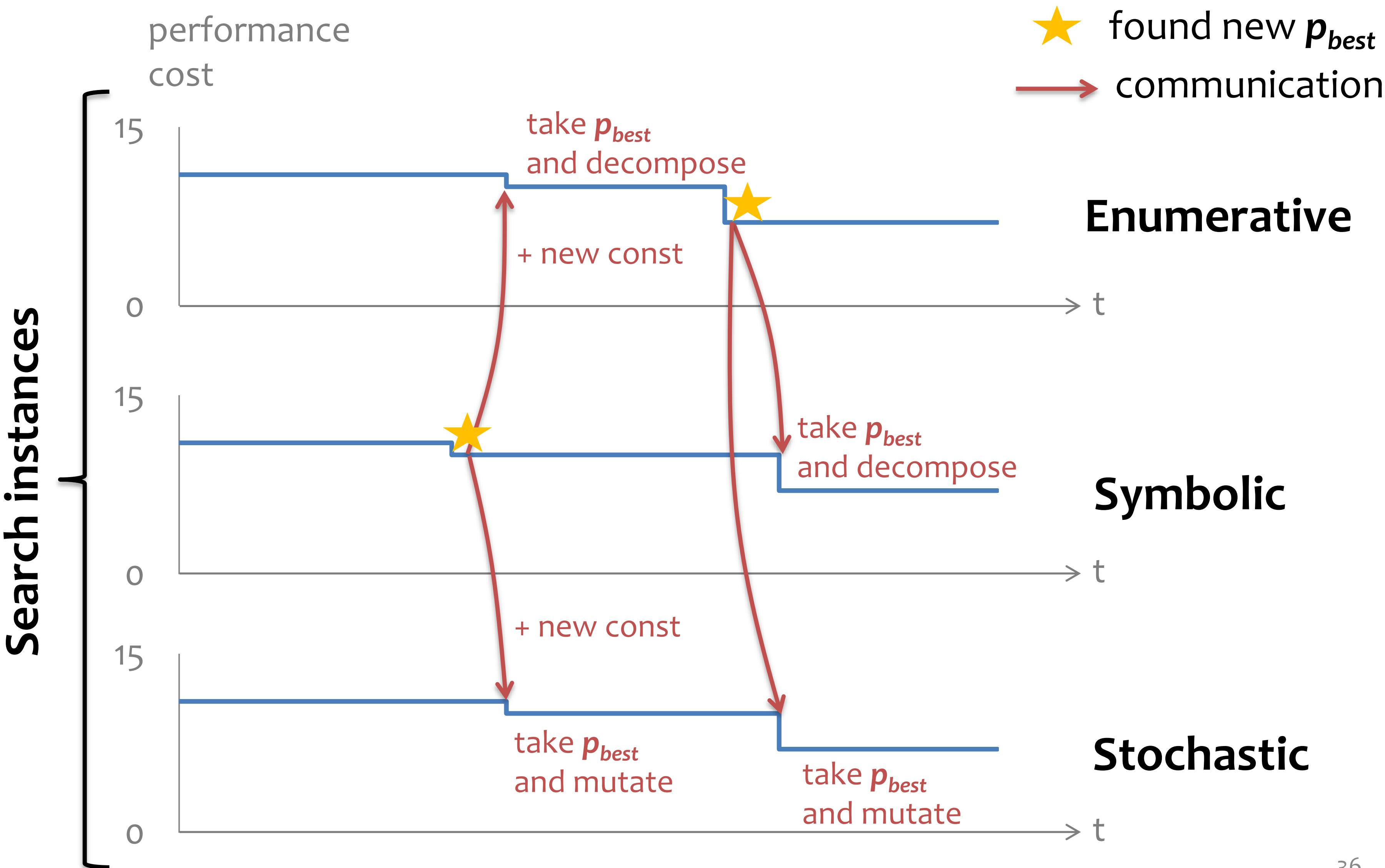
On 6 out of 12 benchmarks, context-aware decomposition improves code significantly.



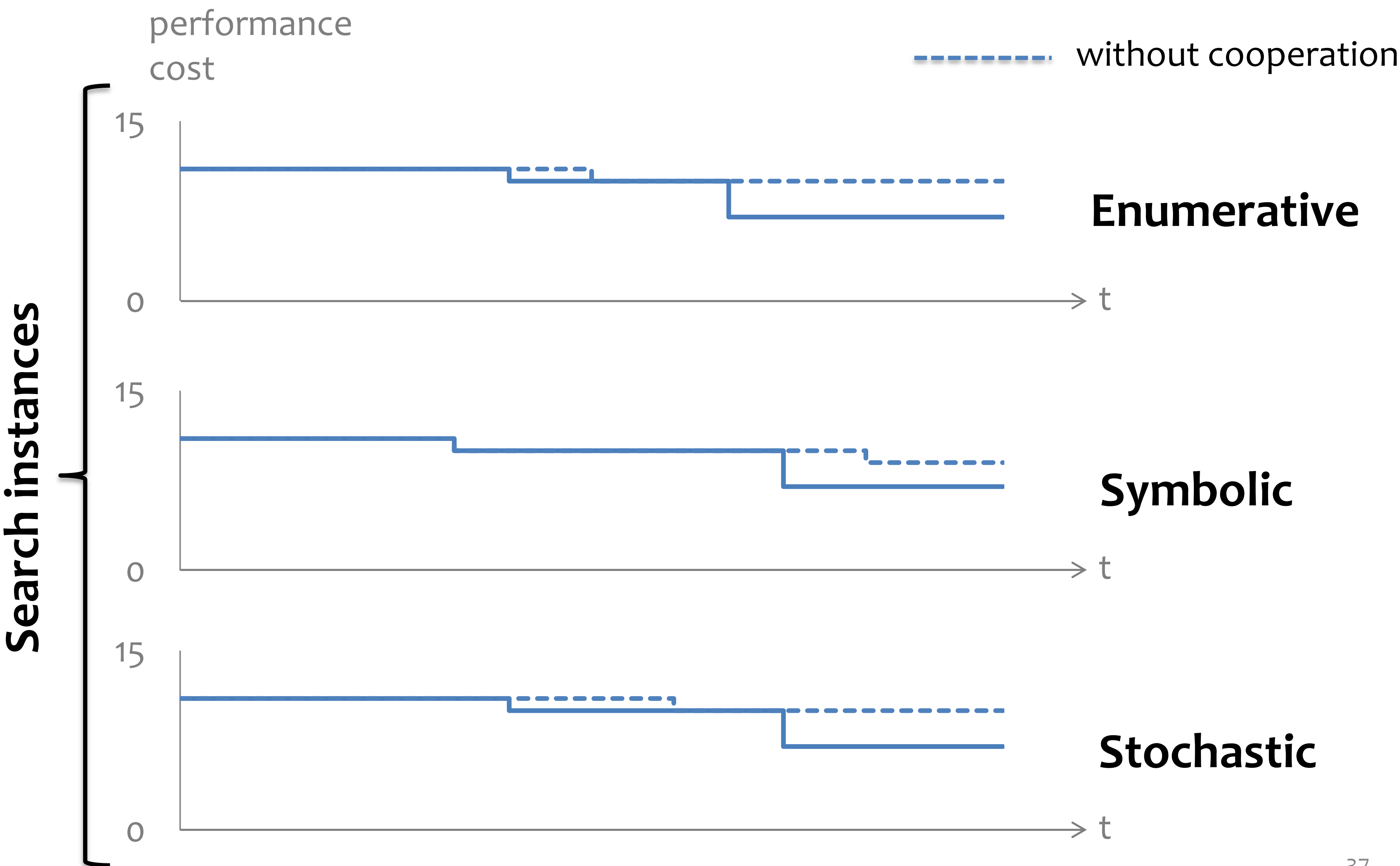
Cooperative Superoptimizer



Cooperative Superoptimizer



Cooperative Superoptimizer



Cooperative Superoptimizer

Optimize bitarray benchmark from
MiBench (embedded system benchmark suite)

```
cmp   r1, #0
mov   r3, r1, asr #31
add   r2, r1, #7
mov   r3, r3, lsr #29
movge r2, r1
ldrb  r0, [r0, r2, asr #3]
add   r1, r1, r3
and   r1, r1, #7
sub   r3, r1, r3
asr   r1, r0, r3
and   r0, r0, #1
```

```
cmp   r1, #0
mov   r3, r1, asr #31
add   r2, r1, #7
mov   r3, r3, lsr #29
movge r2, r1
ldrb  r0, [r0, r2, asr #3]
bic   r1, r2, #248
sub   r3, r1, r3
asr   r1, r0, r3
and   r0, r1, #1
```

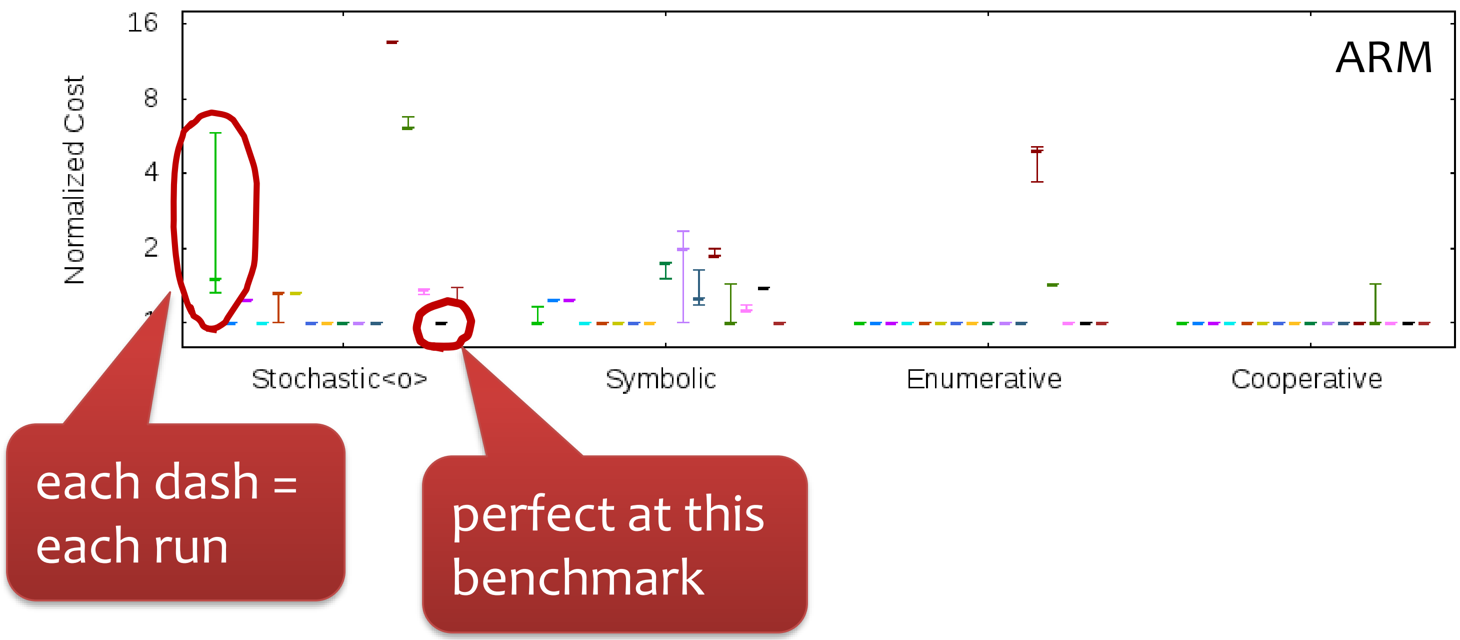
```
asr   r3, r1, #2
add   r2, r1, r3, lsr #29
ldrb  r0, [r0, r2, asr #3]
and   r3, r2, #248
sub   r3, r1, r3
asr   r1, r0, r3
and   r0, r1, #1
```

Symbolic
@ 5 mins

Enumerative
@ 10 mins

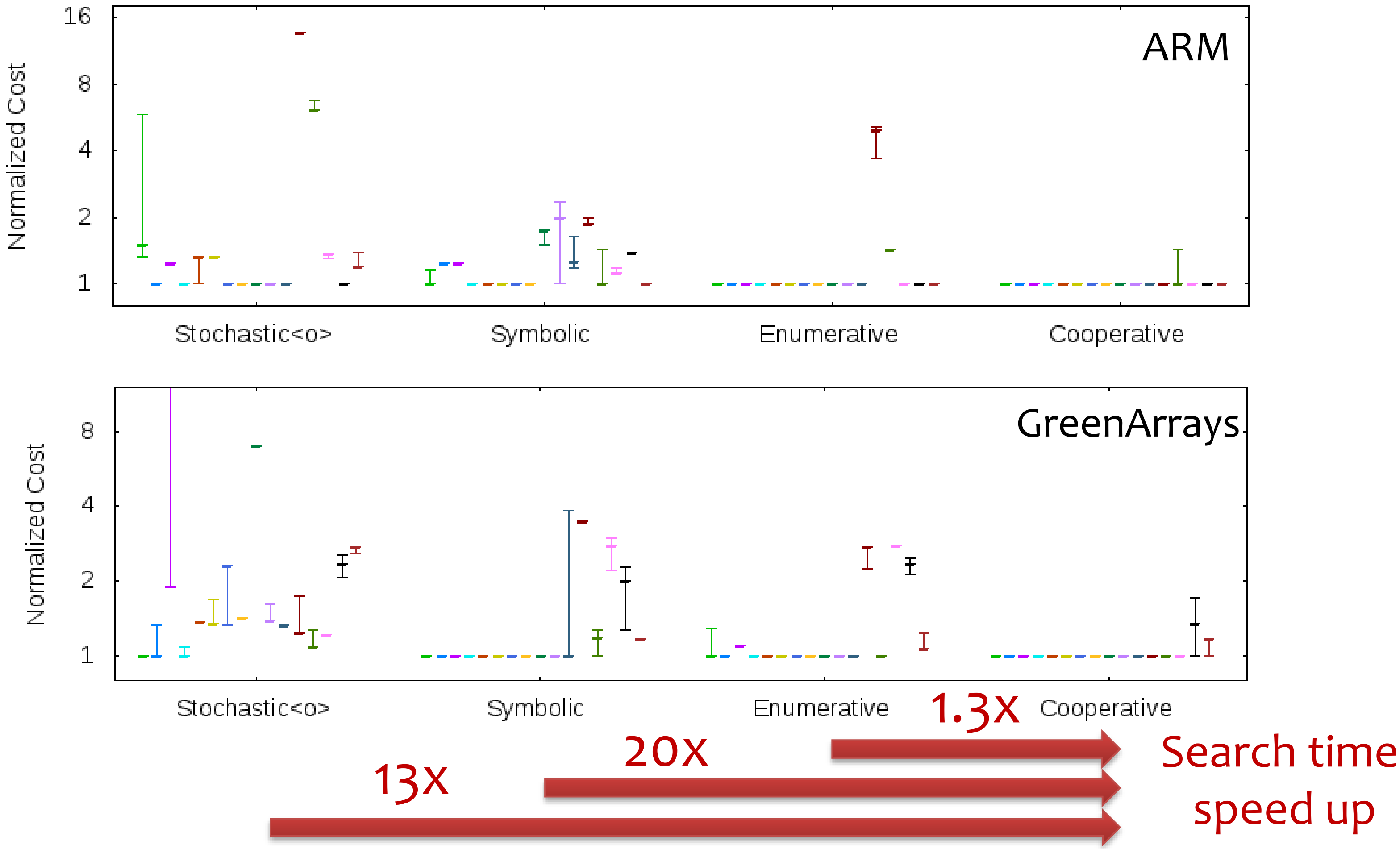
Cooperative: Evaluation

- Run each benchmark 3 times
- Normalize performance costs by cost of best known program
  Lower is better. Everything = 1 is the best.



Cooperative: Evaluation

- Run each benchmark 3 times
- Normalize performance costs by cost of best known program
  Lower is better.   Everything = 1 is the best.



Runtime Speedup

Runtime speedup over **gcc –O3** on an actual **ARM Cortex-A9**

Benchmarks Hacker's Delight, WiBench (wireless system kernel benchmarks), MiBench (embedded system kernel benchmarks)

Program	Search time (s)	gcc –O3 length	Output length	Runtime speedup on ARM Cortex-A9
p18	9	7	4	**2.11**
p21	1139	6	5	**1.81**
p23	665	18	16	**1.48**
p24	151	7	4	**2.75**
p25	2	11	1	**17.80**
WB-txrate5a	32	9	8	**1.31**
WB-txrate5b	66	8	7	**1.29**
MB-bitarray	612	10	6	**1.82**
MB-bitshift	5	9	8	**1.11**
MB-bitcnt	645	27	19	**1.33**
MB-susan-391	32	30	21	**1.26**

Runtime Speedup

Runtime speedup over **unoptimized code** generated by Chlorophyll compiler on actual **GreenArrays** hardware

Superoptimize every basic block in MD5 hash

- Superoptimization adds **49% speedup.**
- Only **19% slower than expert-written code.**
- In 3 functions, found code **1.3x – 2.5x faster** than expert's.

GreenThumb Framework

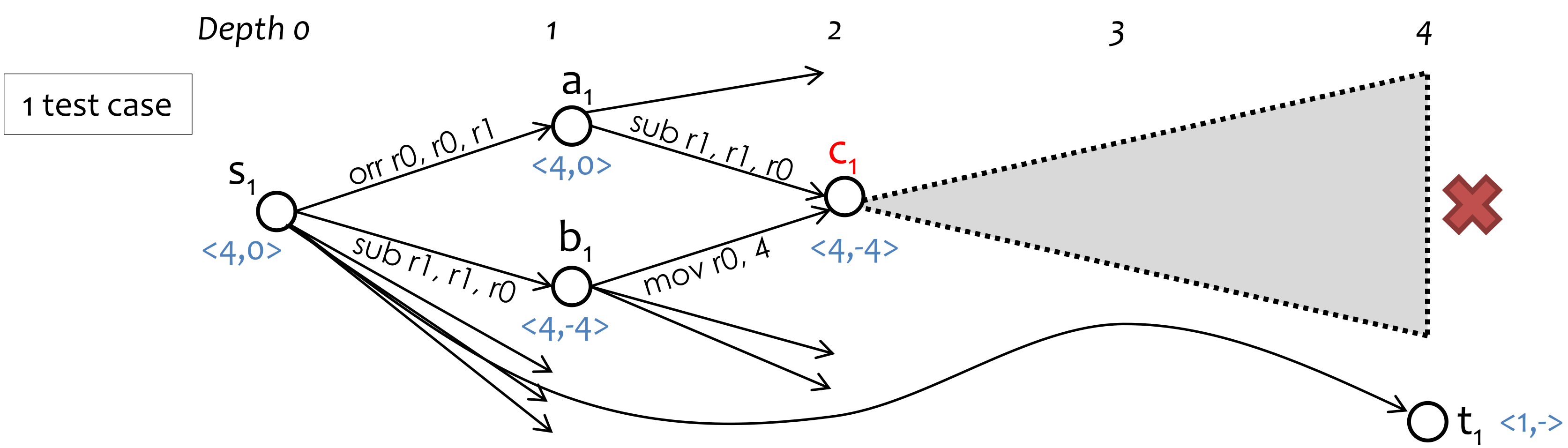
Provide cooperative search strategy.

Enable rapid retargeting of the superoptimizer to a new ISA.

github.com/mangpo/greenthumb

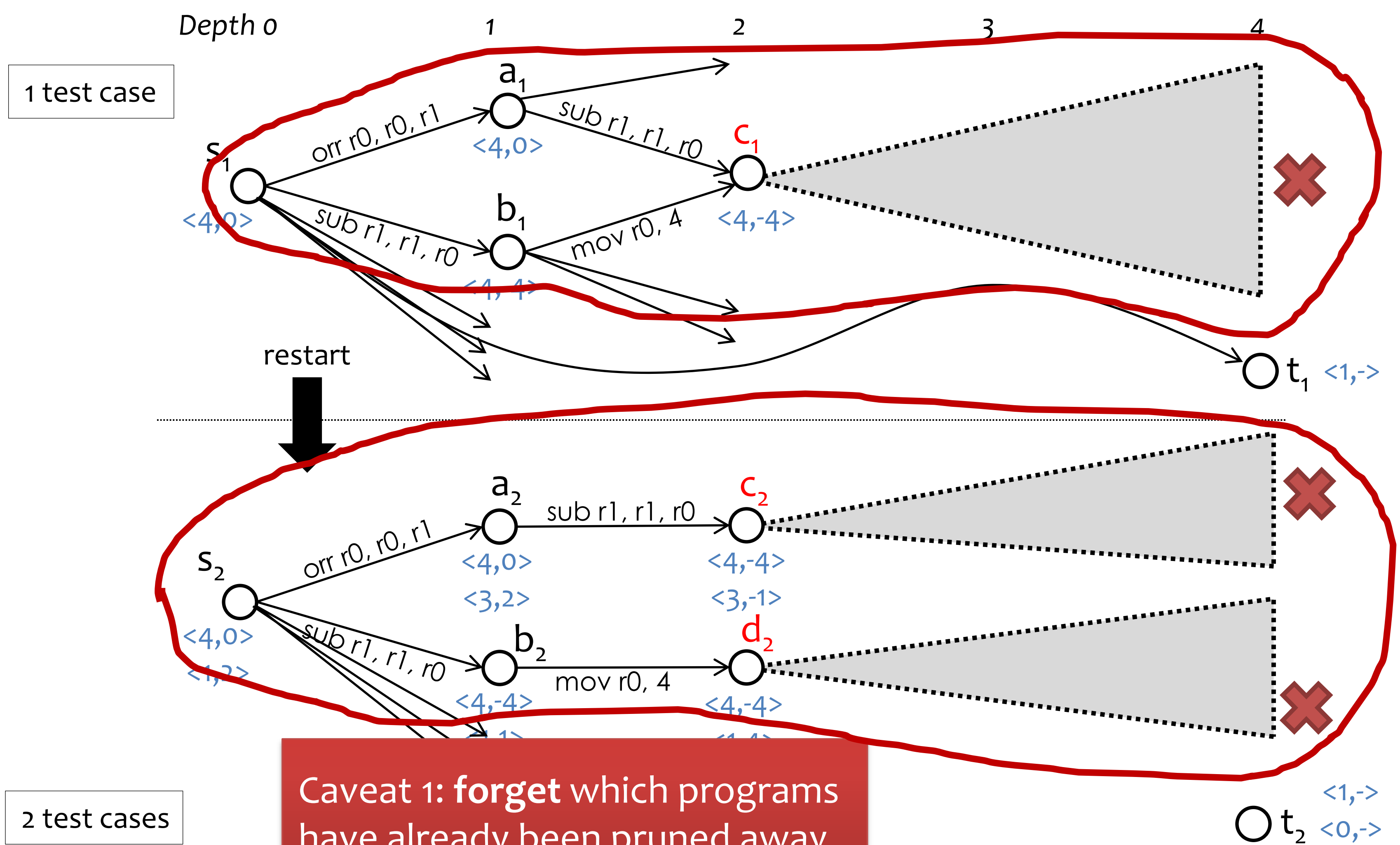
Backup

Existing Enumerative Search



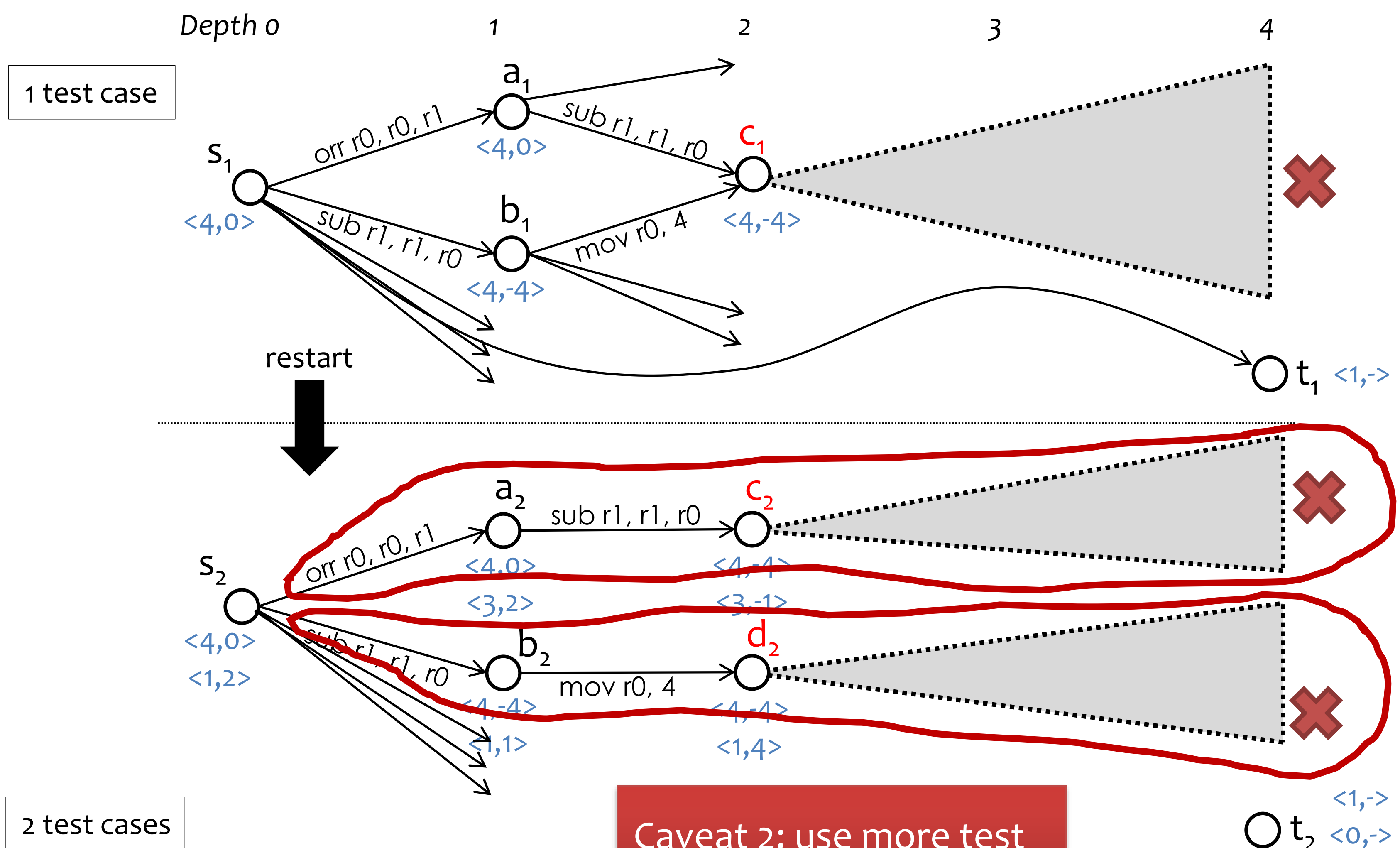
Only need to visit the subtree once!

Existing Enumerative Search



Caveat 1: **forget** which programs have already been pruned away in the previous rounds.

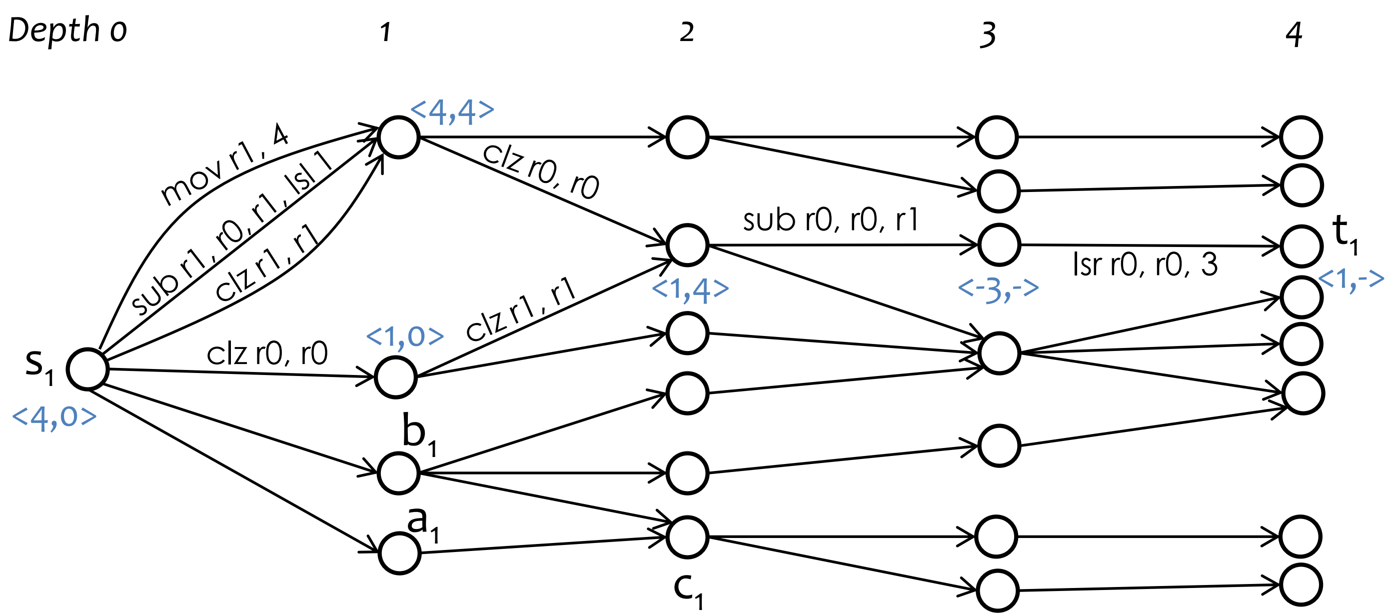
Existing Enumerative Search



Caveat 2: use more test cases than necessary.

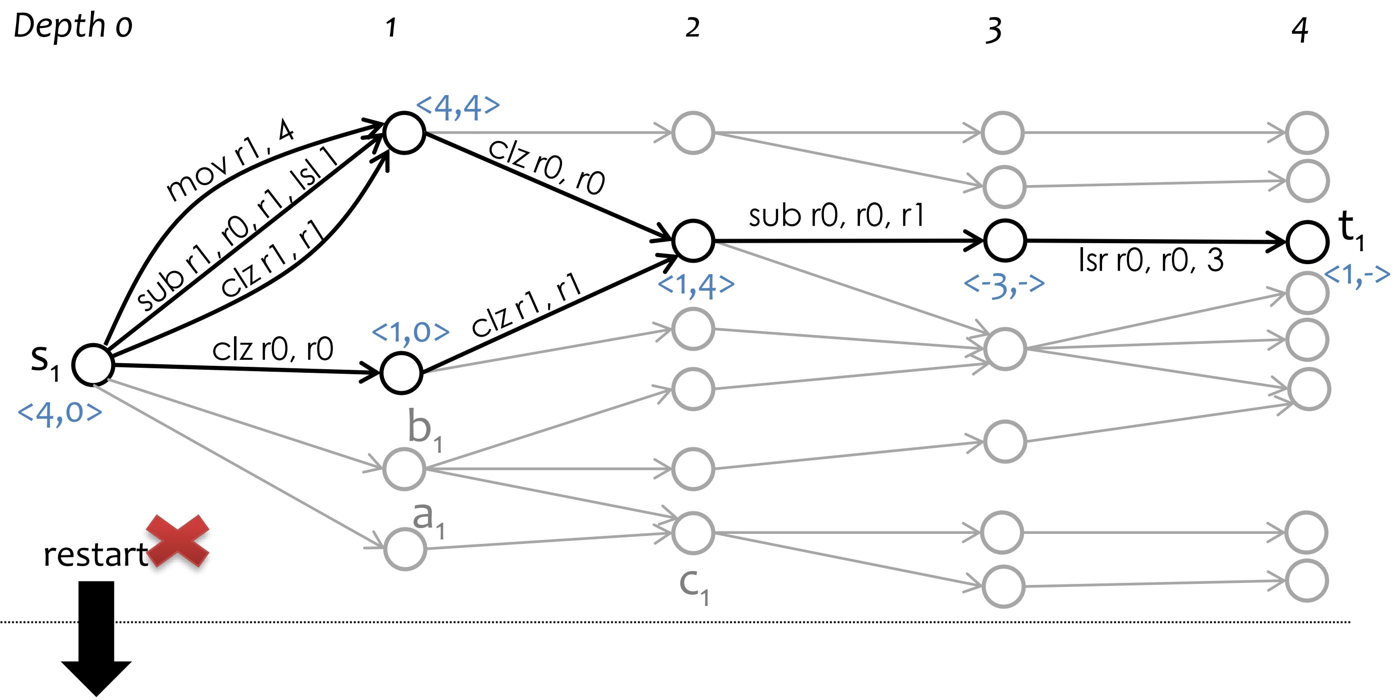
LENS: Selective Refinement

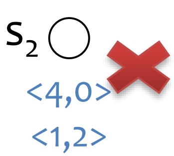
1 test case



LENS: Selective Refinement

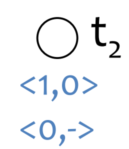
1 test case





Counterexample
$p(<1,2>) \neq p_{spec}(<1,2>)$

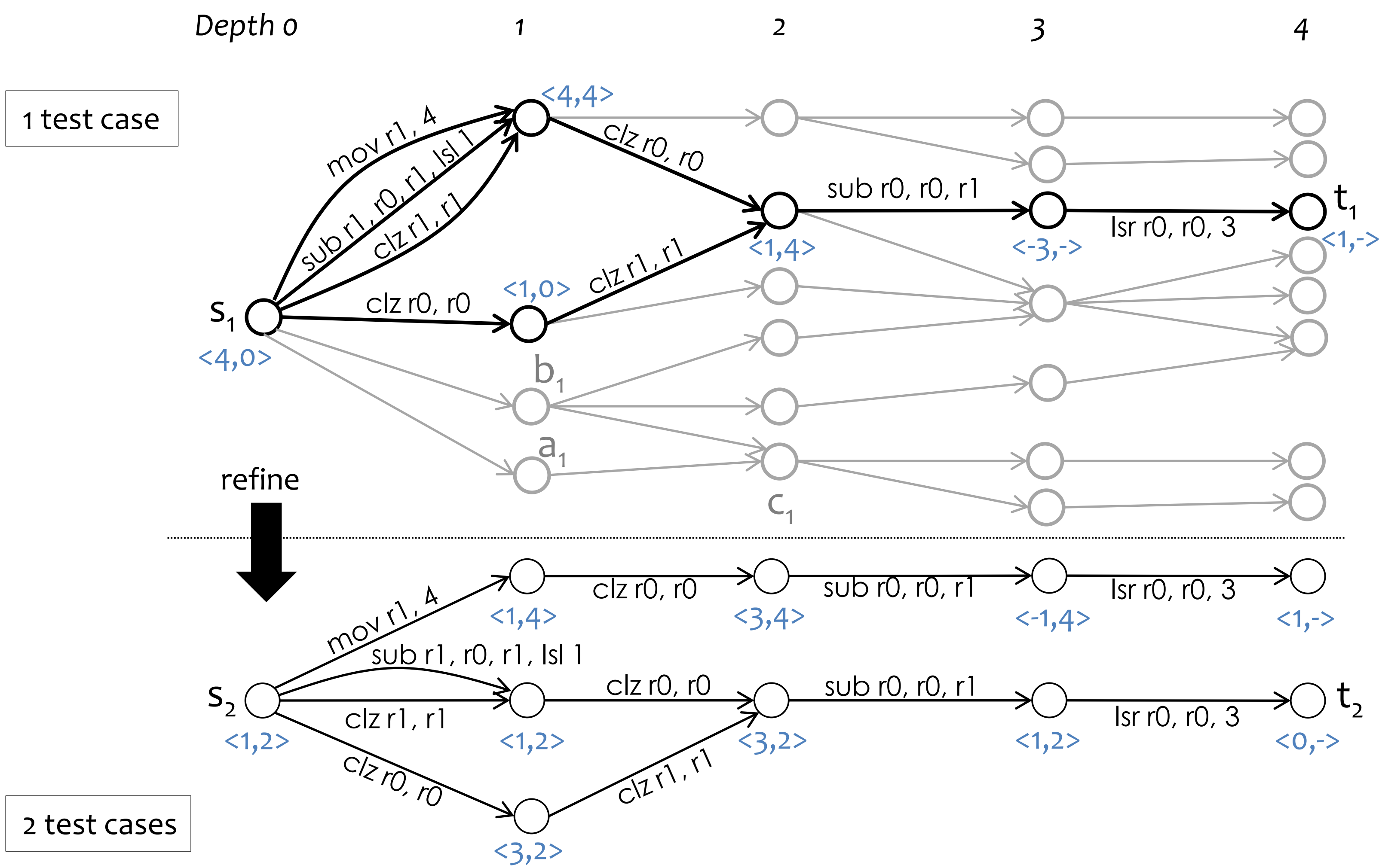
Existing techniques: restart the search



2 test cases

We refine the search instead.

LENS: Selective Refinement



Cooperative Superoptimizer

Run multiple search instances employing different search techniques.

- Enumerative & Symbolic
  optimize $\boldsymbol{p_{best}}$ by applying window decomposition.
- Stochastic
  optimizes $\boldsymbol{p_{best}}$ by mutating it.
- Enumerative & Stochastic
  add new constants in $\boldsymbol{p_{best}}$ to their list.

Concrete Example

Basic block from bitarray benchmark from MiBench (embedded system benchmark suite)

Optimization I: eliminate a conditional branch.

```
cmp    r1, #0
mov    r3, r1, asr #31
add    r2, r1, #7
mov    r3, r3, lsr #29
movge  r2, r1
ldrb   r0, [r0, r2, asr #3]
bic    r1, r2, #248
sub    r3, r1, r3
asr    r1, r0, r3
and    r0, r1, #1
```

E →

```
asr    r3, r1, #2
add    r2, r1, r3, lsr #29
ldrb   r0, [r0, r2, asr #3]
and    r3, r2, #248
sub    r3, r1, r3
asr    r1, r0, r3
and    r0, r1, #1
```

Concrete Example

Basic block from bitarray benchmark from MiBench (embedded system benchmark suite)

Optimization II: context-specific

```
cmp    r1, #0
mov    r3, r1, asr #31
add    r2, r1, #7
mov    r3, r3, lsr #29
movge  r2, r1
ldrb   r0, [r0, r2, asr #3]
bic    r1, r2, #248
sub    r3, r1, r3
asr    r1, r0, r3
and    r0, r1, #1
```

E

```
asr    r3, r1, #2
add    r2, r1, r3, lsr #29
ldrb   r0, [r0, r2, asr #3]
and    r3, r2, #248
sub    r3, r1, r3
asr    r1, r0, r3
and    r0, r1, #1
```

Runtime Speedup

Program	gcc -O3 length	Output length	Search time (s)	Speed -up	Path to best code
p18	7	4	9	2.11	E^s
p21	6	5	1139	1.81	E^o*, SM^o*, ST^o*
p23	18	16	665	1.48	$ST^o* \rightarrow E^o*$
p24	7	4	151	2.75	$ST^o* \rightarrow E^o* \rightarrow ST^o \rightarrow E^o*$
p25	11	1	2	17.8	E^s
wi-txrate5a	9	8	32	1.31	$SM^o \rightarrow ST^o$
wi-txrate5b	8	7	66	1.29	E^o
mi-bitarray	10	6	612	1.82	$SM^o* \rightarrow E^o*$
mi-bitshift	9	8	5	1.11	E^o
mi-bitcnt	27	19	645	1.33	$E^o \rightarrow ST^o \rightarrow E^o \rightarrow ST^o \rightarrow E^o$
mi-susan	30	21	32	1.26	ST^o