Optimizing Homomorphic Evaluation Circuit with Program Synthesis and Term Rewriting

Woosuk Lee

Hanyang University

Joint work with DongKwon Lee (SNU), Hakjoo Oh (Korea Univ.), and Kwangkeun Yi (SNU)

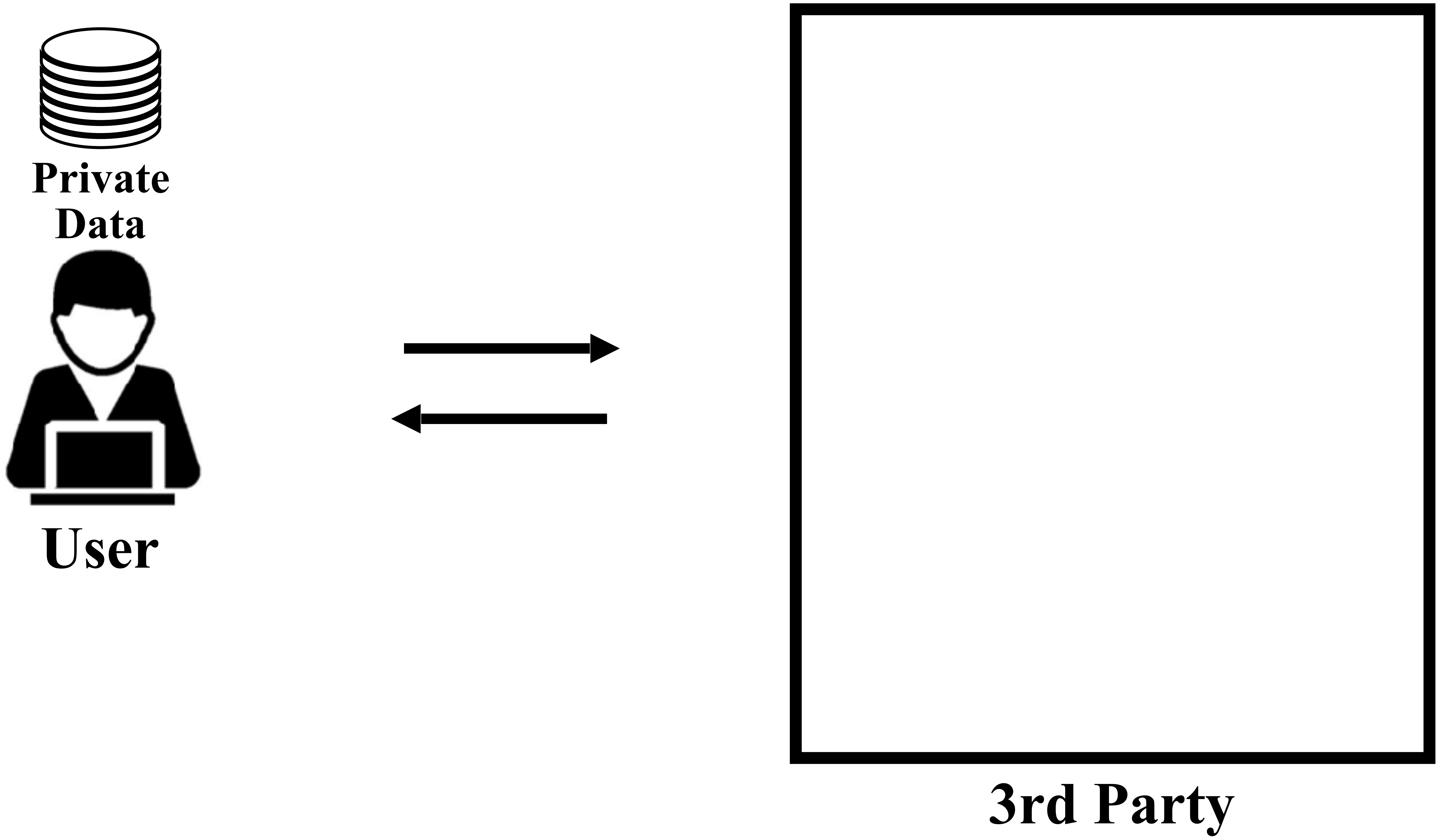
References

- *Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Term Rewriting*, **PLDI 2020**: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation
- *Optimizing Homomorphic Evaluation Circuits by Program Synthesis, Term Rewriting, and Time-bounded Exhaustive Search*, **TOPLAS**: ACM Transactions on Programming Languages and Systems (under review)

Homomorphic Evaluation(HE) (1/3)

Privacy Preserving Secure Computation

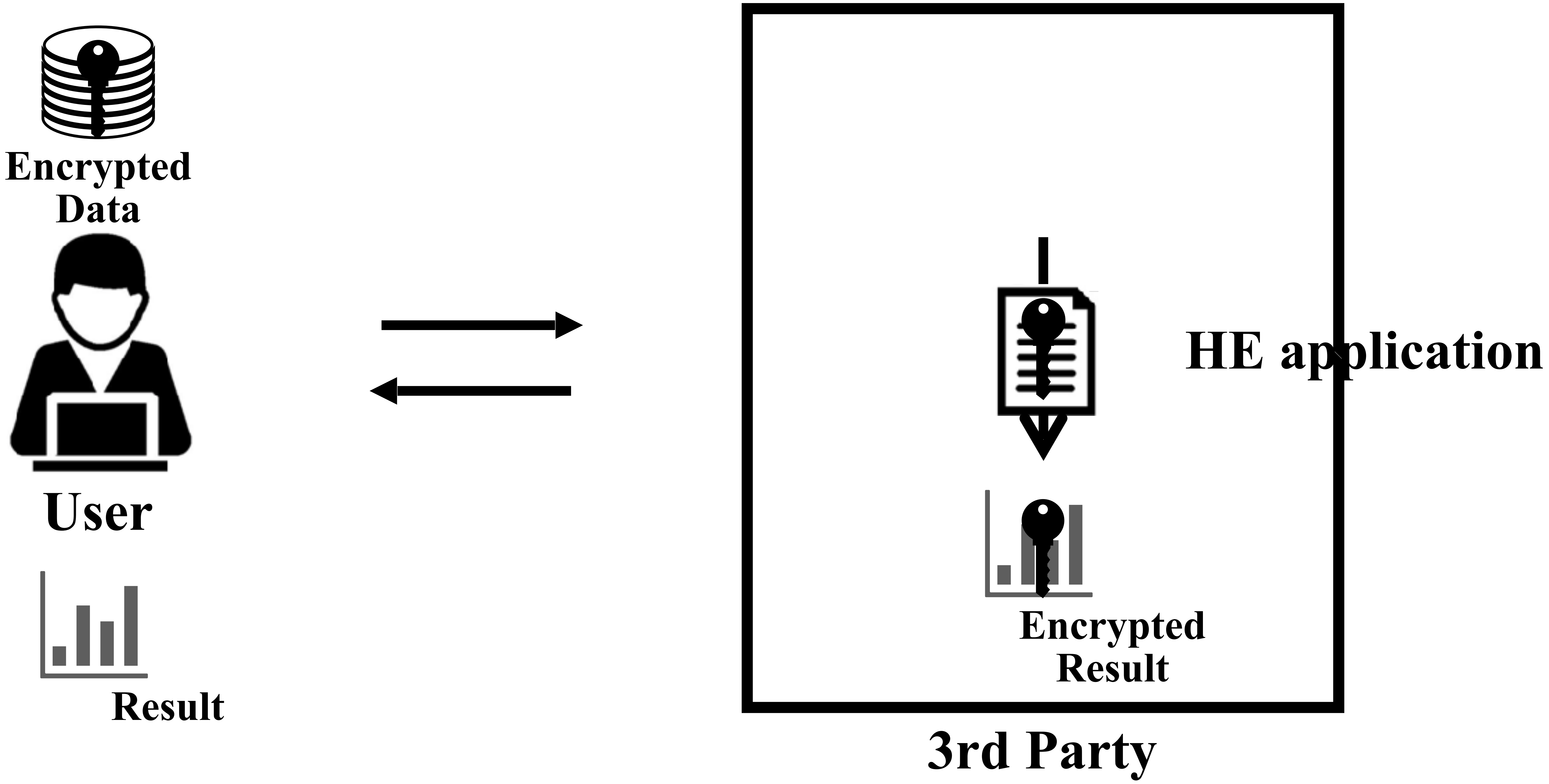
- Allows for computation on encrypted data
- Enables the outsourcing of private data storage/processing



Homomorphic Evaluation(HE) (1/3)

Privacy Preserving Secure Computation

- Allows for computation on encrypted data
- Enables the outsourcing of private data storage/processing



Homomorphic Evaluation(HE) (2/3)

Building HE applications

Choose parameters

Track noise level

Generate/manage keys, hints

Add maintenance operations

Write code in low-level HE instructions

requires
expertise

Application

HE developer

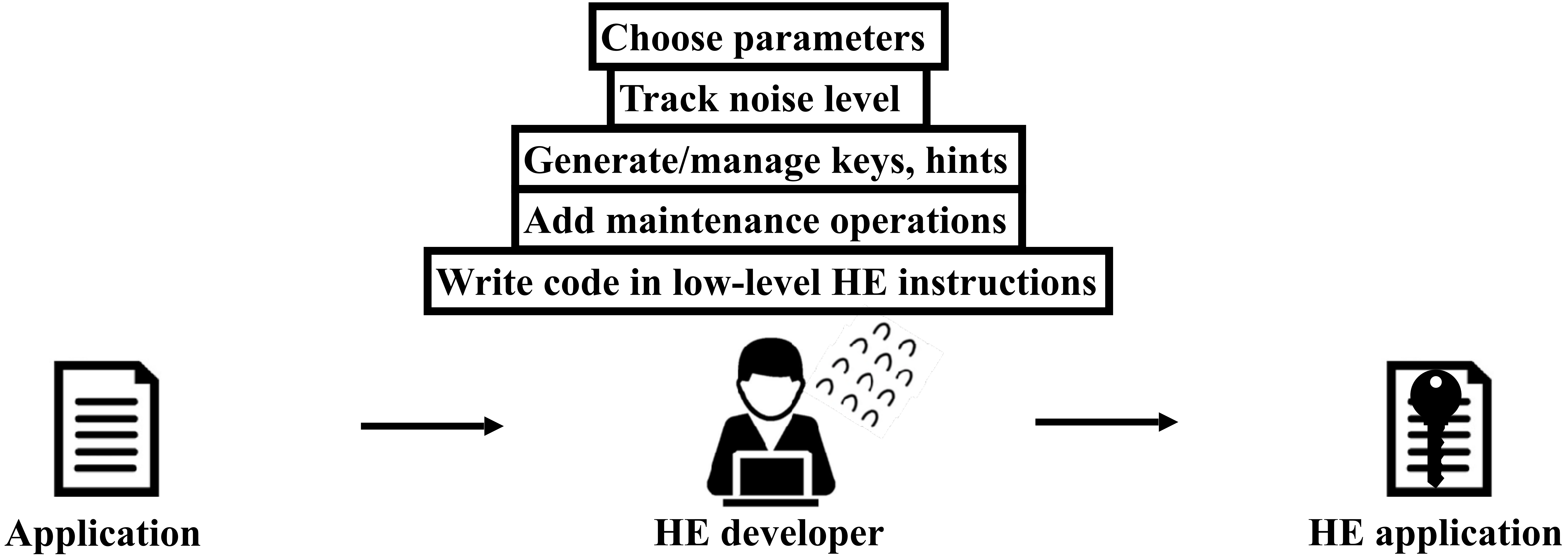
complicate
d

HE application

suboptim
al

Homomorphic Evaluation(HE) (3/3)

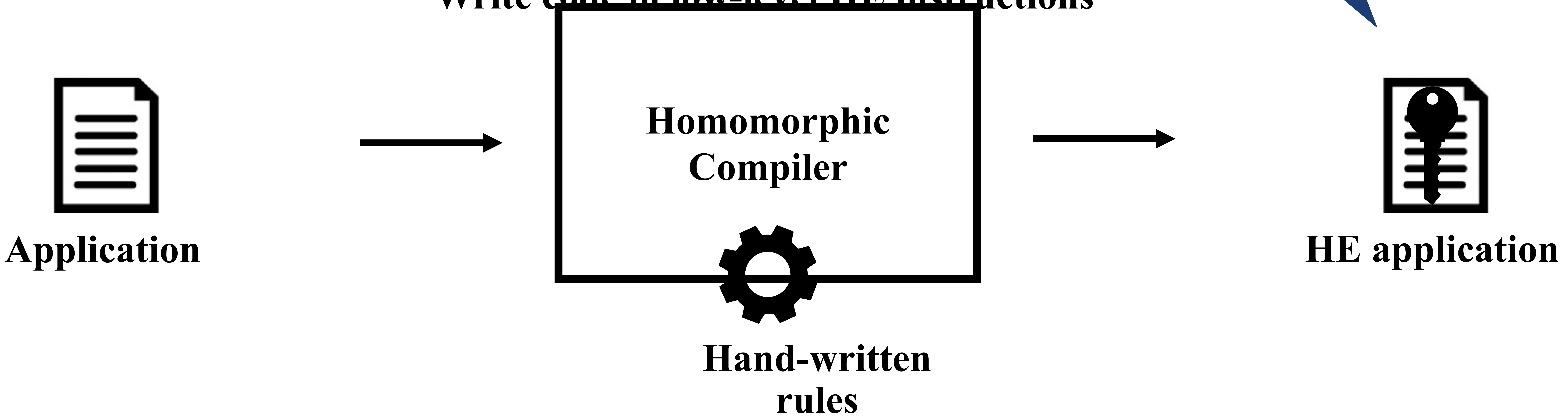
Existing Homomorphic Compiler



Homomorphic Evaluation(HE) (3/3)

Existing Homomorphic Compiler

- Generates HE applications automatically
- Optimization : several hand-written rules



Homomorphic Evaluation(HE) (2/3)

Code for homomorphic addition of two integers

```
#include "FHE.h"
#include "EncryptedArray.h"
#include <NTL/lzz_pXFactoring.h>
#include <fstream>
#include <sstream>
#include <sys/time.h>

int main(int argc, char **argv)
{
    long m=0, p=2, r=1; // Native plaintext space
                        // Computations will be 'modulo p'
    long L=16;          // Levels
    long c=3;           // Columns in key switching matrix
    long w=64;          // Hamming weight of secret key
    long d=0;
    long security = 128;
    ZZX G;
    m = FindM(security,L,c,p, d, 0, 0);
    FHEcontext context(m, p, r);
    buildModChain(context, L, c);
    FHESecKey secretKey(context);
    const FHEPubKey& publicKey = secretKey;
    G = context.alMod.getFactorsOverZZ()[0];
    secretKey.GenSecKey(w);
    addSome1DMatrices(secretKey);
    EncryptedArray ea(context, G);
    vector<long> v1;
    v1.push_back(atoi(argv[1]));
    Ctxt ct1(publicKey);
    ea.encrypt(ct1, publicKey, v1);
    v2.push_back(atoi(argv[2]));
    Ctxt ct2(publicKey);
    ea.encrypt(ct2, publicKey, v2);
    Ctxt ctSum = ct1;
    ctSum += ct2;
}
```

Manually written using HElib

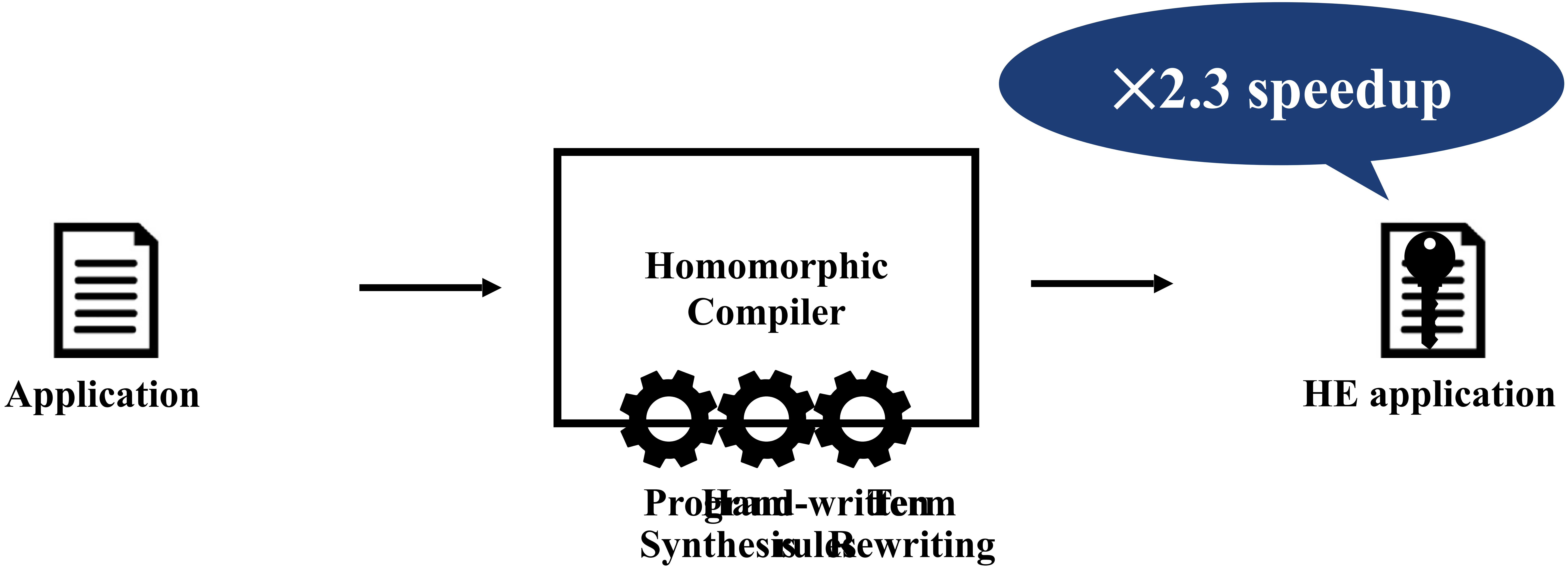
```
#include <iostream>
#include <fstream>
#include <integer.hxx>

int main()
{
        Integer8 a, b, c;

        cin >> a;
        cin >> b;
        c = a + b;

        cout << c;
        FINALIZE_CIRCUIT(blif_name);
}
```

Input to Cingulata (a HE compiler)

Our Contributions (1/2)

Automatic, Aggressive HE optimization Framework

- Generates HE applications automatically
- Optimization : search machine found rules by program synthesis + applying by term rewriting



Our Contributions (2/2)

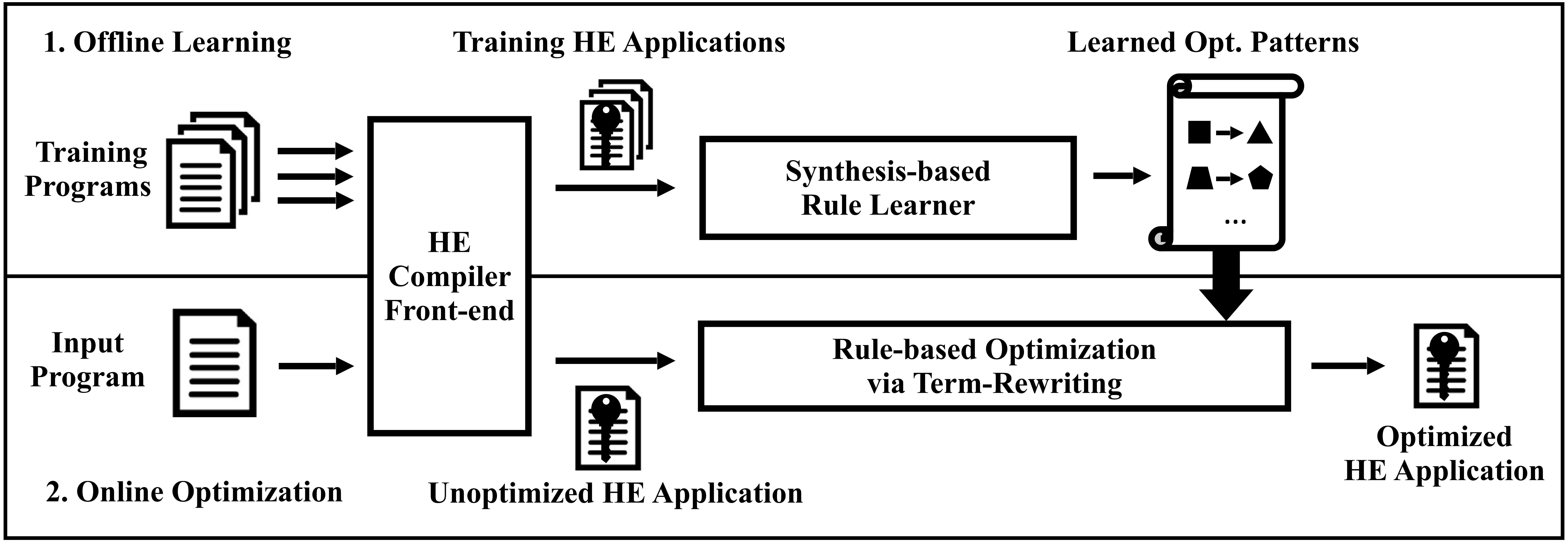
Automatic, Aggressive HE optimization Framework

- Learning Optimization Patterns by Program Synthesis
- Applying Learned Patterns by Term Rewriting
- Theorem : Semantic Preservation & Termination Guaranteed
- Performance (vs state-of-the-art HE Optimizer)
  - Optimized 22 out of 25 Applications (vs 15)
  - x5.43 Speedup in Maximum (vs x3.0)
  - x2.26 Speedup on Average (vs x1.53)
- Open Tool Available : https://github.com/dklee0501/Lobster

Our Lobster 

Learning to **O**ptimize **B**oolean circuit using **S**ynthesis and **TE**rm **R**ewriting

- Offline Learning via Program Synthesis + Online Optimization via Term Rewriting



Simple HE Scheme

- Based on approximate common divisor problem
- p : integer as a secret key
- q : random integer
- $r(\ll |p|)$: random noise for security

$$Enc_p(\mu \in \{0,1\}) = pq + 2r + \mu$$
$$Dec_p(c) = \underline{(c \bmod p)}\,\underline{\bmod 2}$$
$$Dec_p(Enc_p(\mu)) = Dec_p(\cancel{pq} + \cancel{2r} + \mu) = \mu$$

- For ciphertexts $\underline{\mu_i} \leftarrow Enc_p(\mu_i)$, the following holds

$$Dec_p(\underline{\mu_1} + \underline{\mu_2}) = \mu_1 + \mu_2$$
$$Dec_p(\underline{\mu_1} \times \underline{\mu_2}) = \mu_1 \times \mu_2$$

- The scheme can evaluate all boolean circuits as $+$ and $\times$ in $\mathbb{Z}_2 = \{0,1\}$ are equal to XOR and AND

Performance Hurdle : Growing Noise

- Noise increases during homomorphic operations.
- For $\underline{\mu_i} = pq_i + 2r_i + \mu_i$

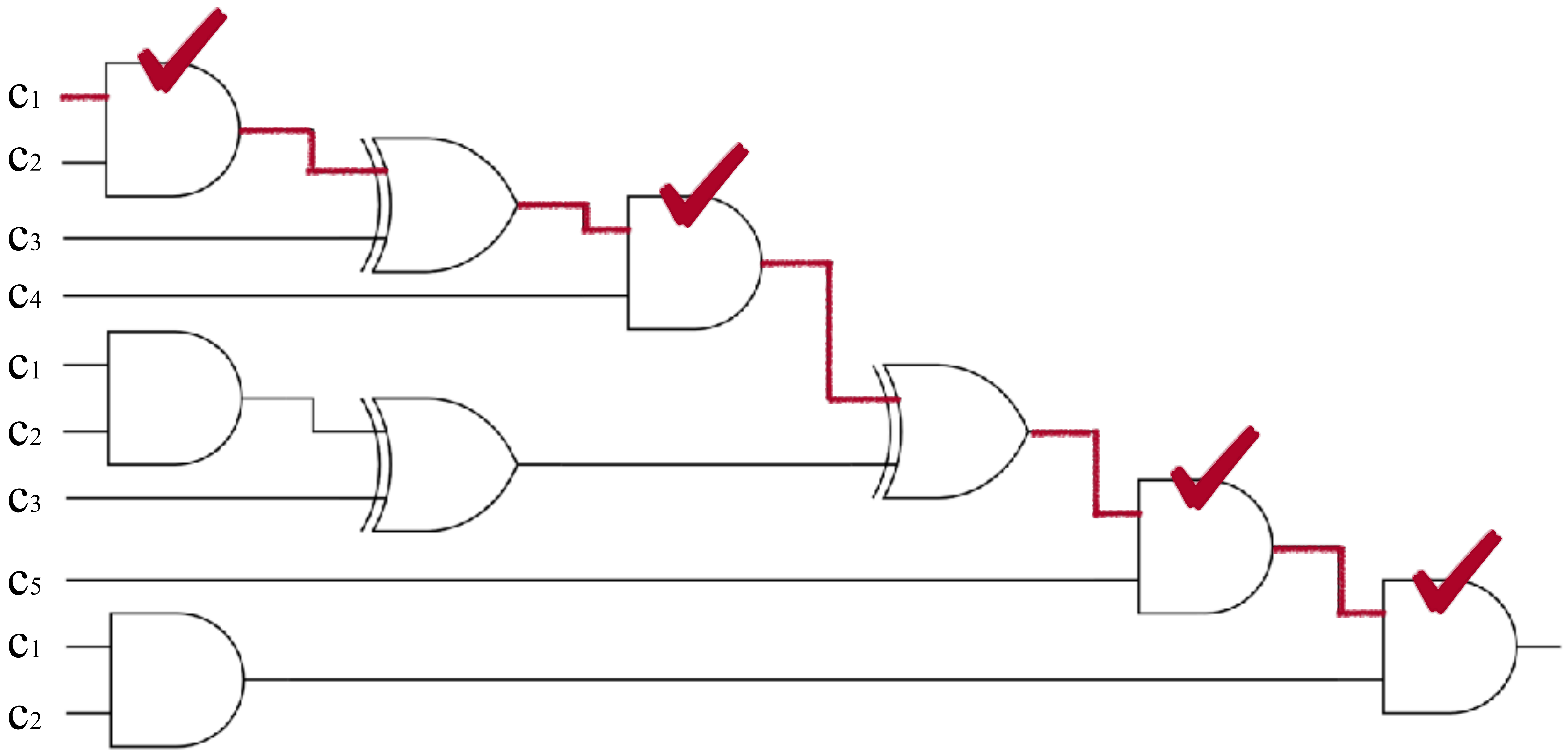
$$
\begin{aligned}
\underline{\mu_1} + \underline{\mu_2} &= p(q_1 + q_2) + \boxed{2(r_1 + r_2) + (\mu_1 + \mu_2)} \ \text{double increase} \\
\underline{\mu_1} \times \underline{\mu_2} &= p(pq_1q_2 + \cdots) + \boxed{2(2r_1r_2 + r_1\mu_2 + r_2\mu_1) + (\mu_1 \times \mu_2)} \ \text{quadratic increase}
\end{aligned}
$$

noise

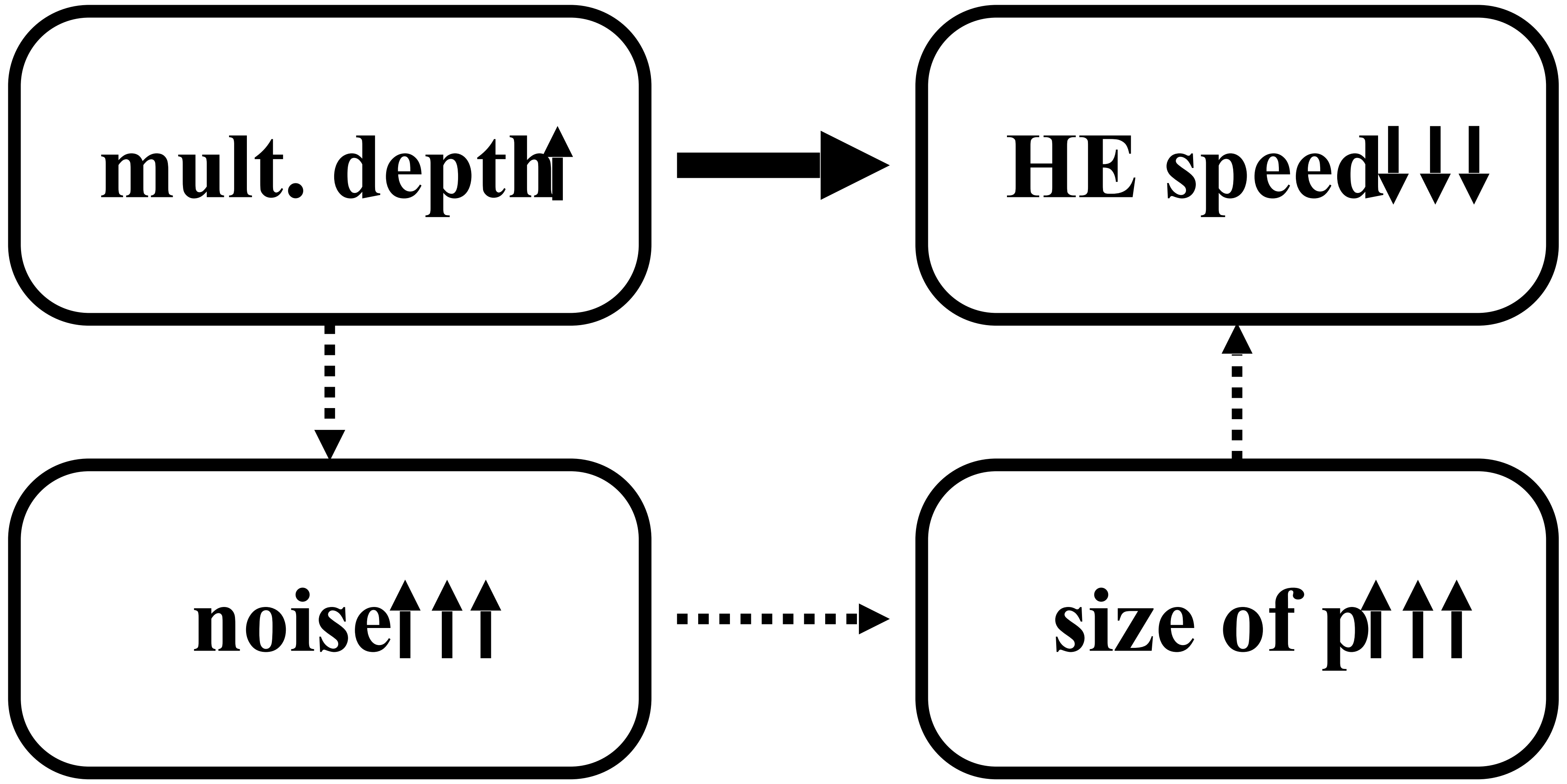
- **if (noise $>$ p) then incorrect results**

Multiplicative Depth : a Decisive Performance Factor

- Multiplicative depth : the maximum number of sequential multiplications from input to output

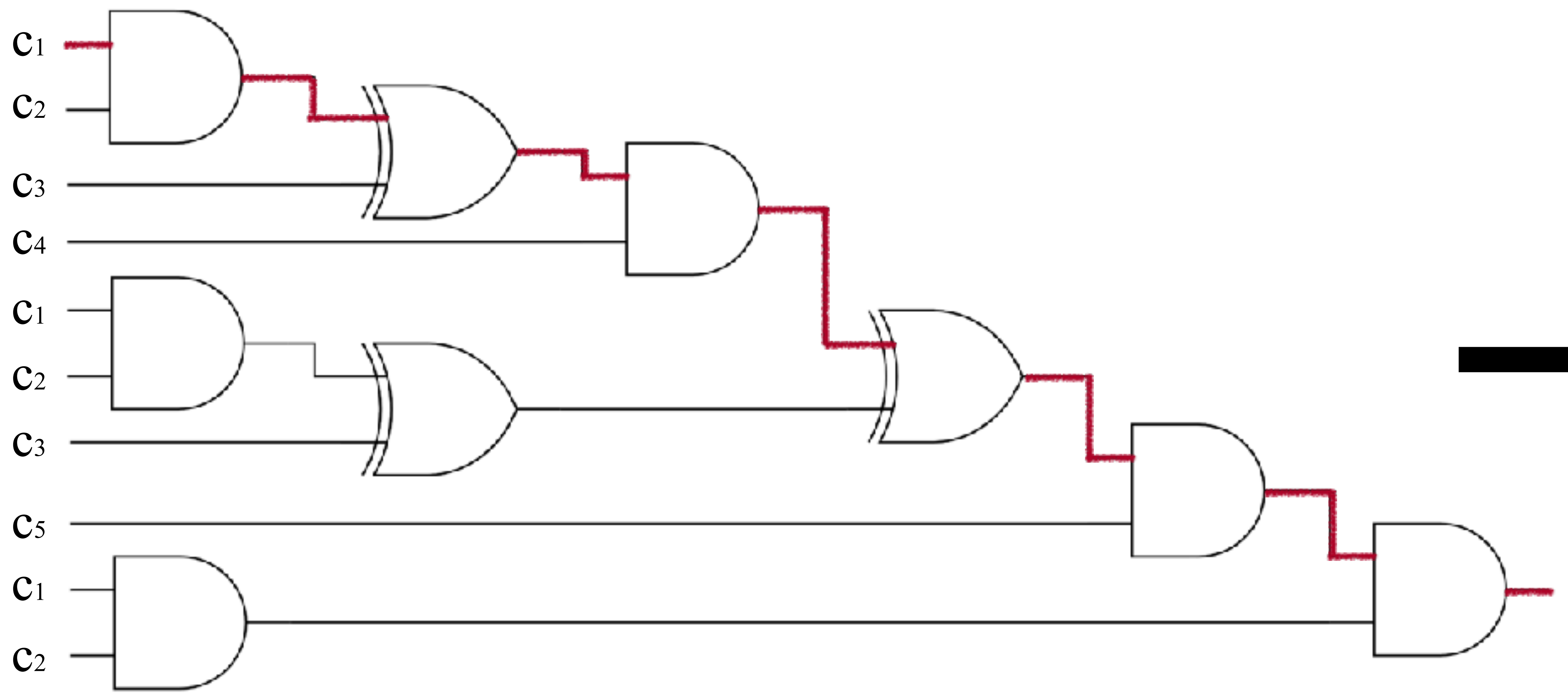


depth 4

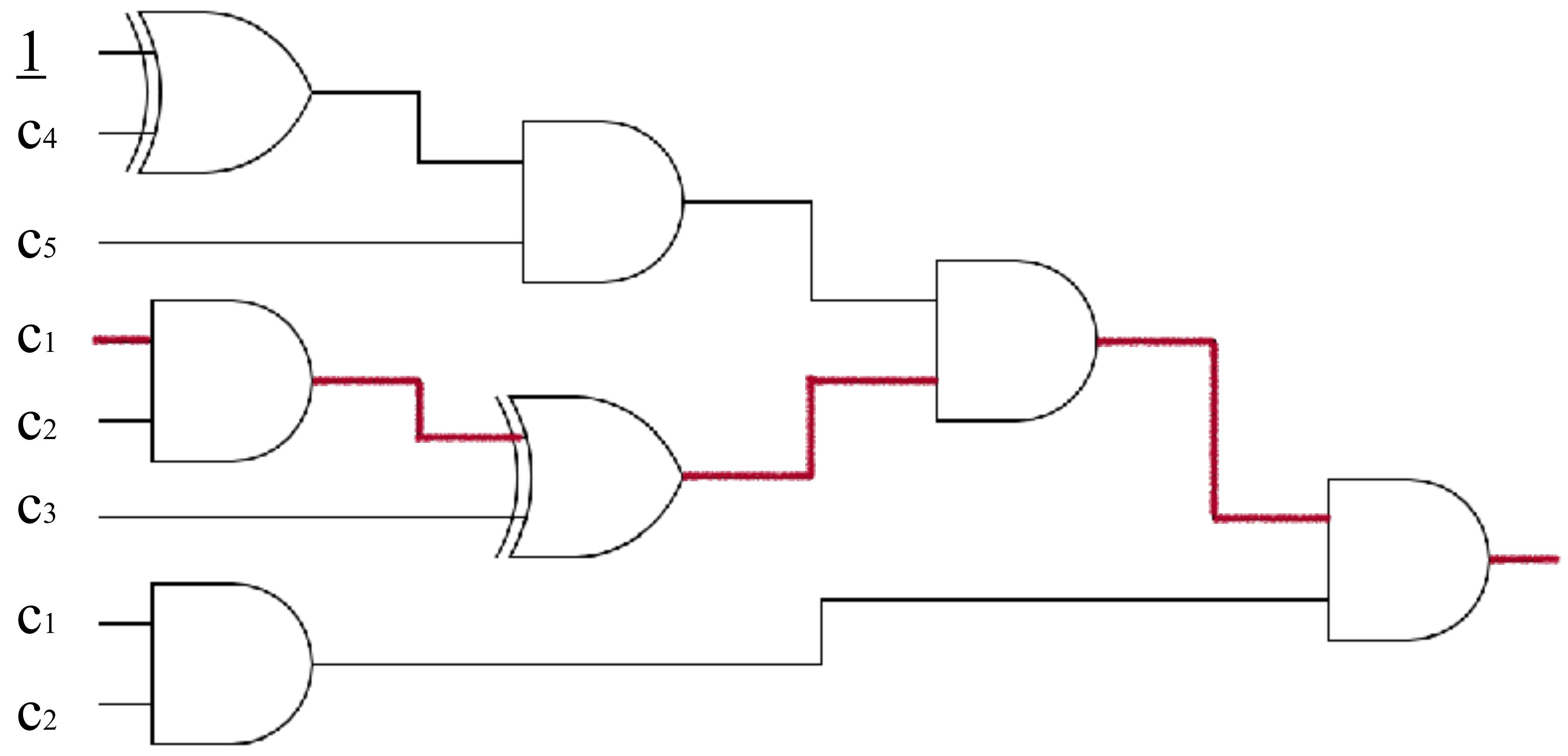


What is HE optimization?

- Finding a new circuit that has smaller mult. depth

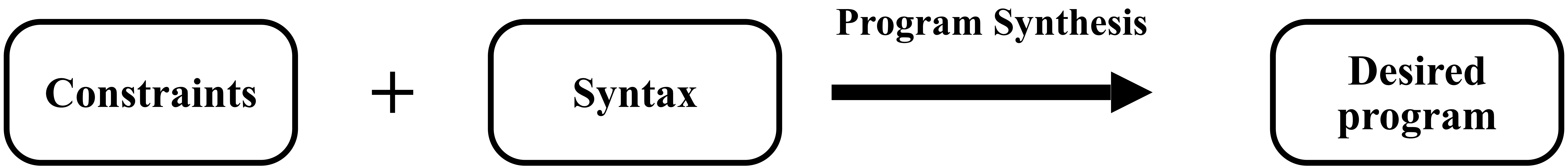


depth 4



depth 3

HE optimization via Synthesis

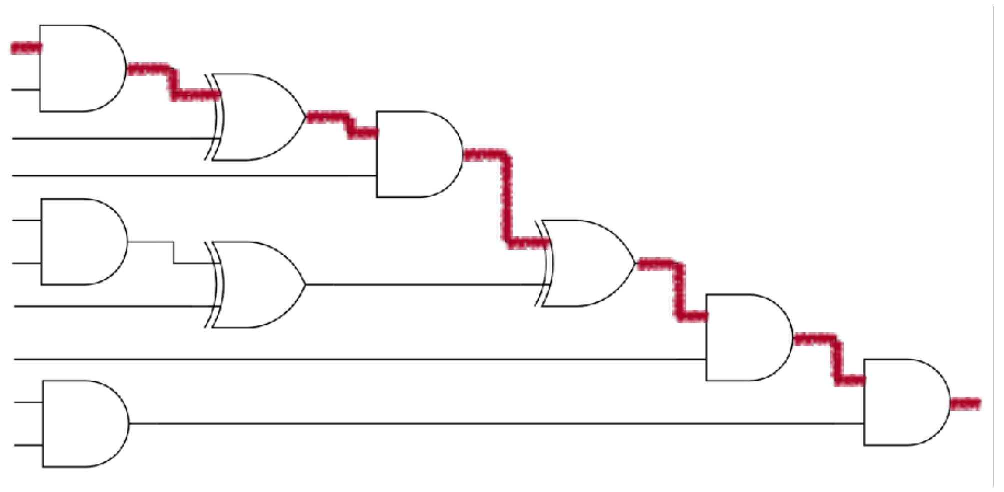


HE optimization via Synthesis

Constraints + Syntax

Program Synthesis

Desired program

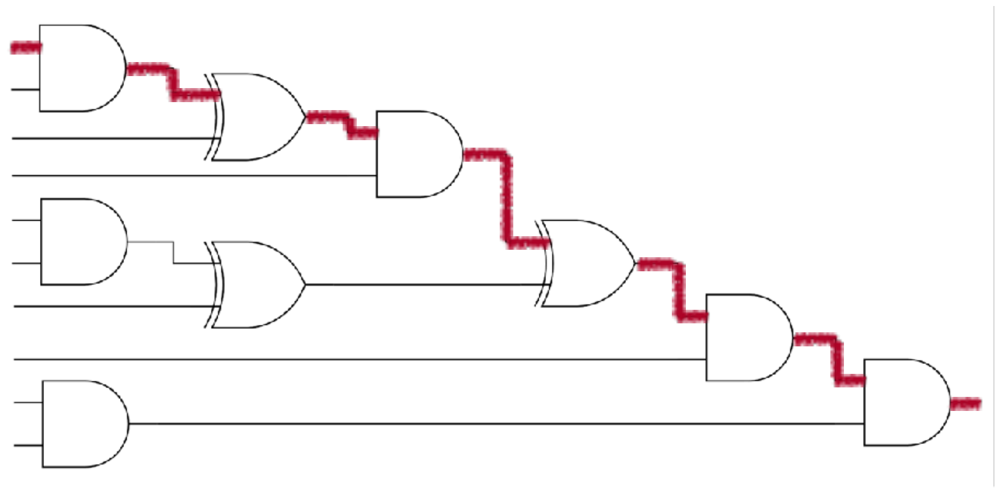


depth 4

same semantics

HE optimization via Synthesis

Constraints + **Syntax** —**Program Synthesis**→ **Desired program**



depth 4

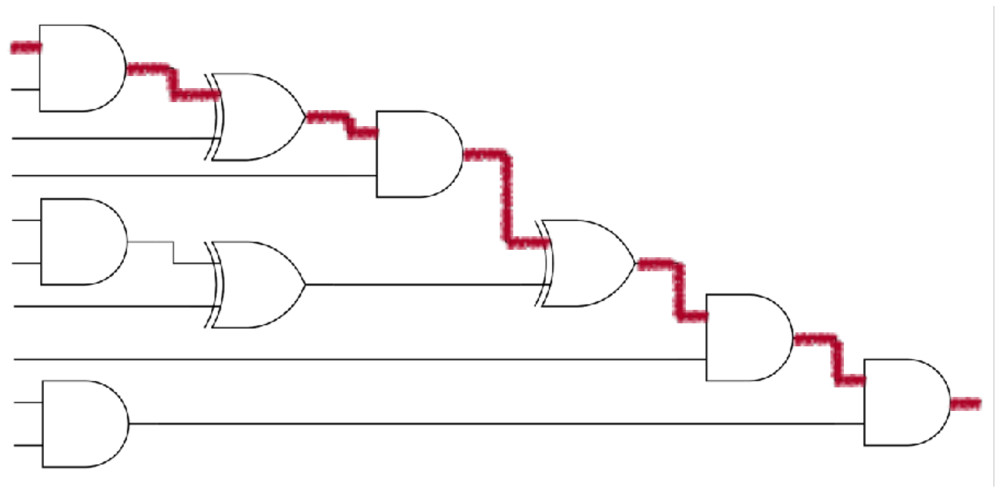
same semantics

$$
\begin{array}{rcl}
S & \rightarrow & d_3 \\
d_3 & \rightarrow & d_2 \wedge d_2 \mid d_3 \oplus d_3 \mid d_2 \\
d_2 & \rightarrow & d_1 \wedge d_1 \mid d_2 \oplus d_2 \mid d_1 \\
d_1 & \rightarrow & d_0 \wedge d_0 \mid d_1 \oplus d_1 \mid d_0 \\
d_0 & \rightarrow & 0 \mid 1 \mid c_1 \mid c_2 \mid c_3 \mid c_4 \mid c_5
\end{array}
$$

depth-restricting syntax

HE optimization via Synthesis

Constraints + **Syntax** $\xrightarrow{\text{Program Synthesis}}$ **Desired program**

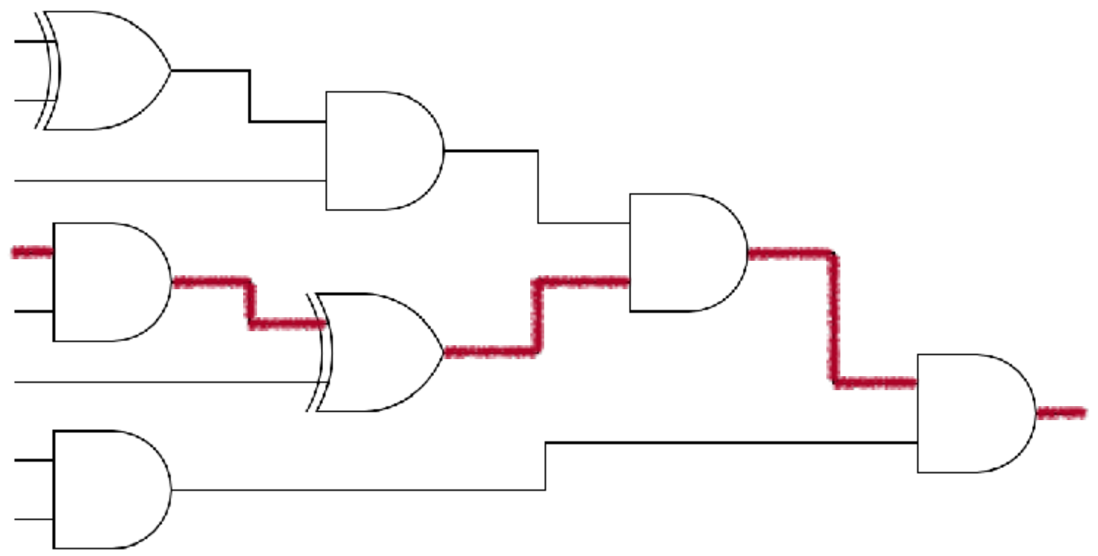


depth 4

same semantics

$$
\begin{array}{lcl}
S & \rightarrow & d_3 \\
d_3 & \rightarrow & d_2 \land d_2 \mid d_3 \oplus d_3 \mid d_2 \\
d_2 & \rightarrow & d_1 \land d_1 \mid d_2 \oplus d_2 \mid d_1 \\
d_1 & \rightarrow & d_0 \land d_0 \mid d_1 \oplus d_1 \mid d_0 \\
d_0 & \rightarrow & 0 \mid 1 \mid c_1 \mid c_2 \mid c_3 \mid c_4 \mid c_5
\end{array}
$$

depth-restricting syntax

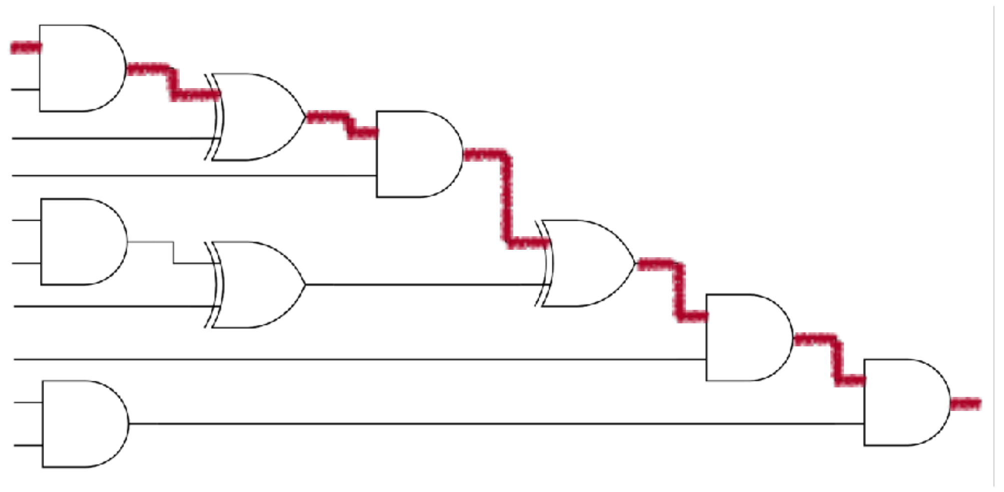


depth 3

optimized HE circuit

HE optimization via Synthesis

Constraints + **Syntax** $\xrightarrow{\text{Optimizing Synthesis}}$ **Desired program**

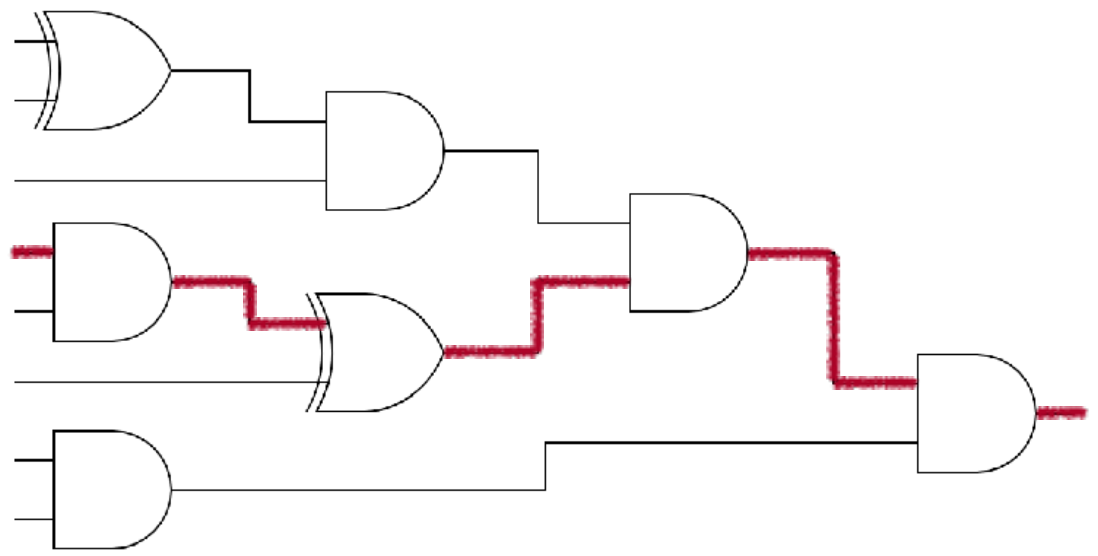


depth 4

same semantics

$$
\begin{array}{rcl}
S & \rightarrow & d_3 \\
d_3 & \rightarrow & d_2 \wedge d_2 \mid d_3 \oplus d_3 \mid d_2 \\
d_2 & \rightarrow & d_1 \wedge d_1 \mid d_2 \oplus d_2 \mid d_1 \\
d_1 & \rightarrow & d_0 \wedge d_0 \mid d_1 \oplus d_1 \mid d_0 \\
d_0 & \rightarrow & 0 \mid 1 \mid c_1 \mid c_2 \mid c_3 \mid c_4 \mid c_5
\end{array}
$$

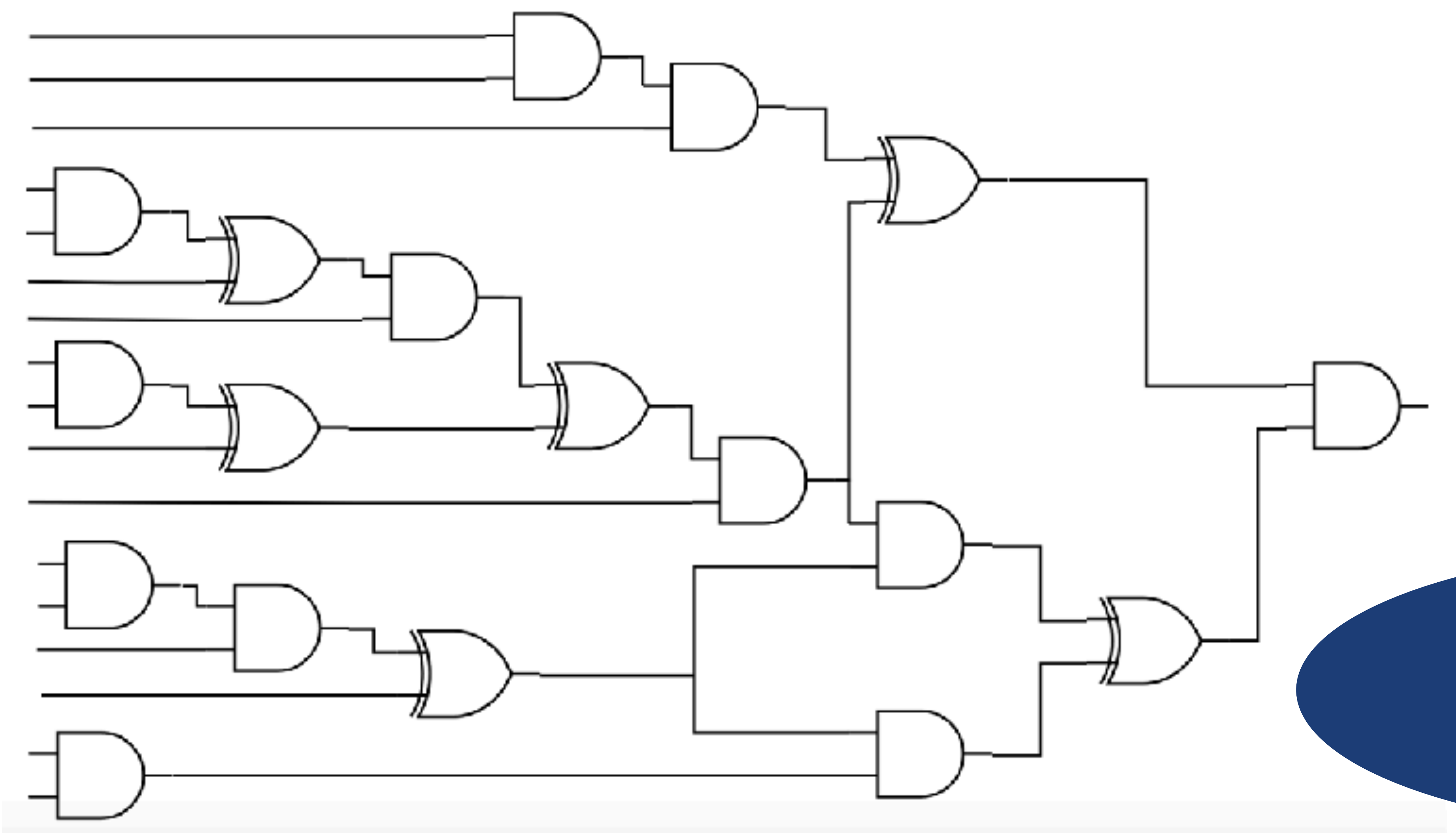
depth-restricting syntax



depth 3

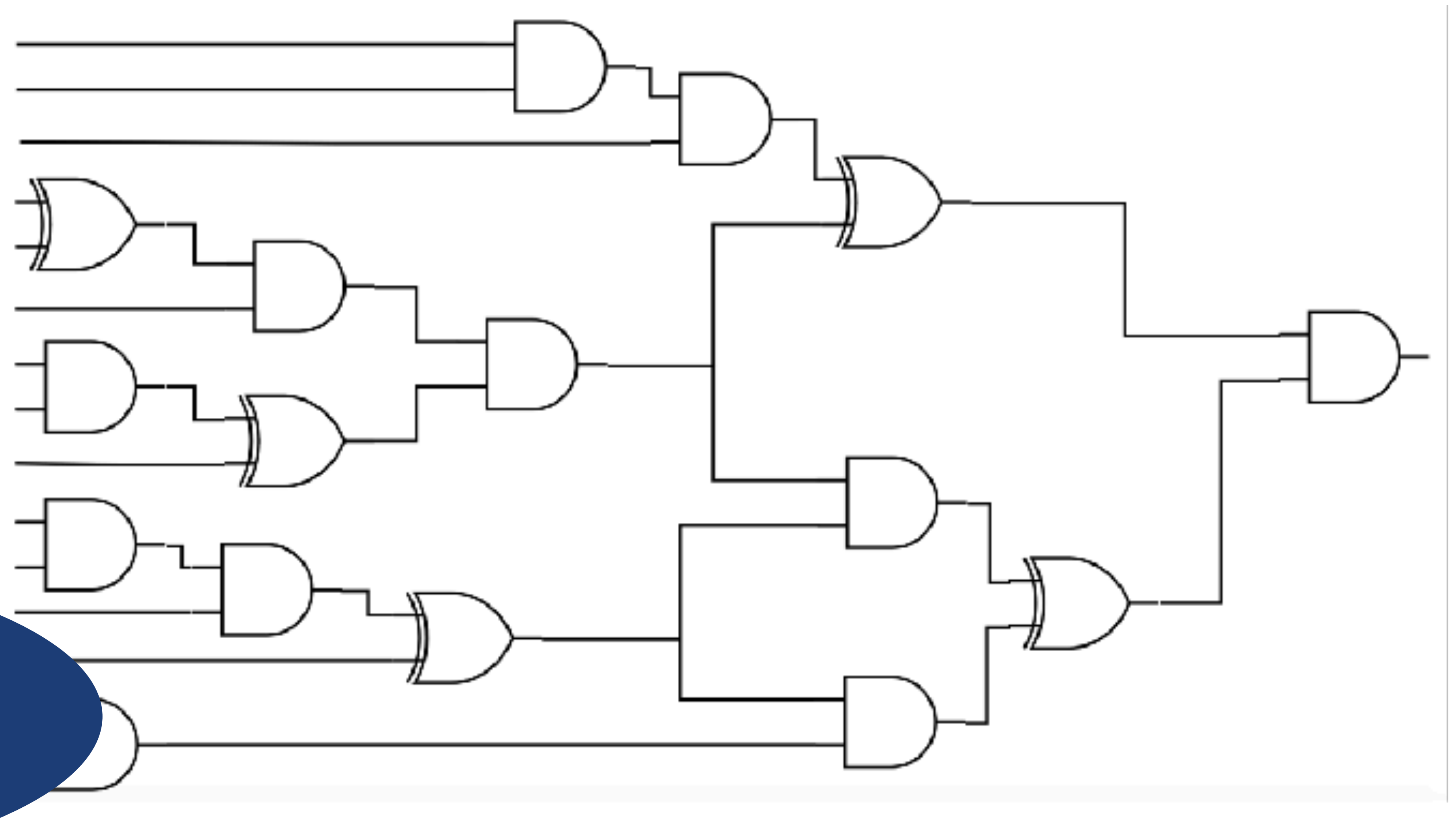
optimized HE circuit

Hurdle : Synthesis Scalability

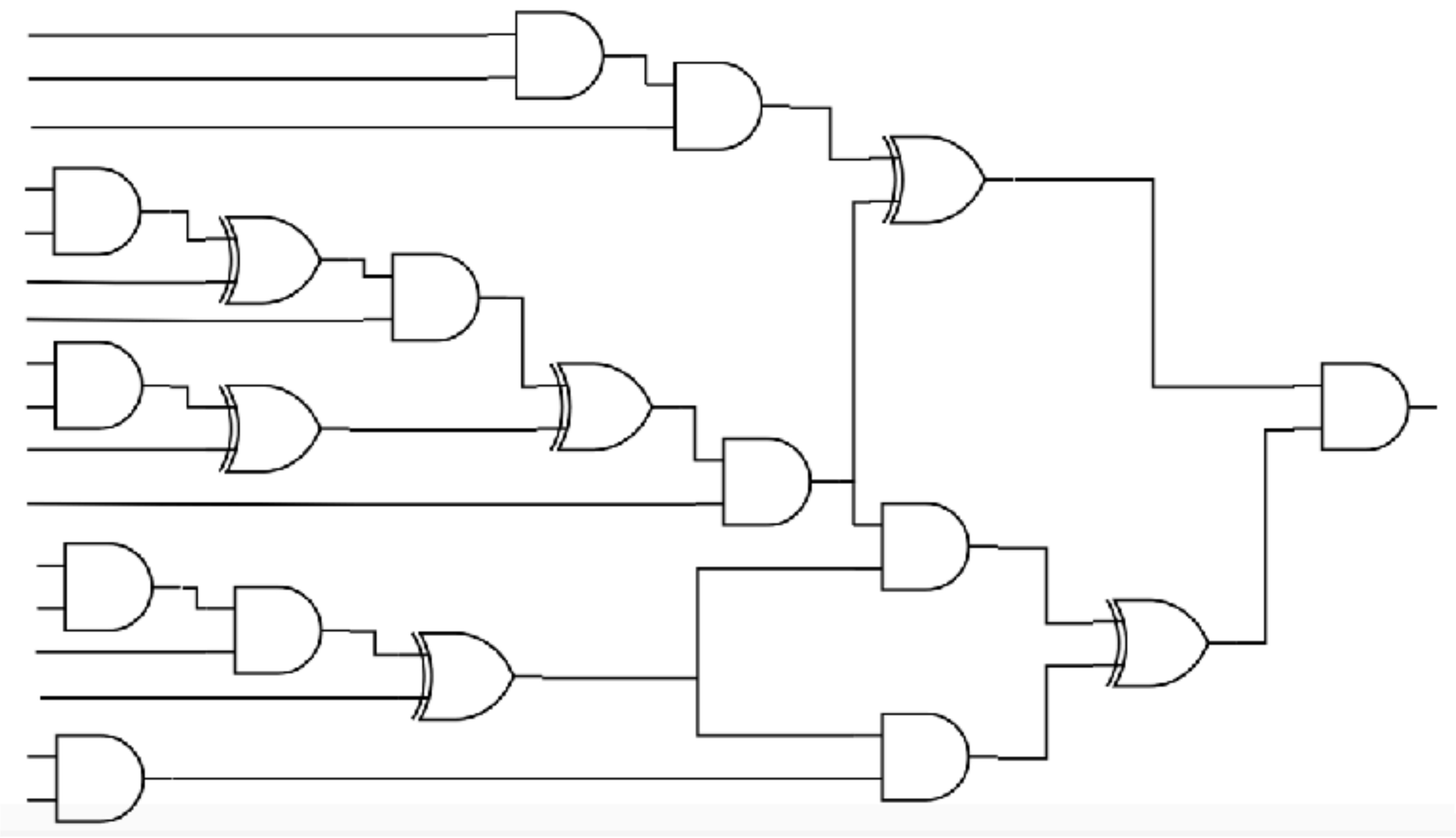


Optimizing
Synthesis

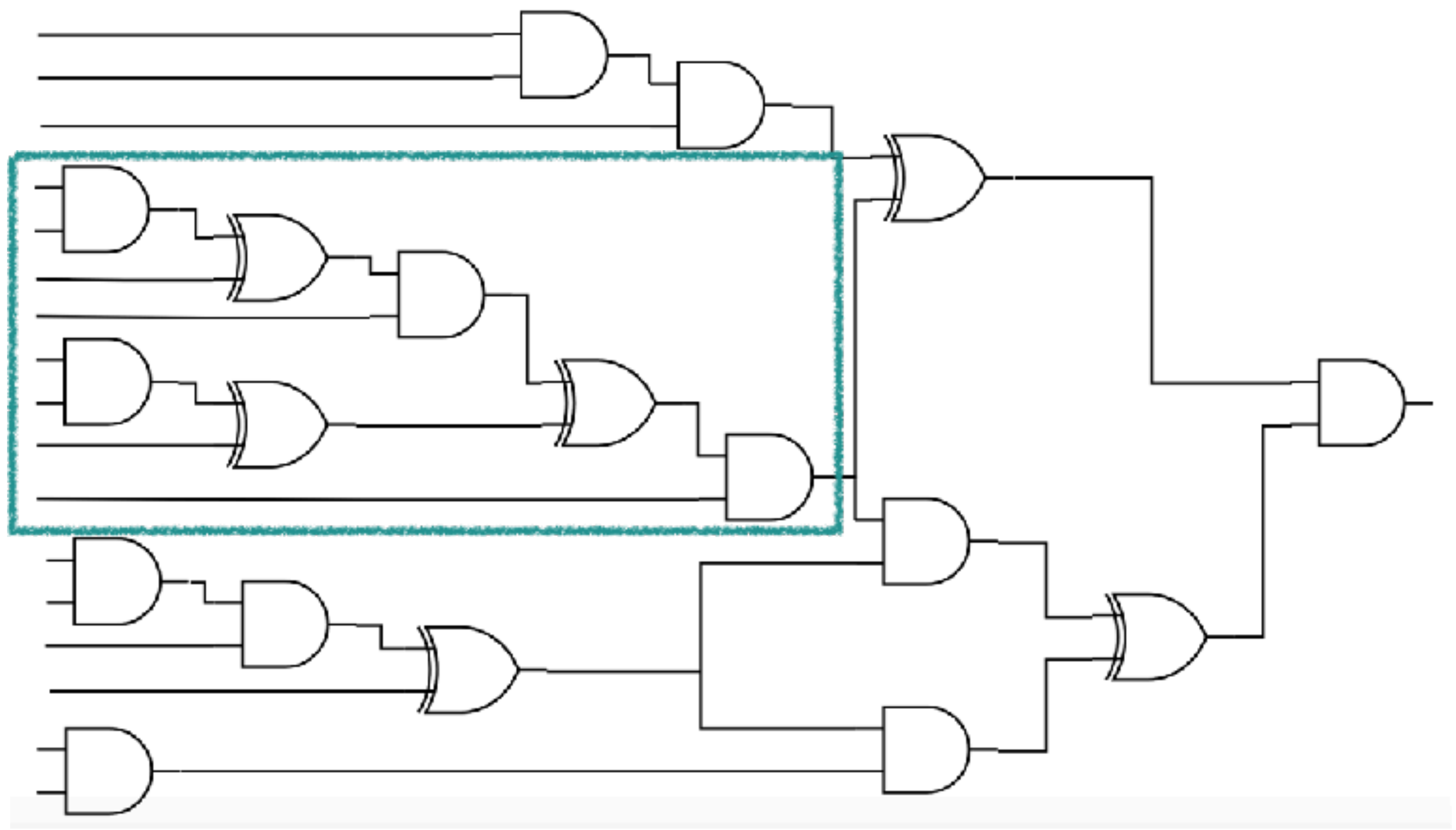
too slow



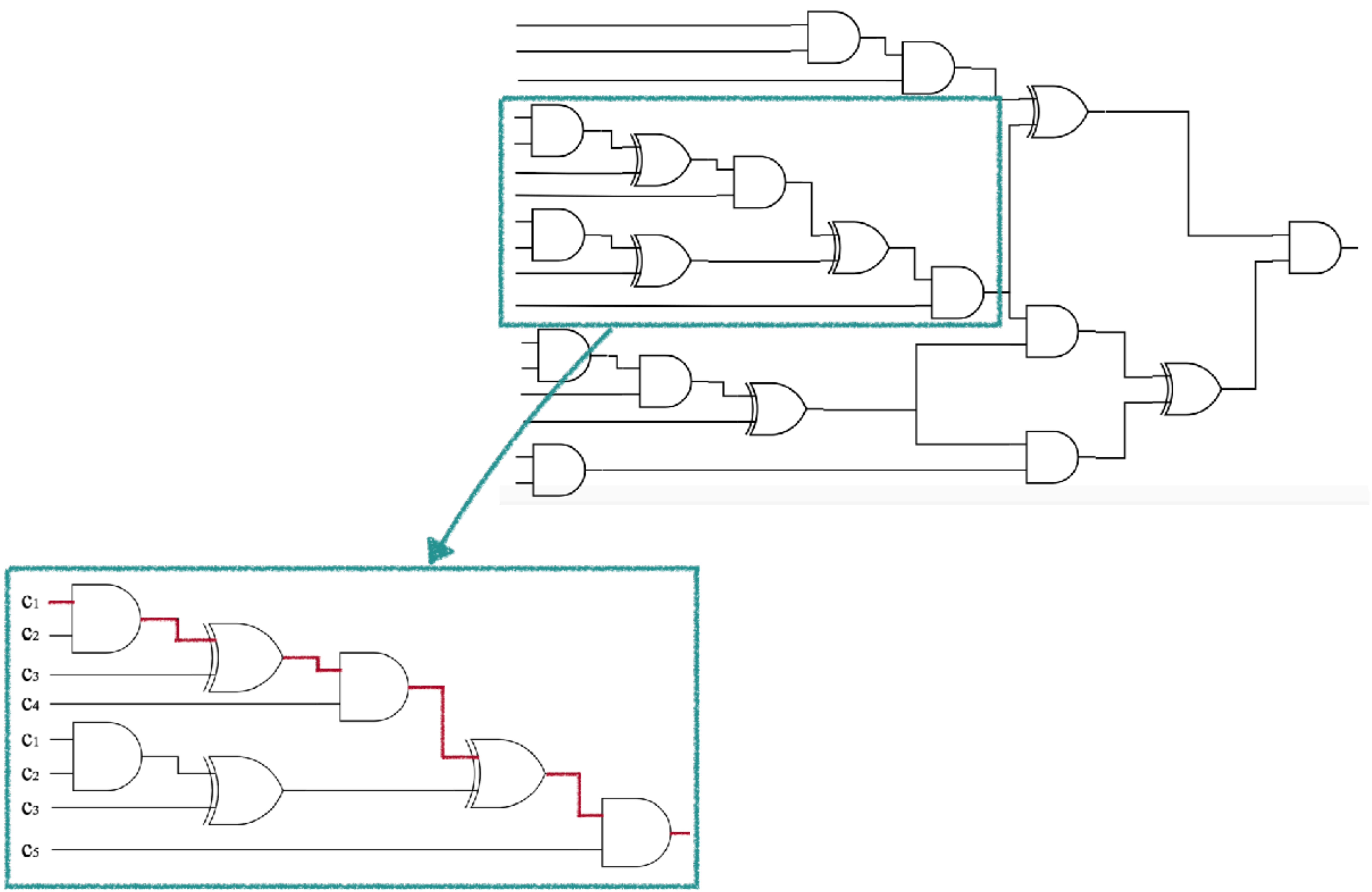
Solution1 : Synthesis via Localization



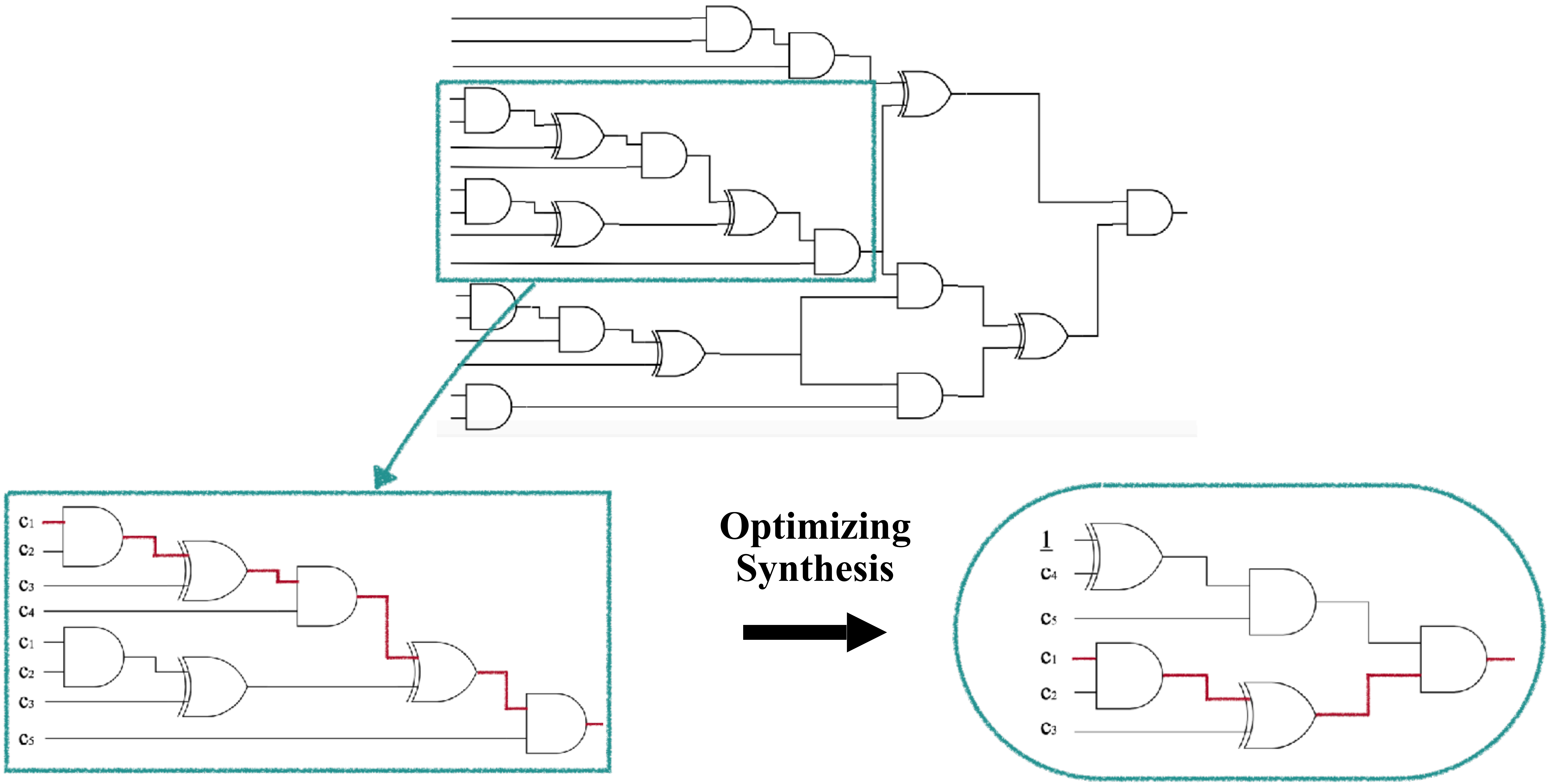
Solution1 : Synthesis via Localization



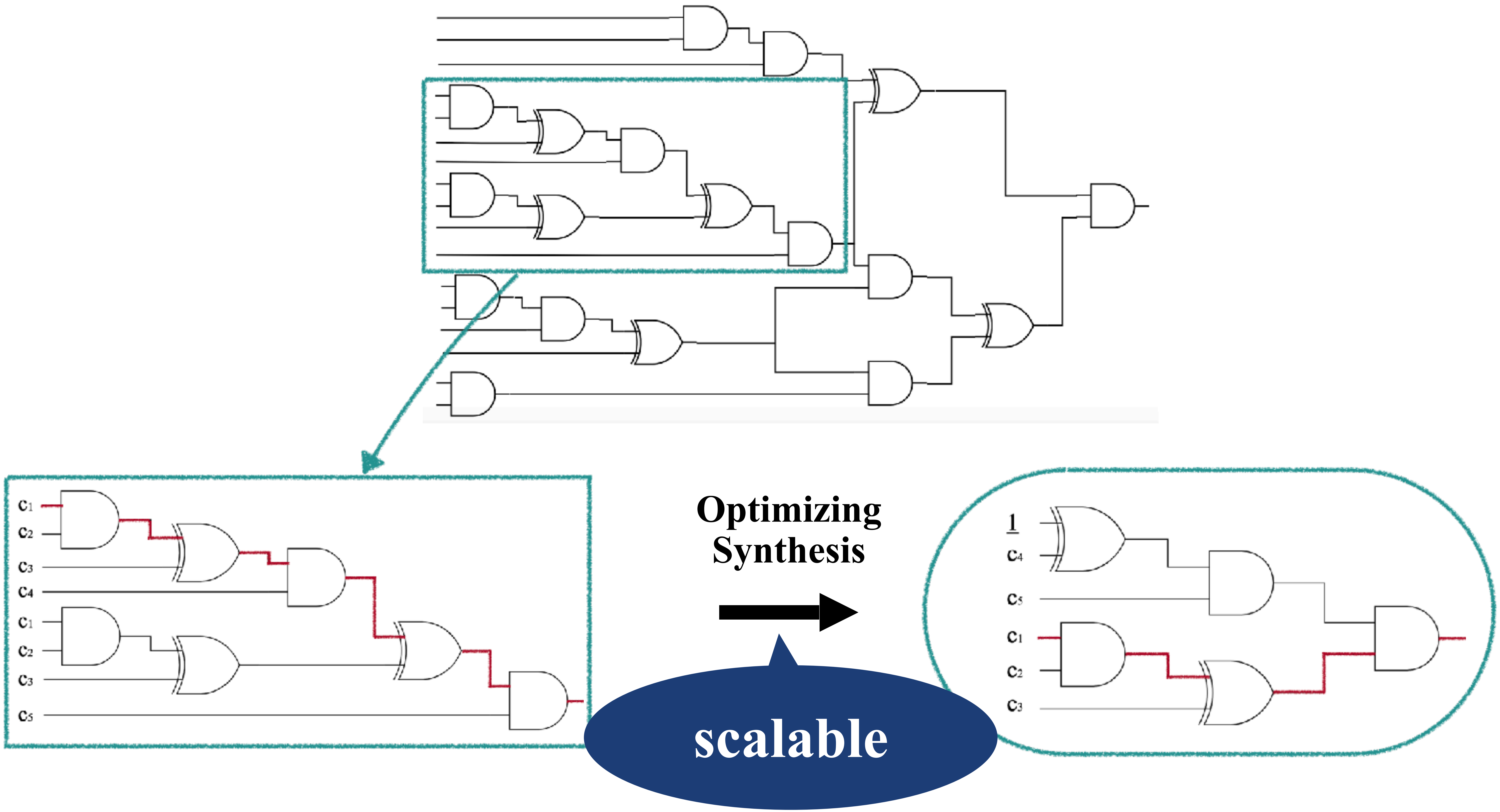
Solution1 : Synthesis via Localization



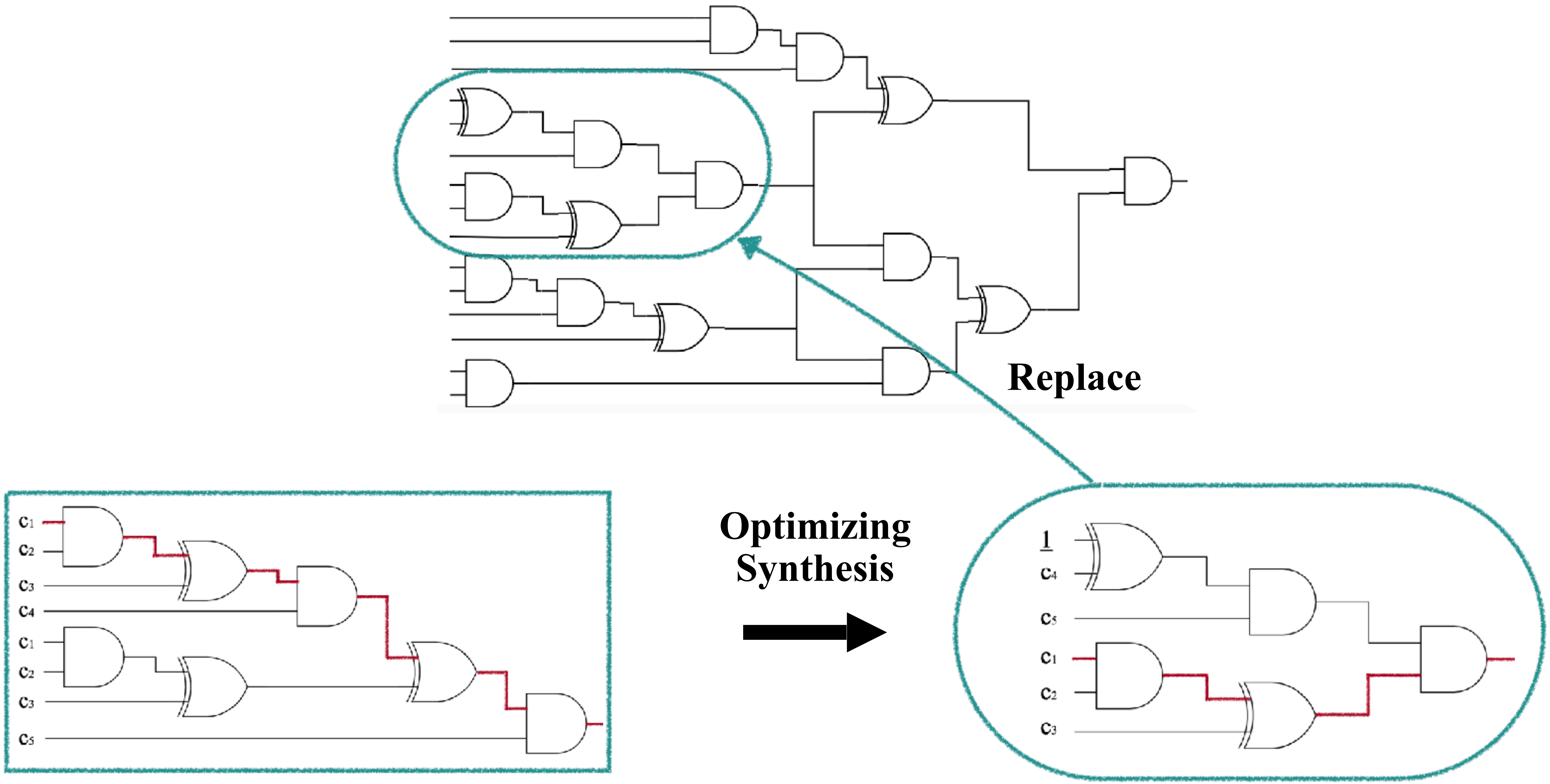
Solution1 : Synthesis via Localization



Solution1 : Synthesis via Localization



Solution1 : Synthesis via Localization

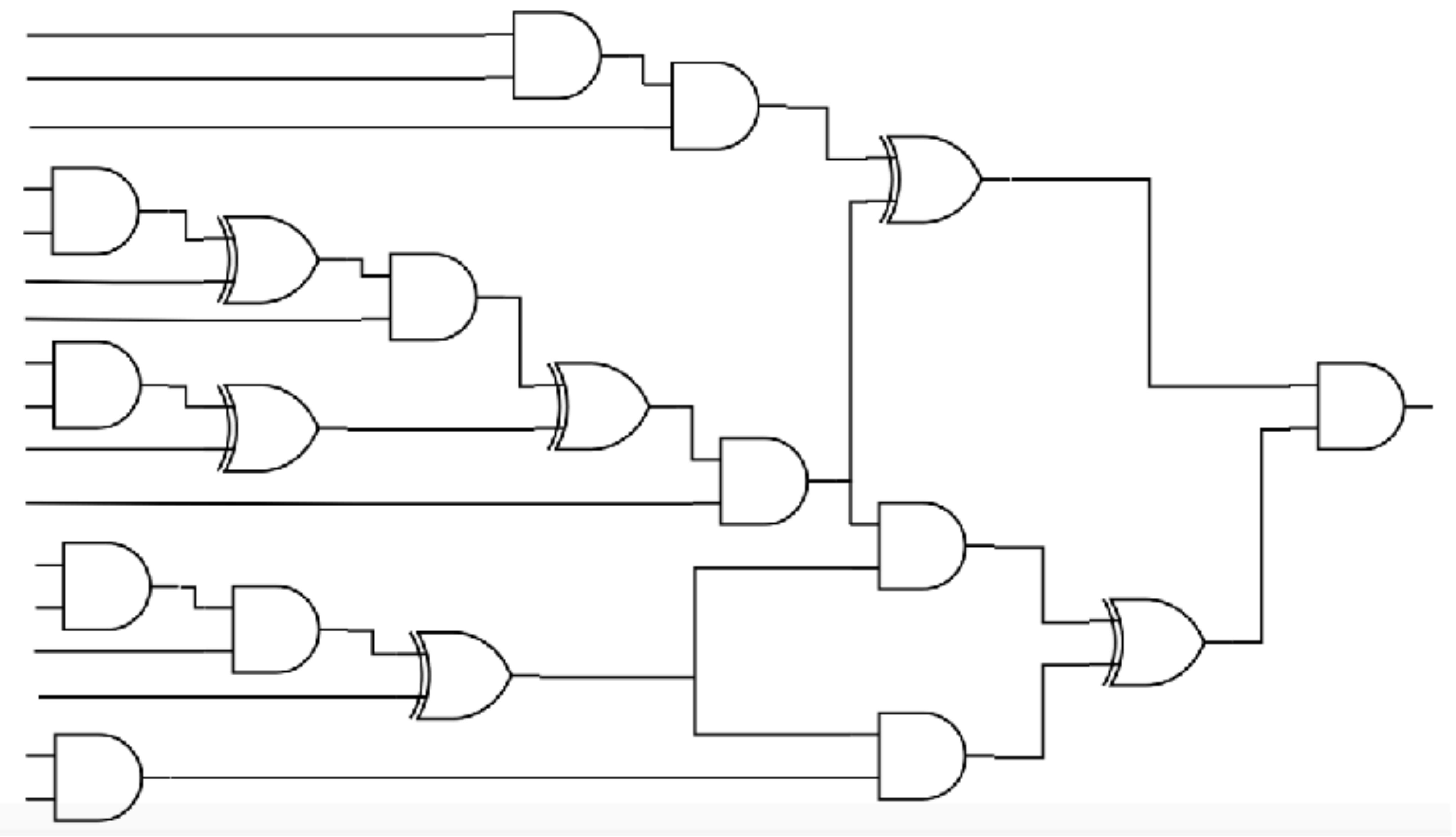


Solution 2: Learning Successful Synthesis Patterns

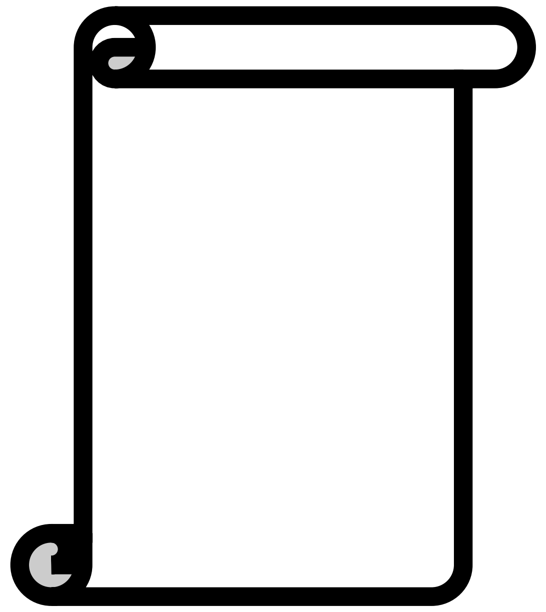
- Offline Learning
  - Collect successful synthesis patterns
- Online Optimization
  - Applying the patterns by term rewriting

Offline Learning to Collect Opt. Patterns

Training HE Applications

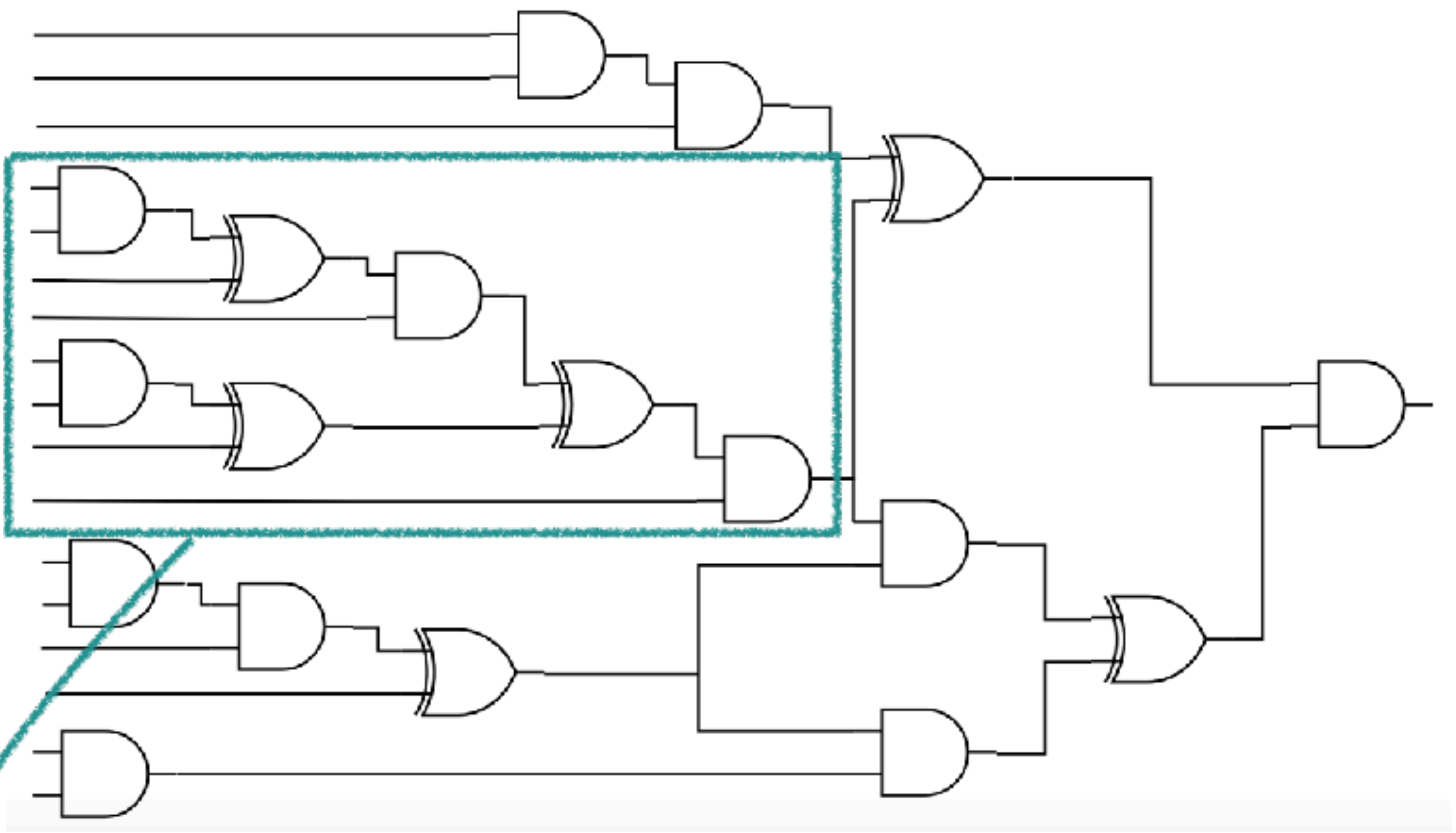


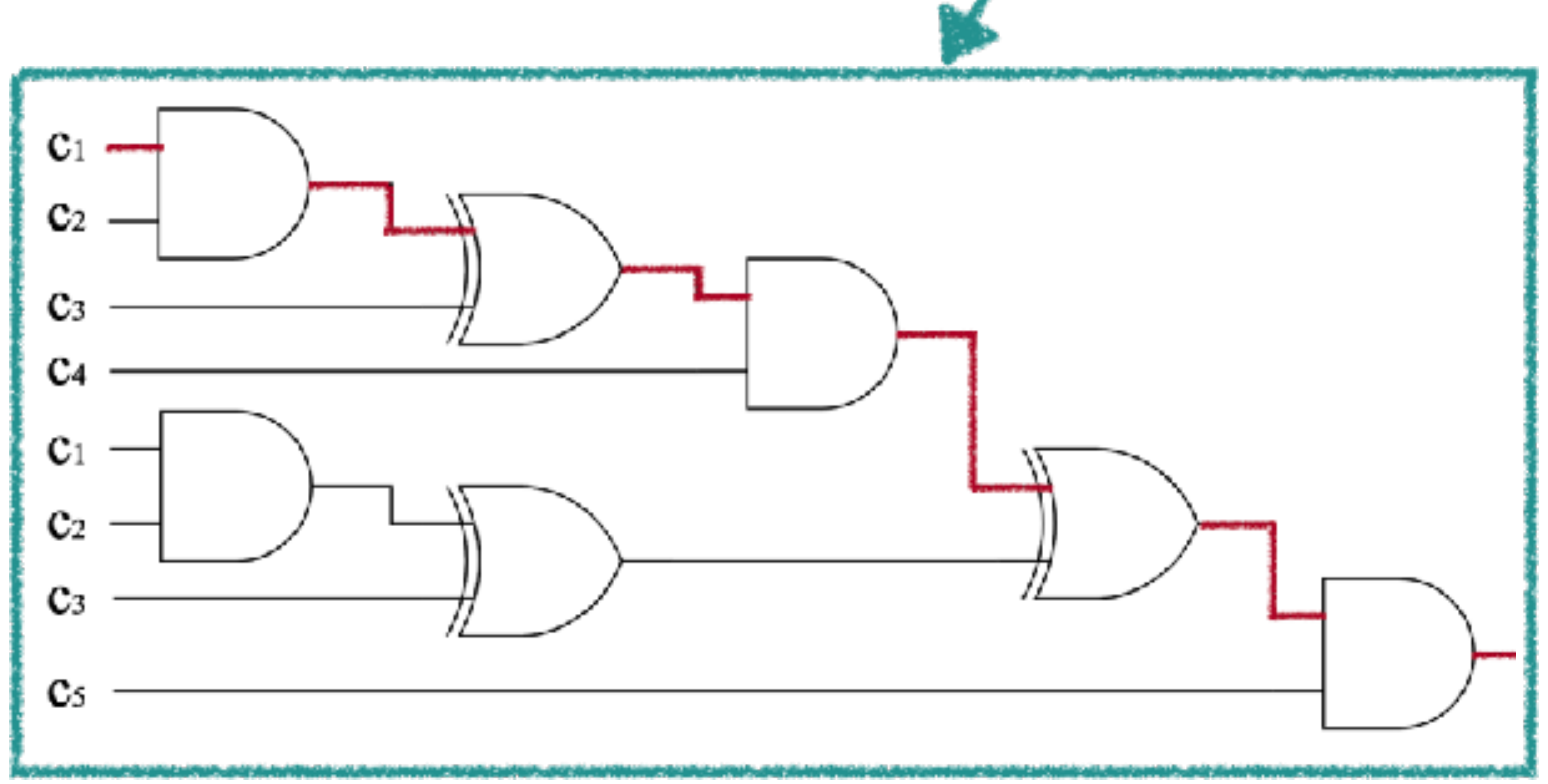
Collected Opt. Patterns



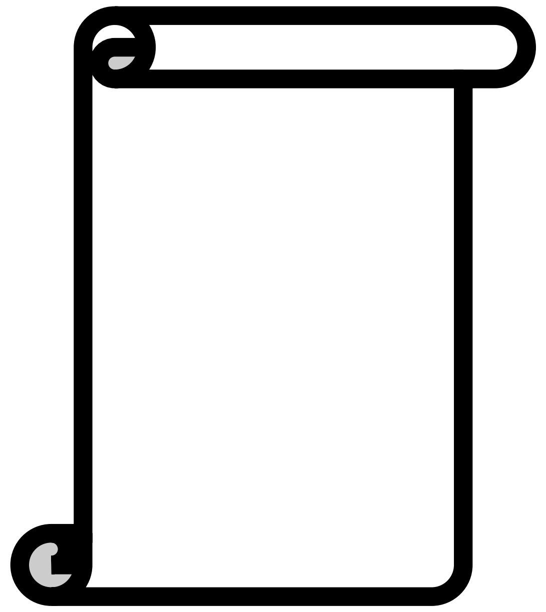
Offline Learning to Collect Opt. Patterns

Training
HE Applications

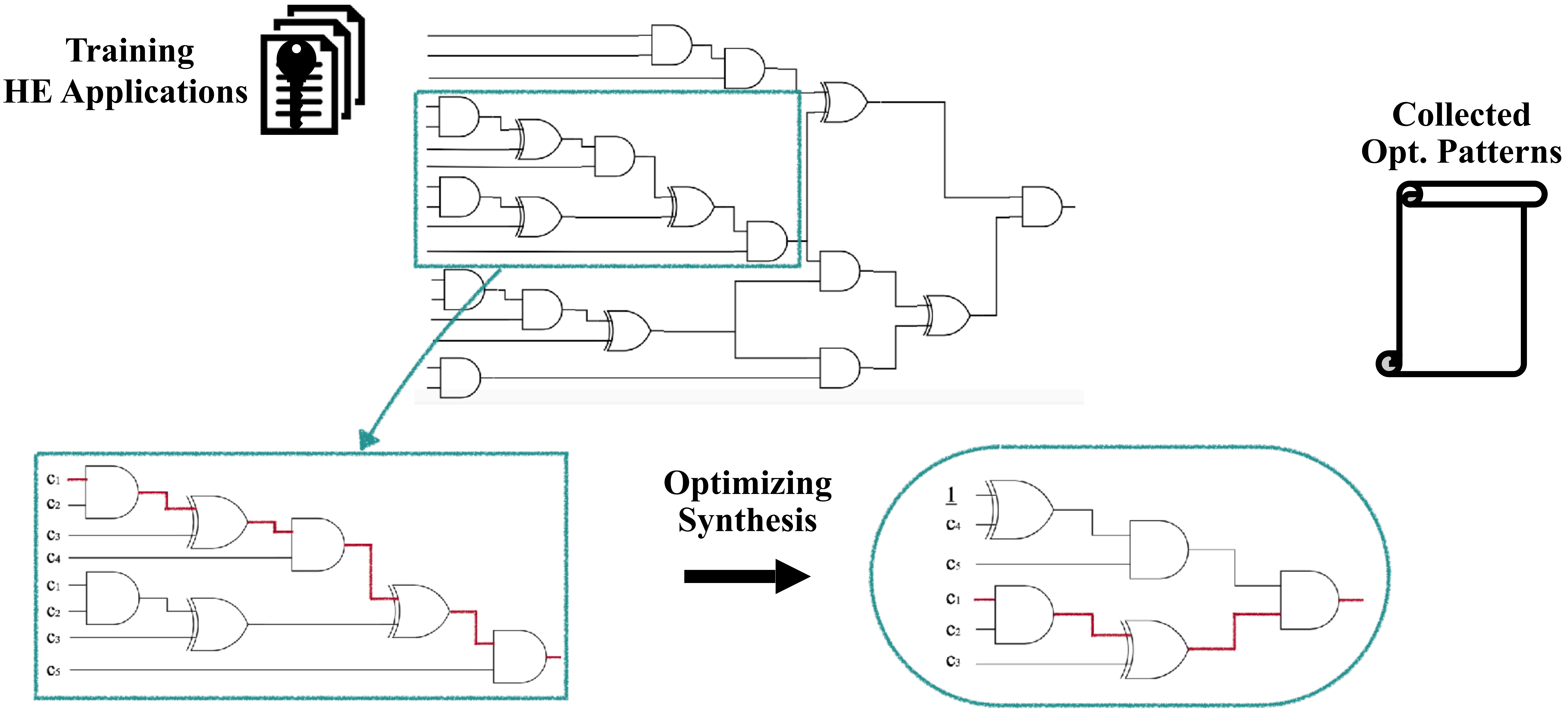




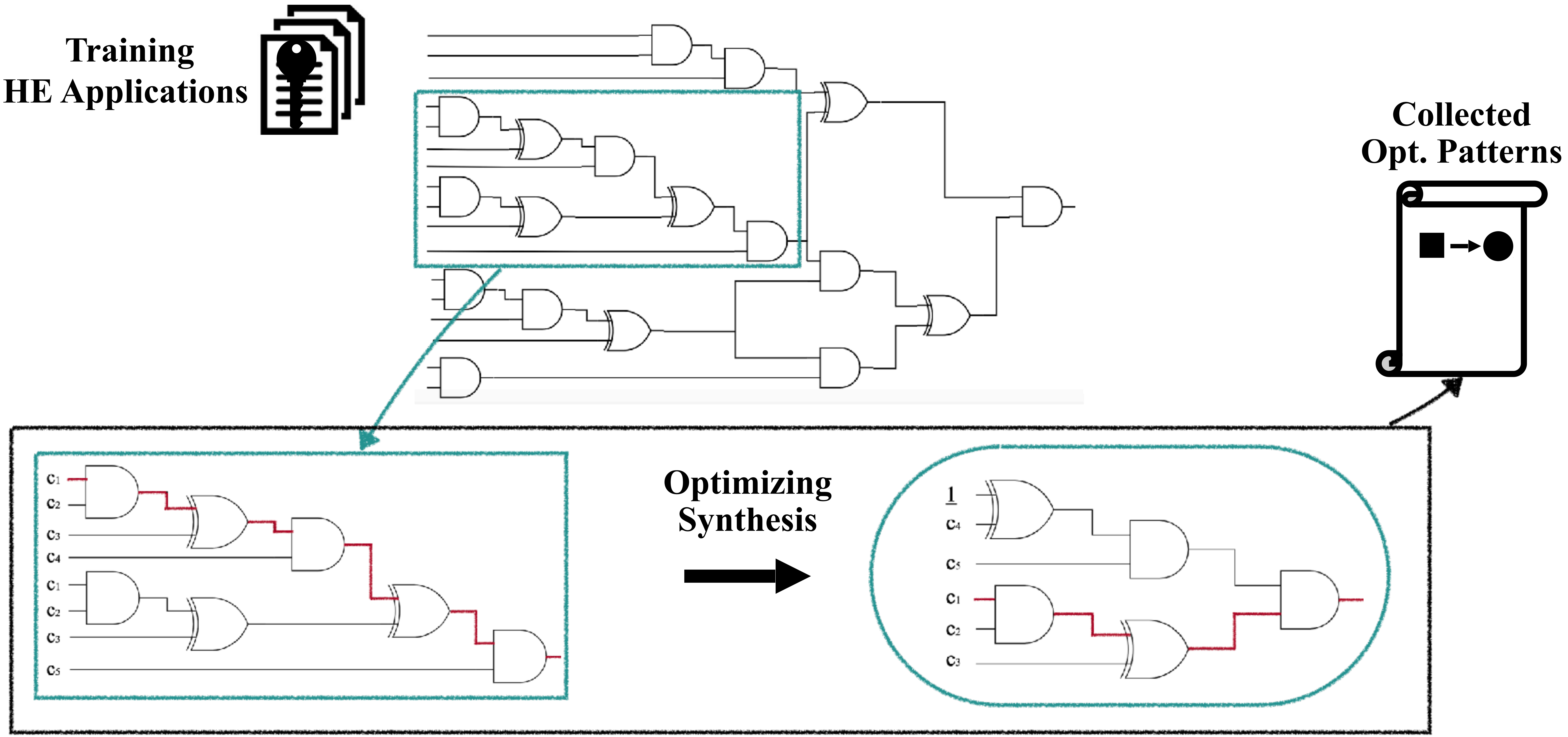
Collected
Opt. Patterns



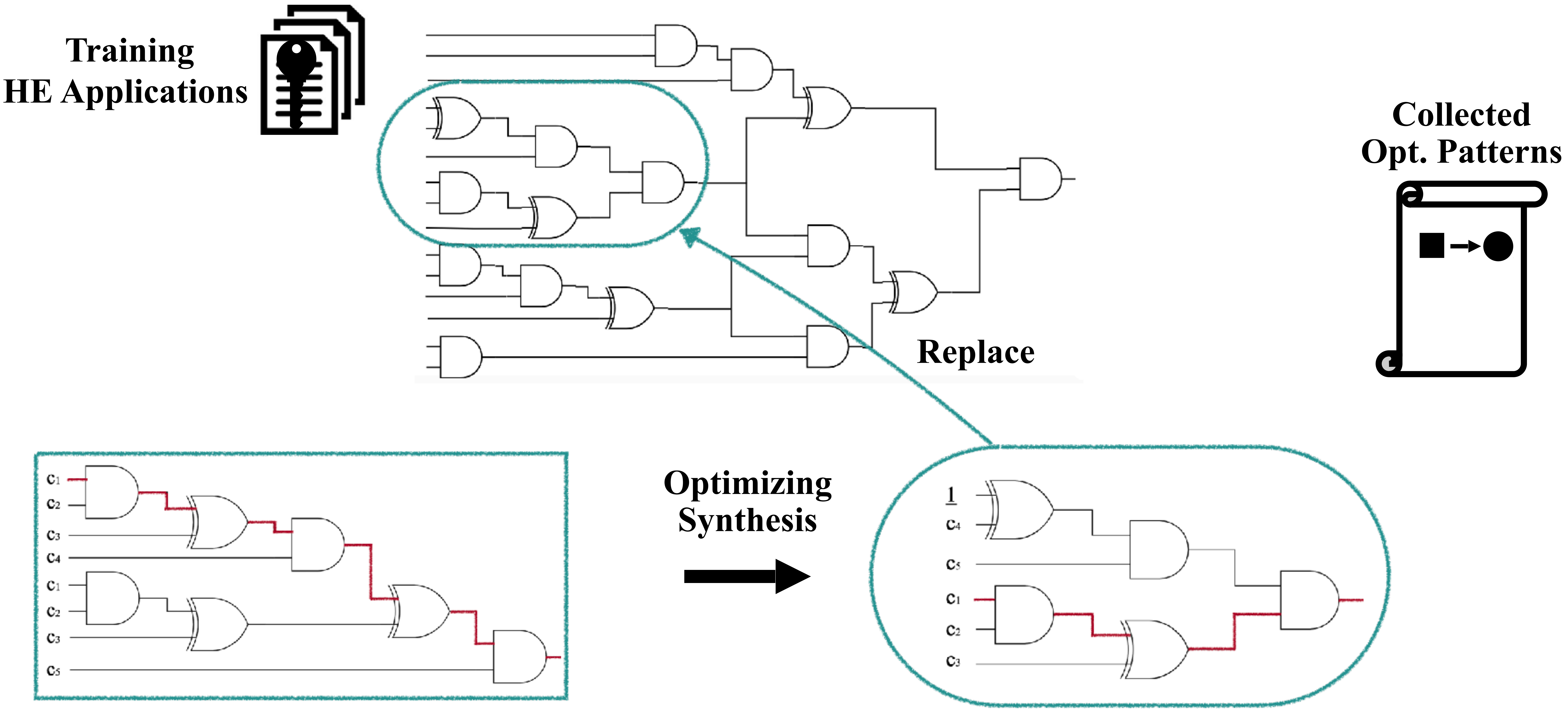
Offline Learning to Collect Opt. Patterns



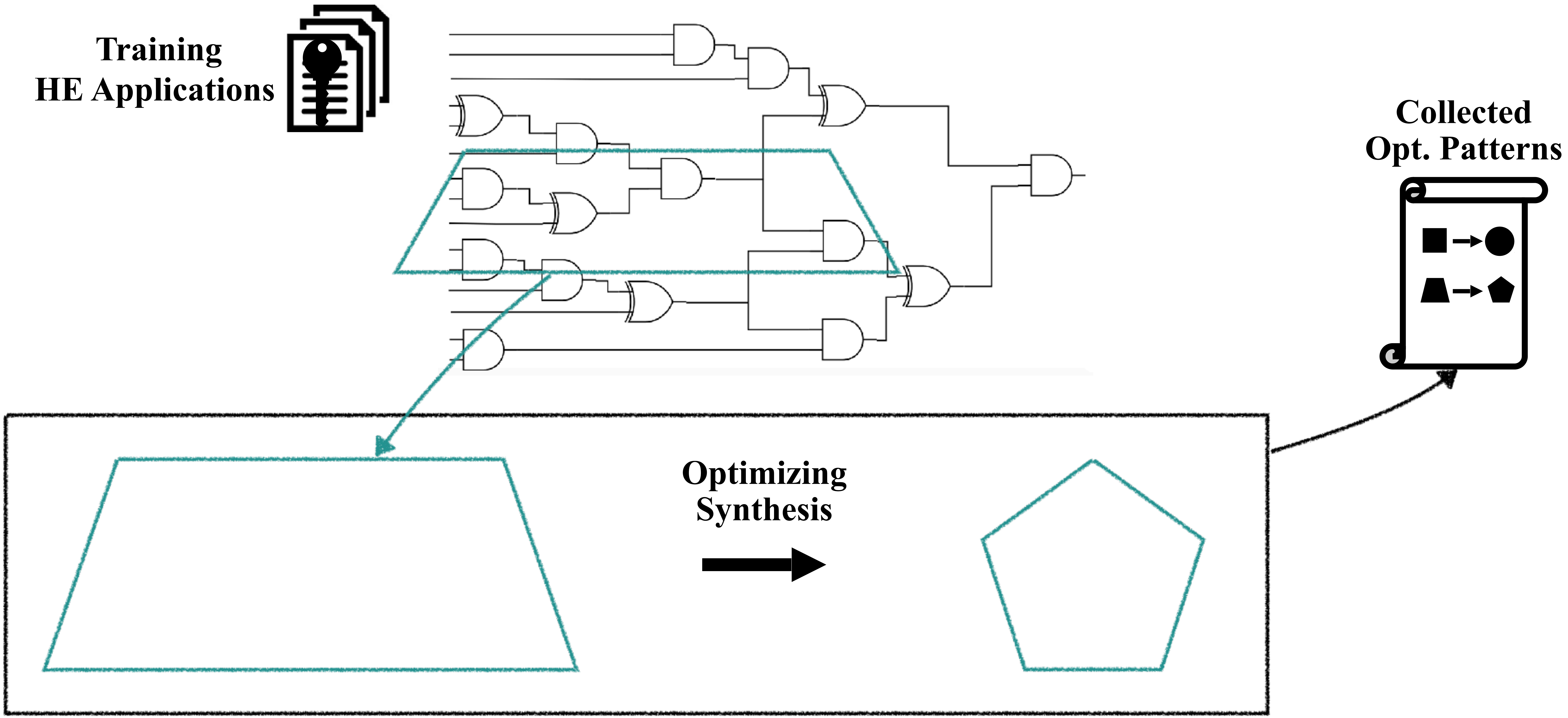
Offline Learning to Collect Opt. Patterns



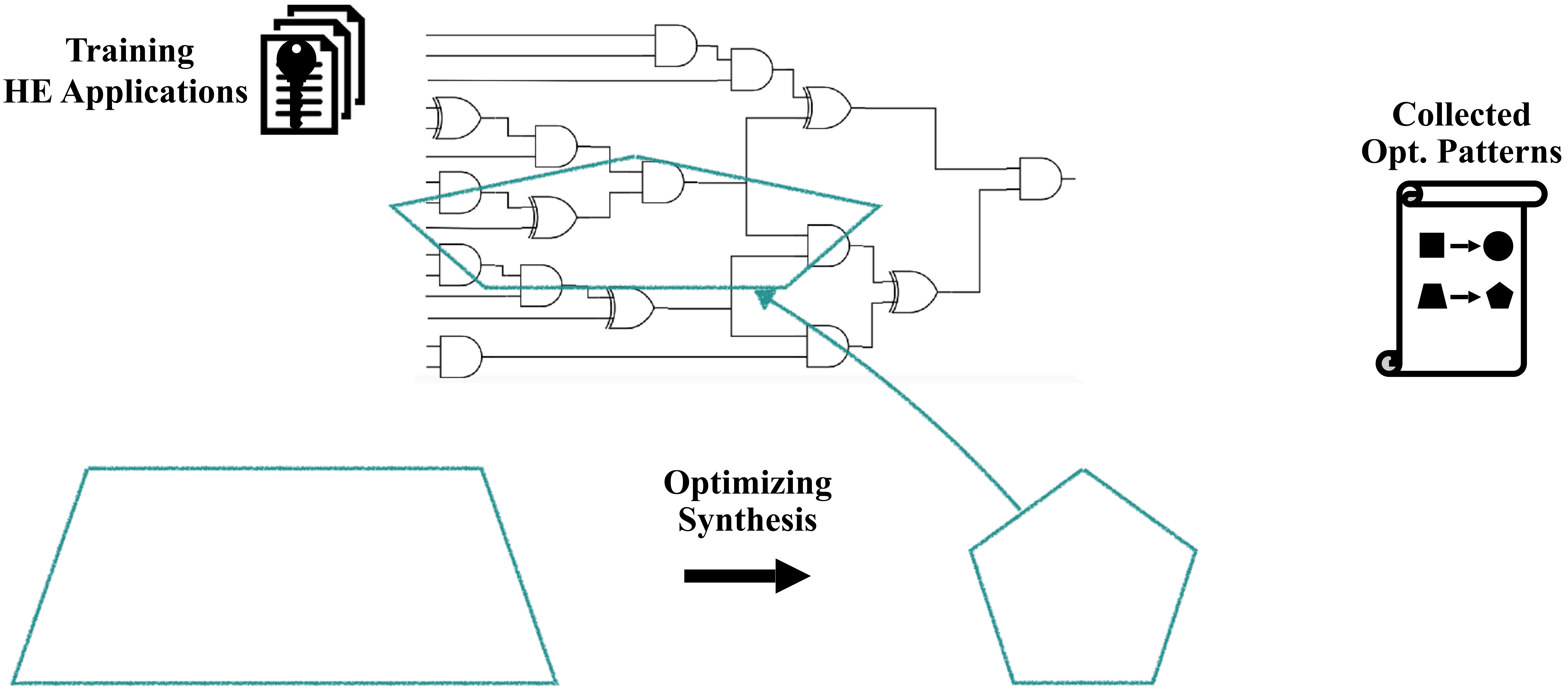
Offline Learning to Collect Opt. Patterns



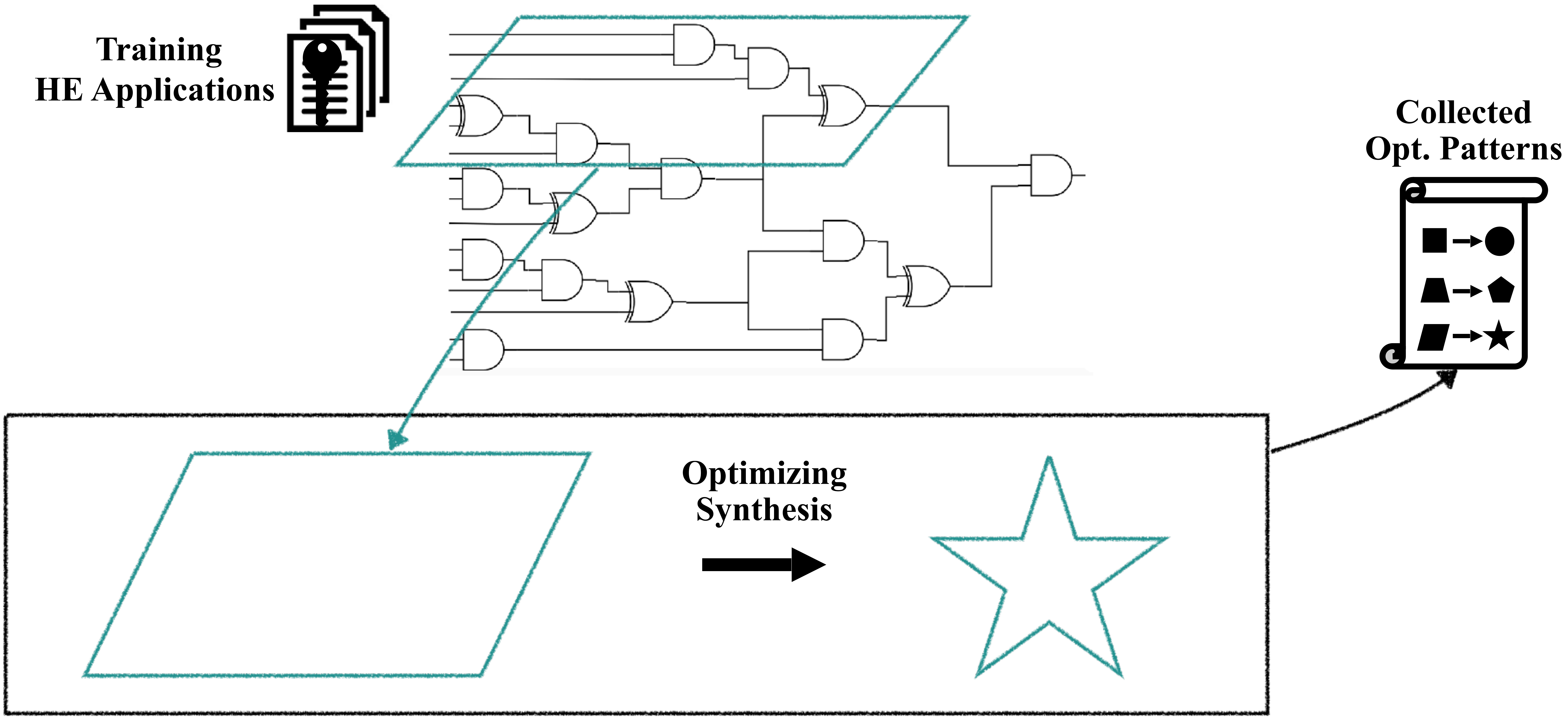
Offline Learning to Collect Opt. Patterns



Offline Learning to Collect Opt. Patterns

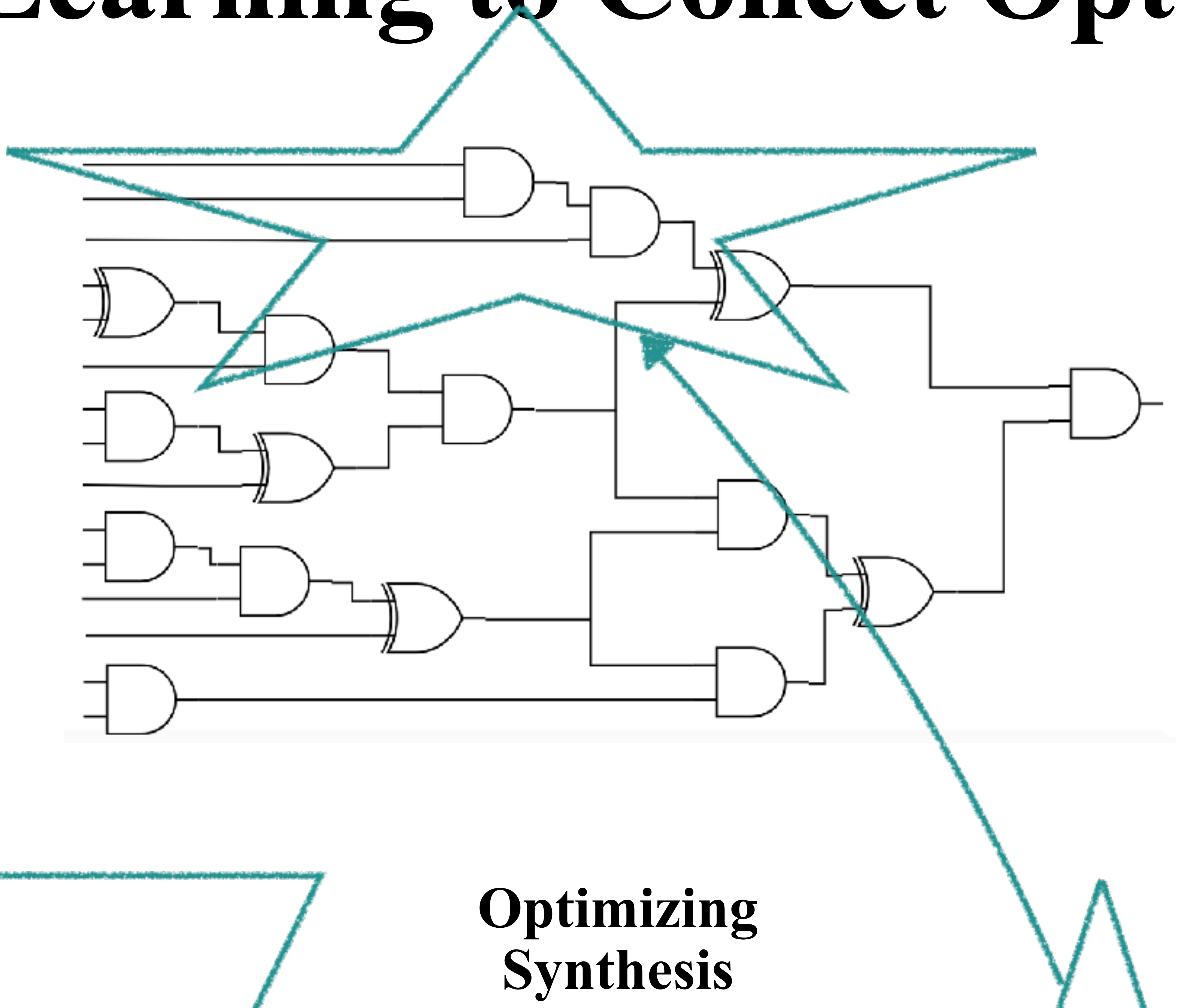


Offline Learning to Collect Opt. Patterns

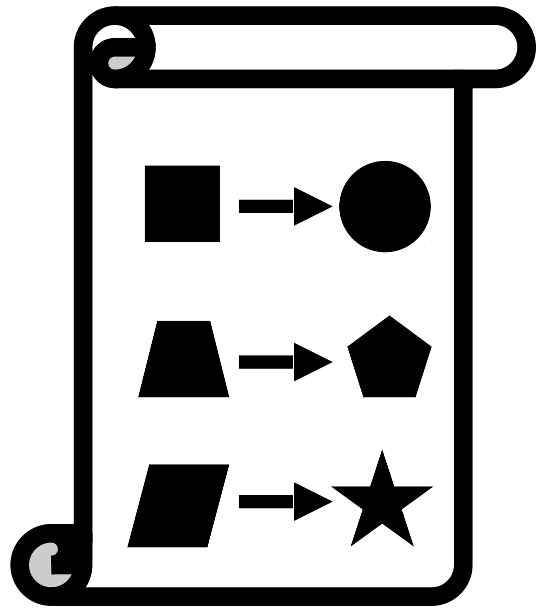


Offline Learning to Collect Opt. Patterns

Training
HE Applications



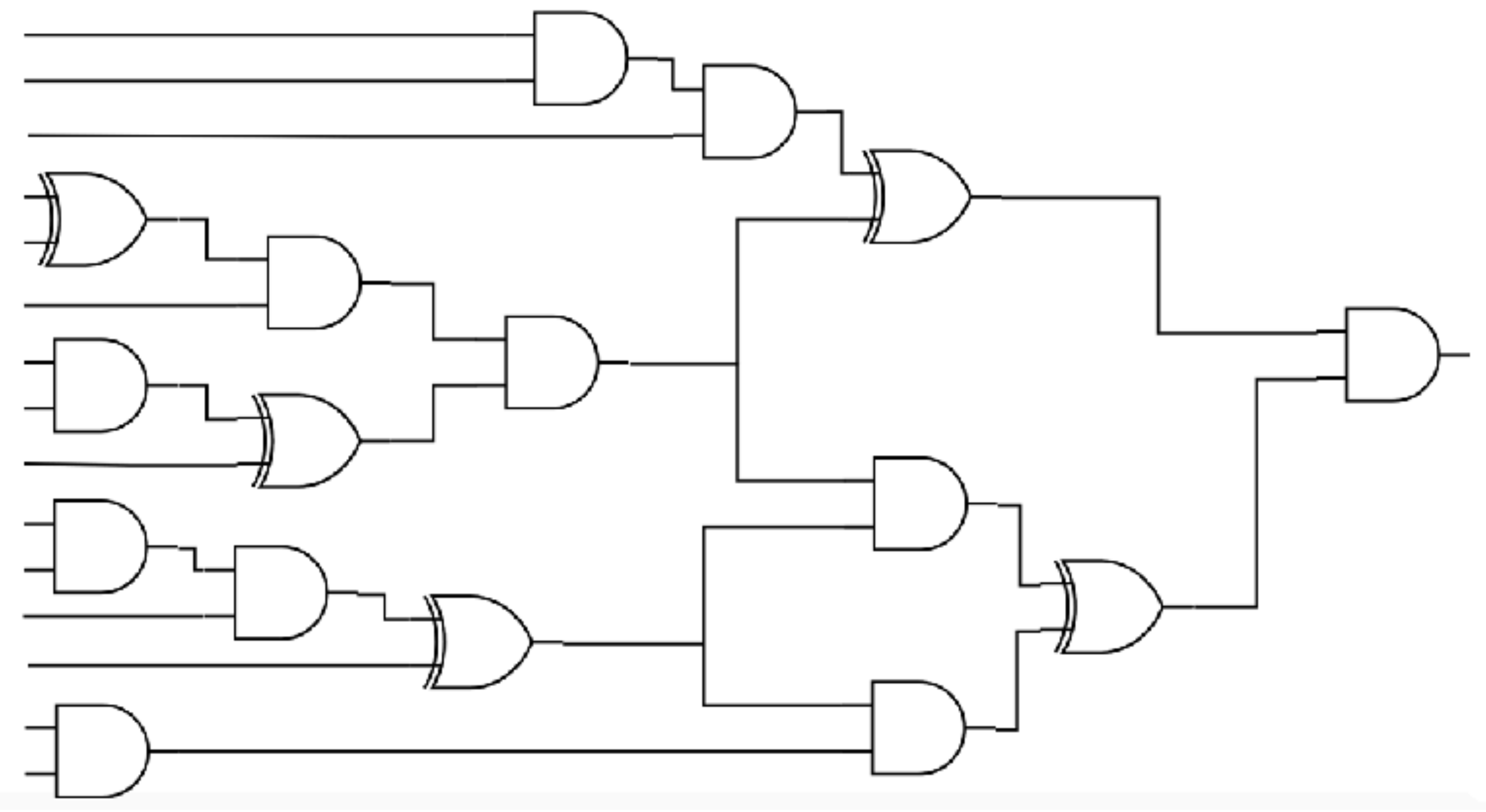
Collected
Opt. Patterns



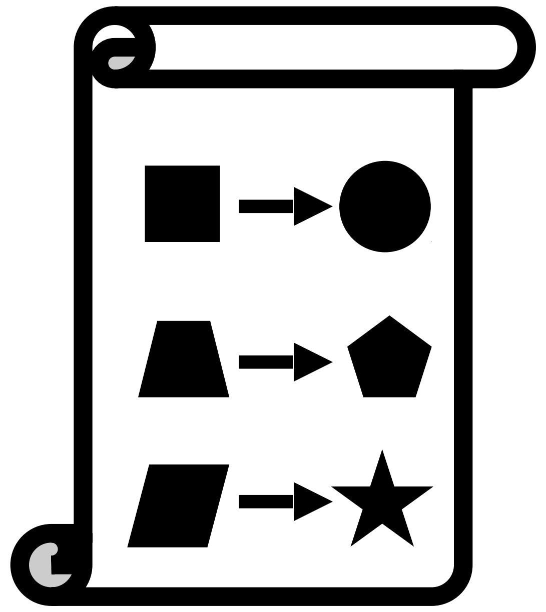
Optimizing
Synthesis

Offline Learning to Collect Opt. Patterns

Training
HE Applications

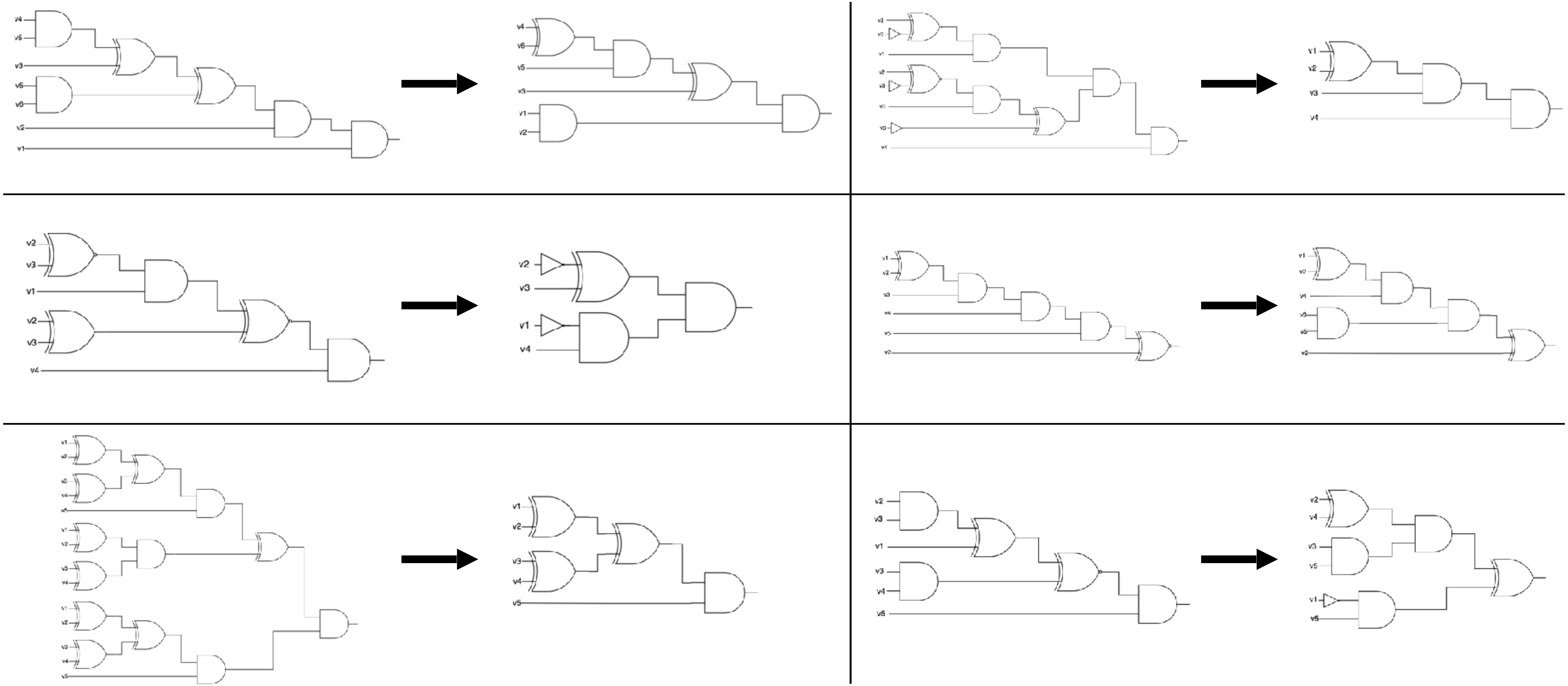


Collected
Opt. Patterns



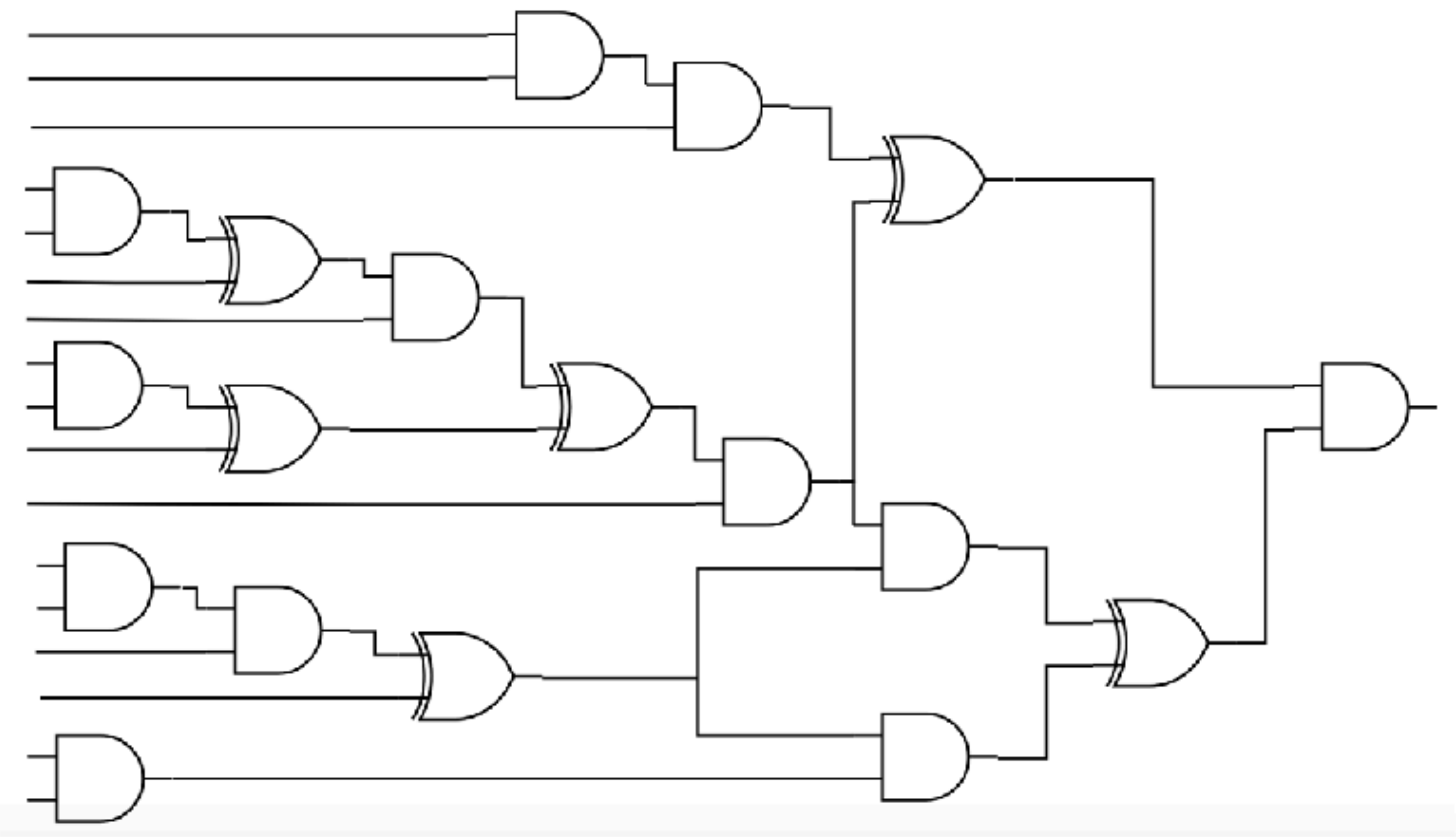
186 Opt.
patterns

Learned Optimization Patterns : examples

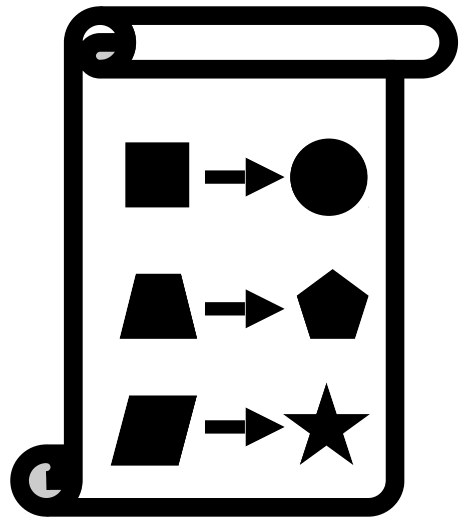


Online Rule-based Optimization

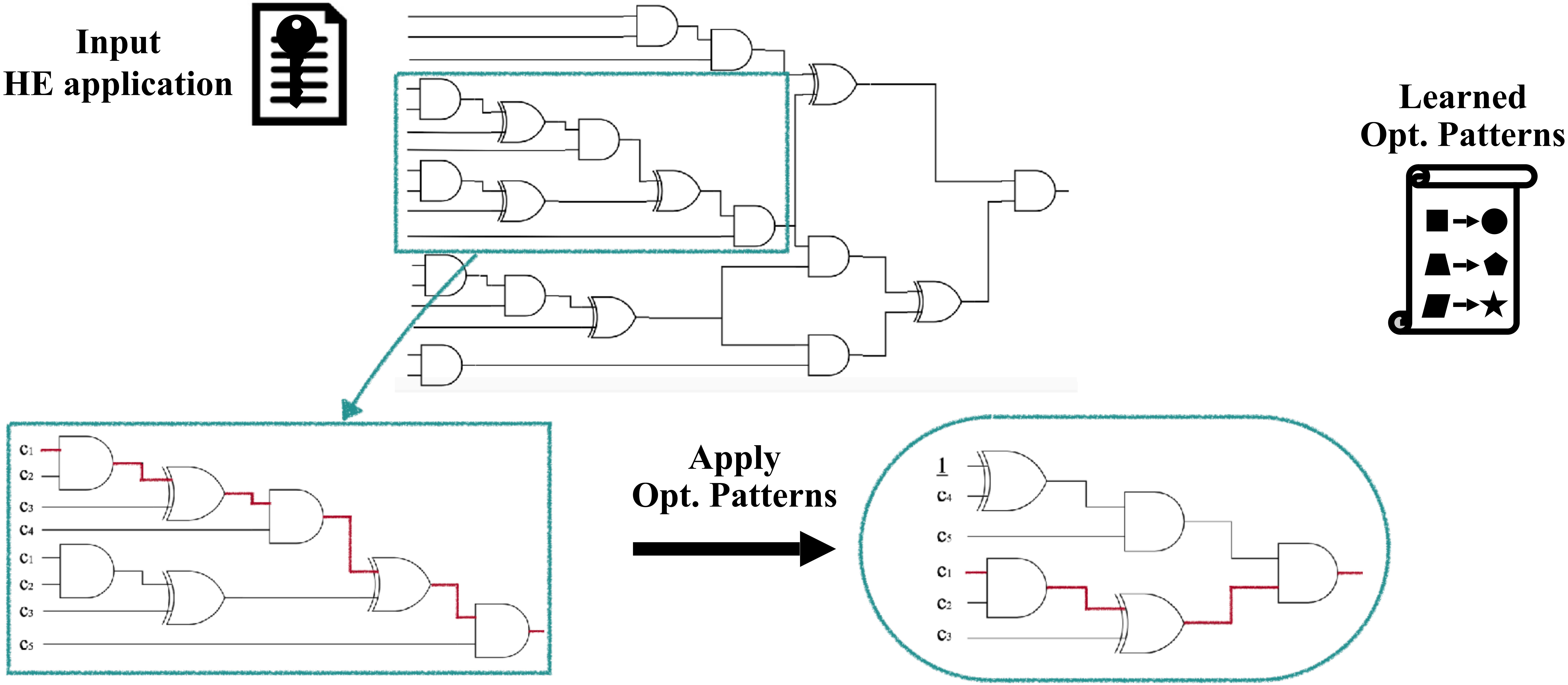
Input
HE application



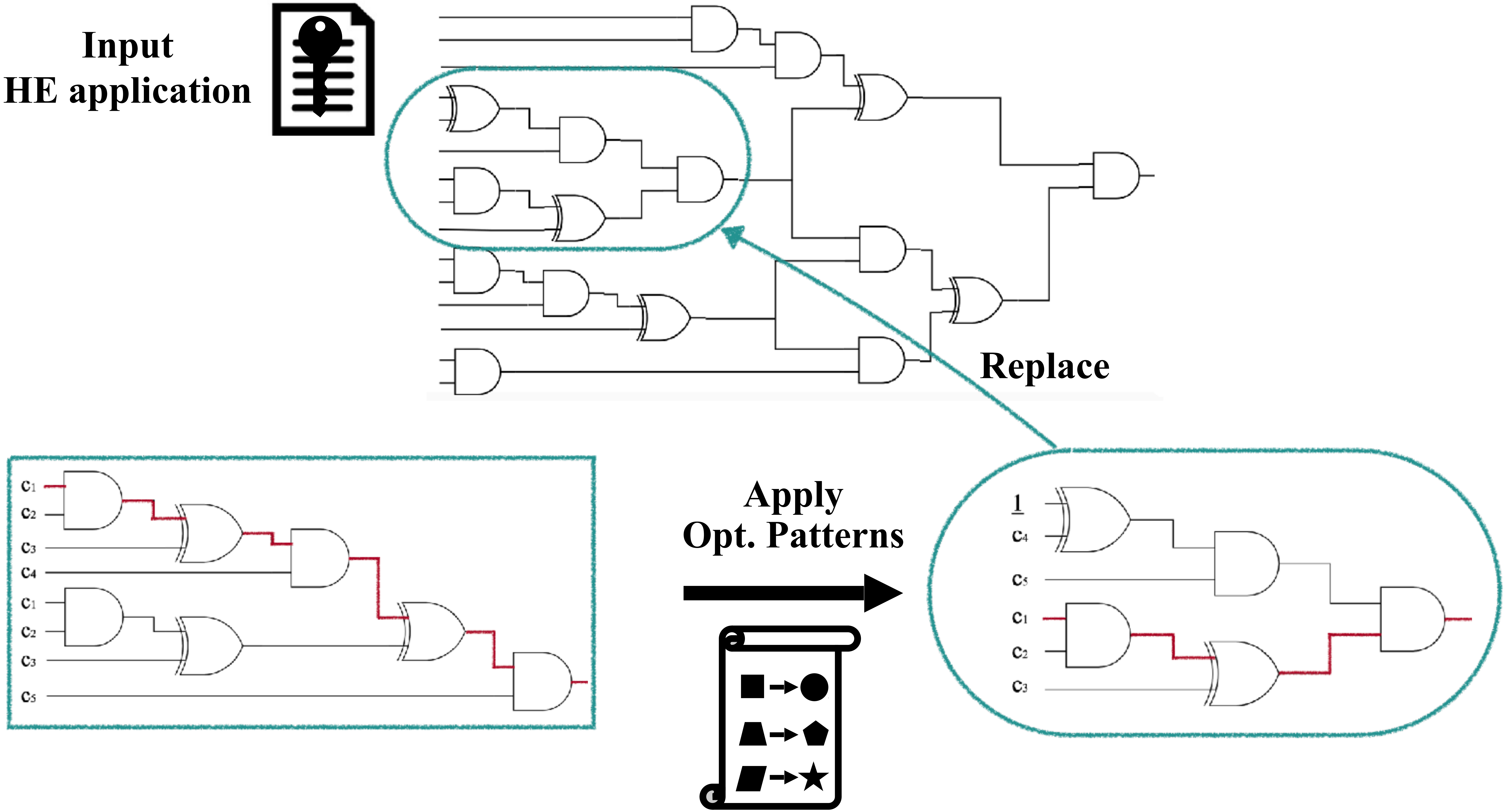
Learned
Opt. Patterns



Online Rule-based Optimization

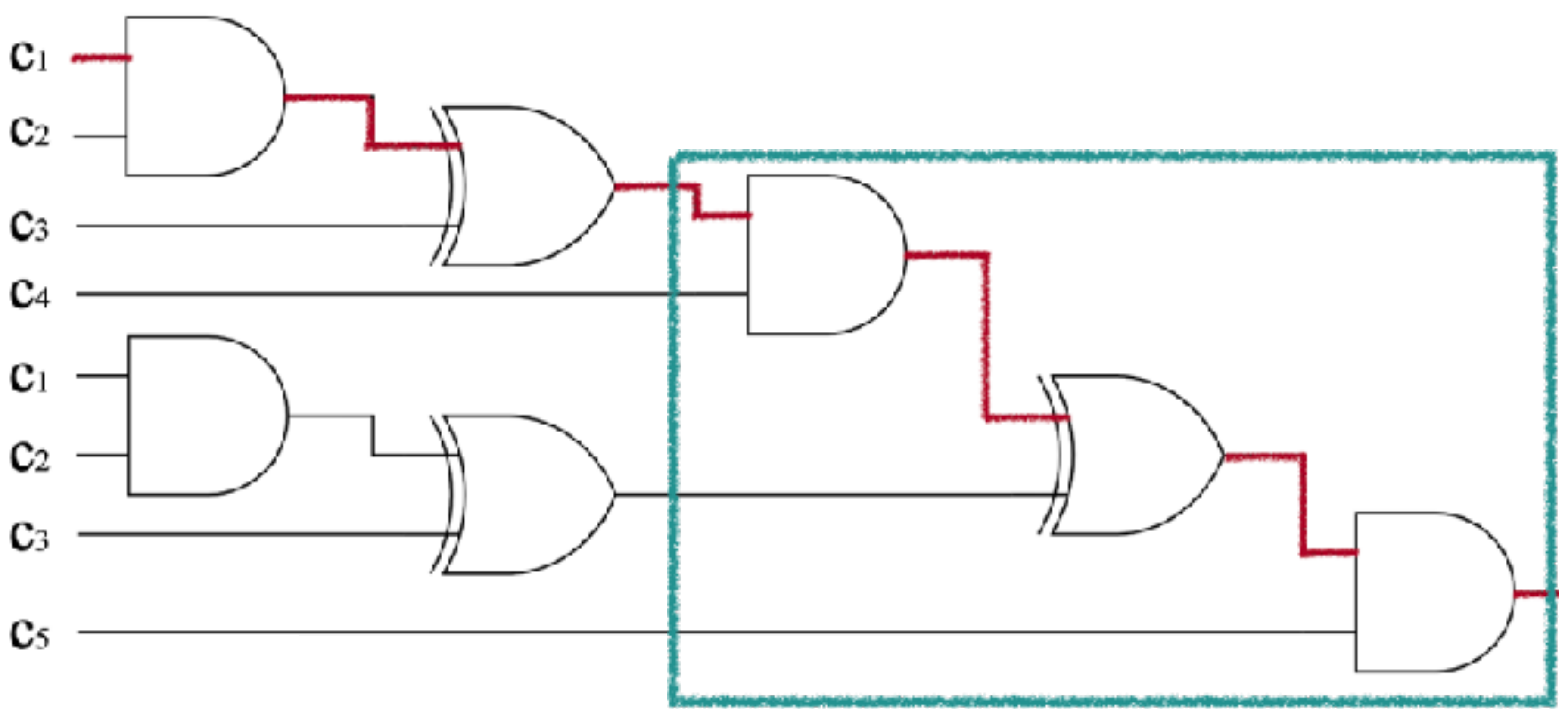


Online Rule-based Optimization

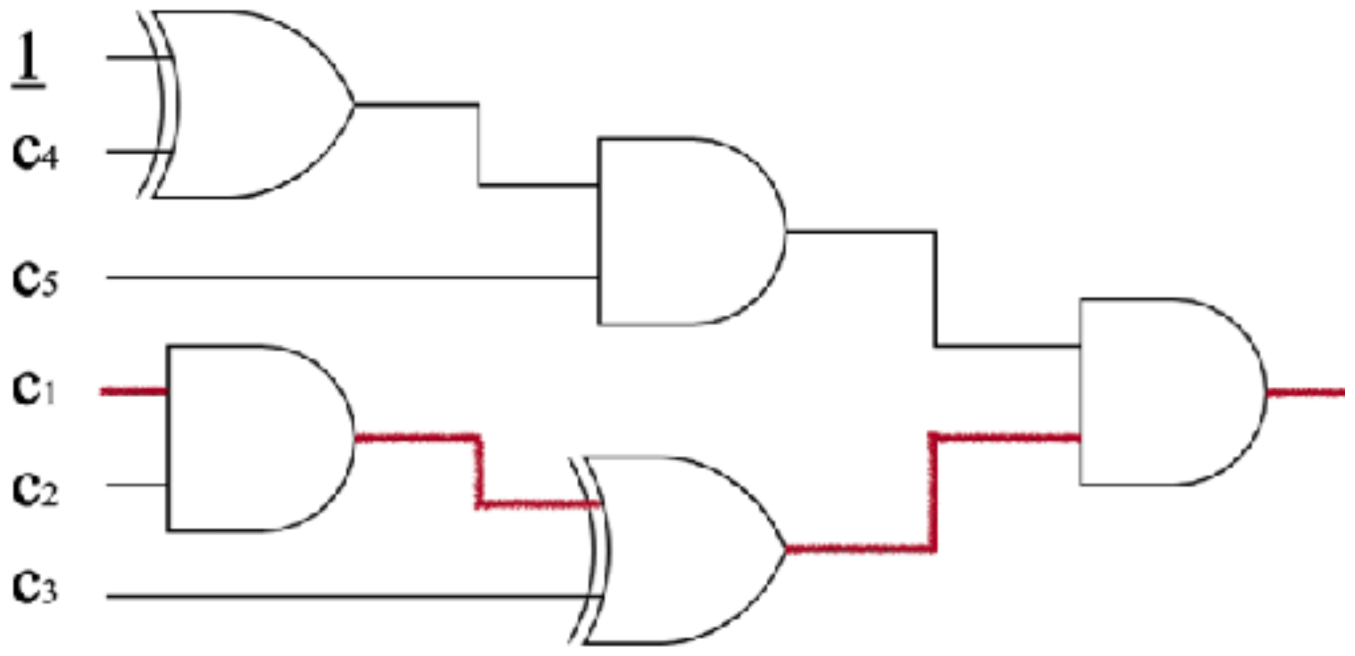


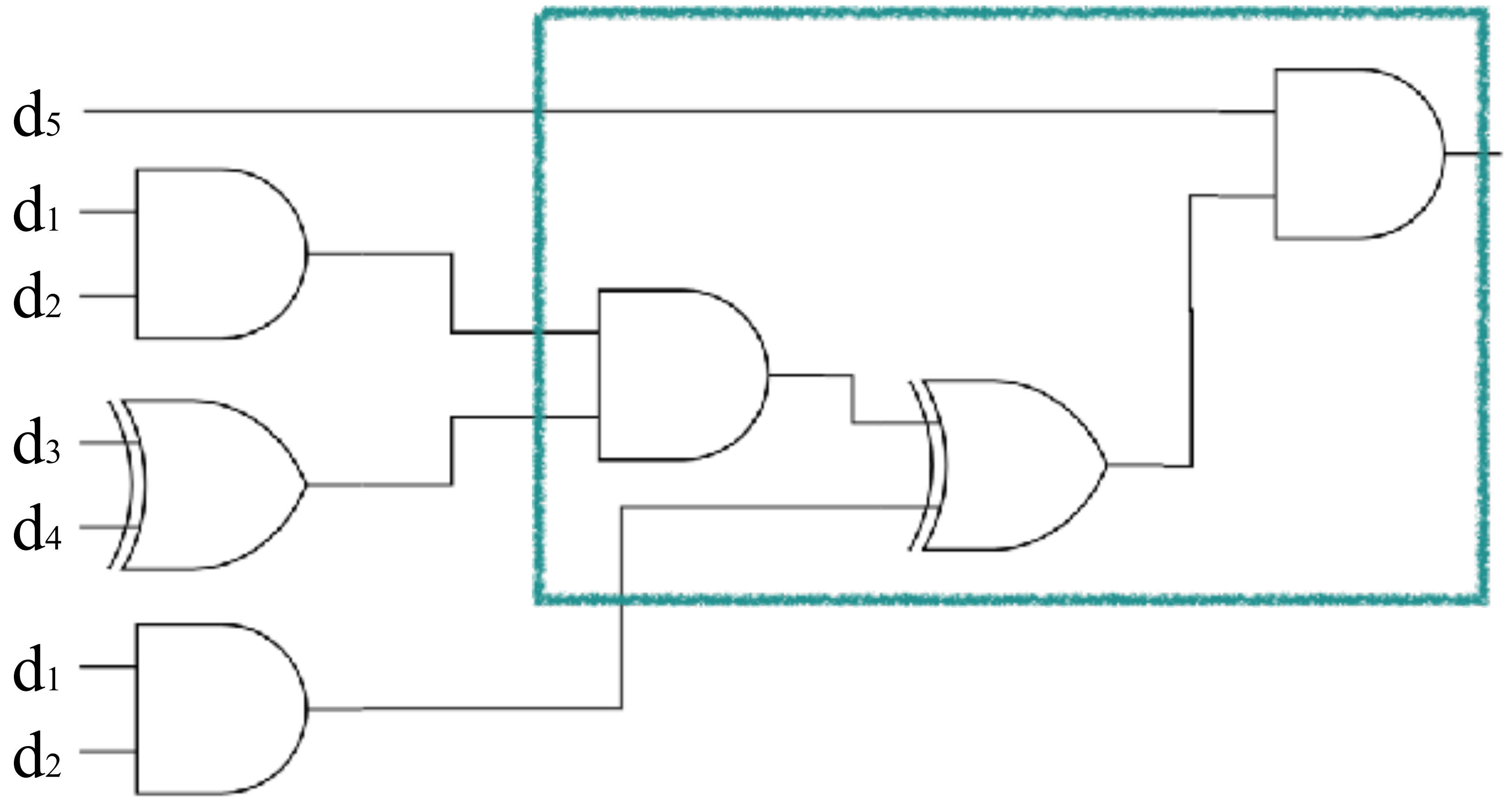
Applying Learned Optimization Patterns (1/2)

Syntactic Matching is Not Effective



Learned Opt. Patterns



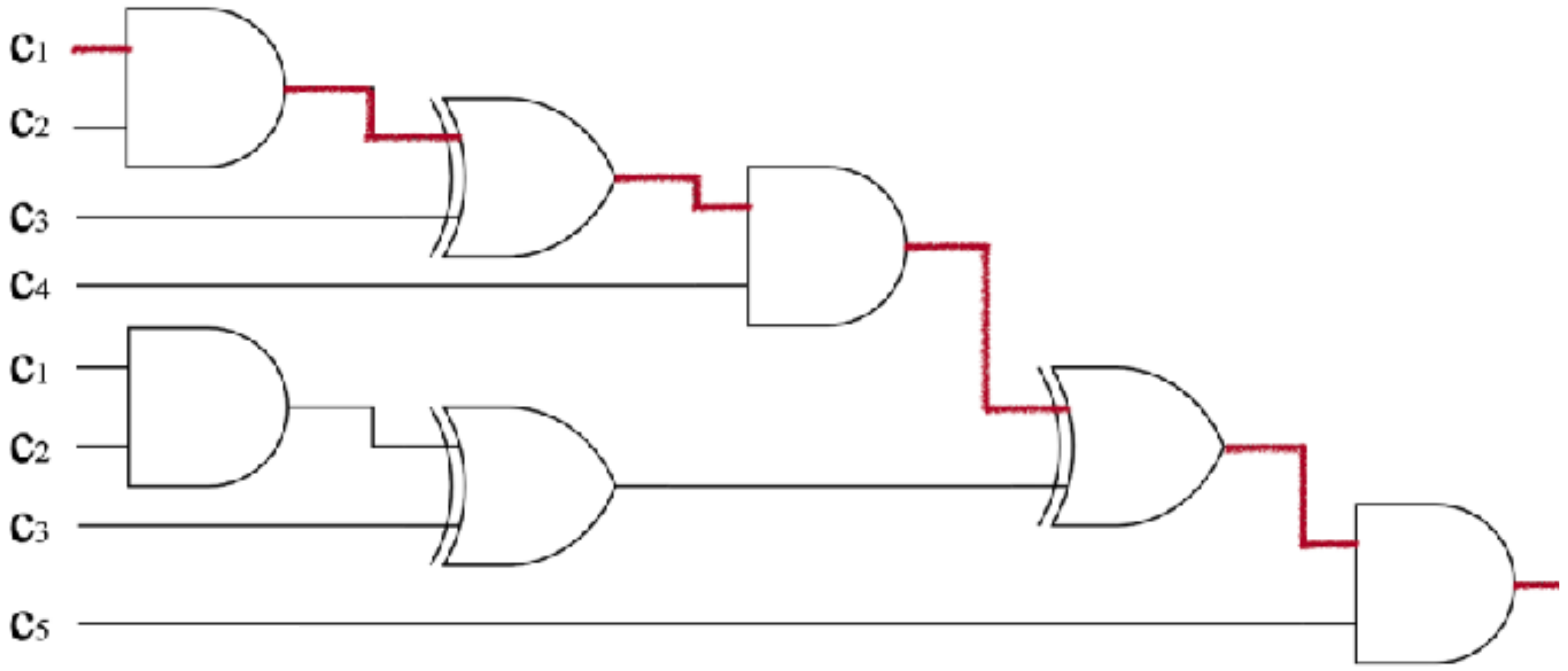


New Input Circuit Optimization

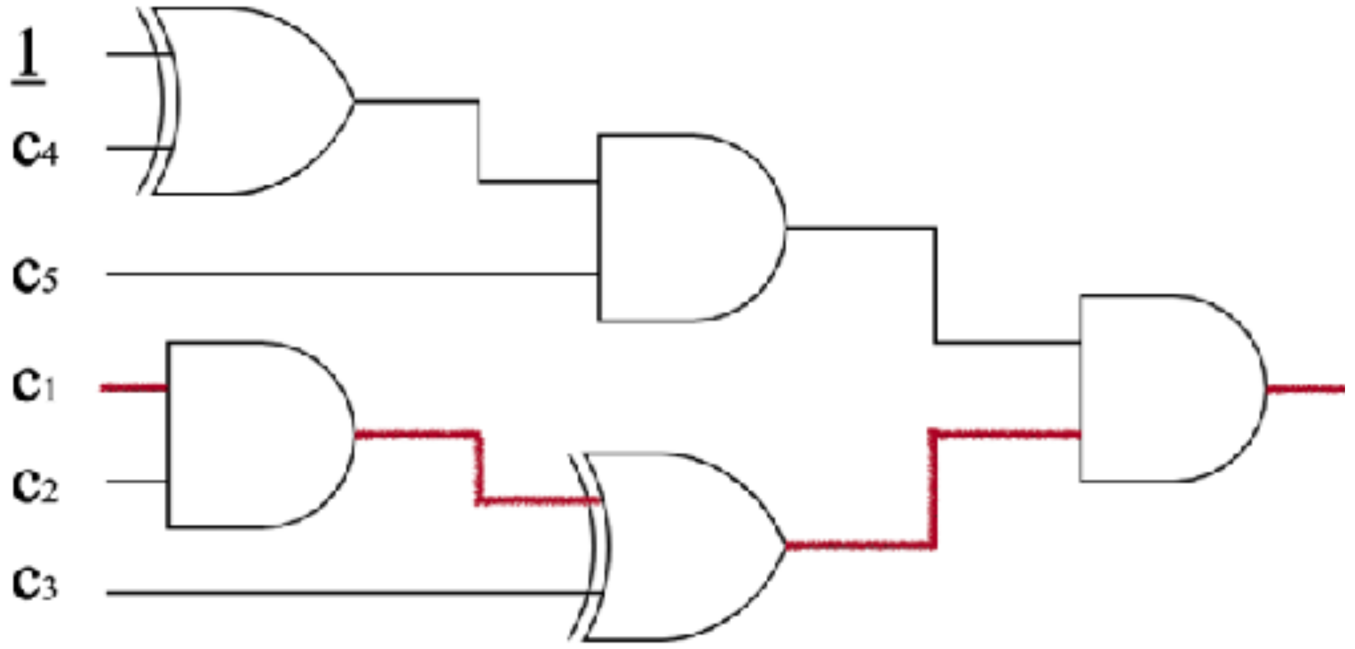
?

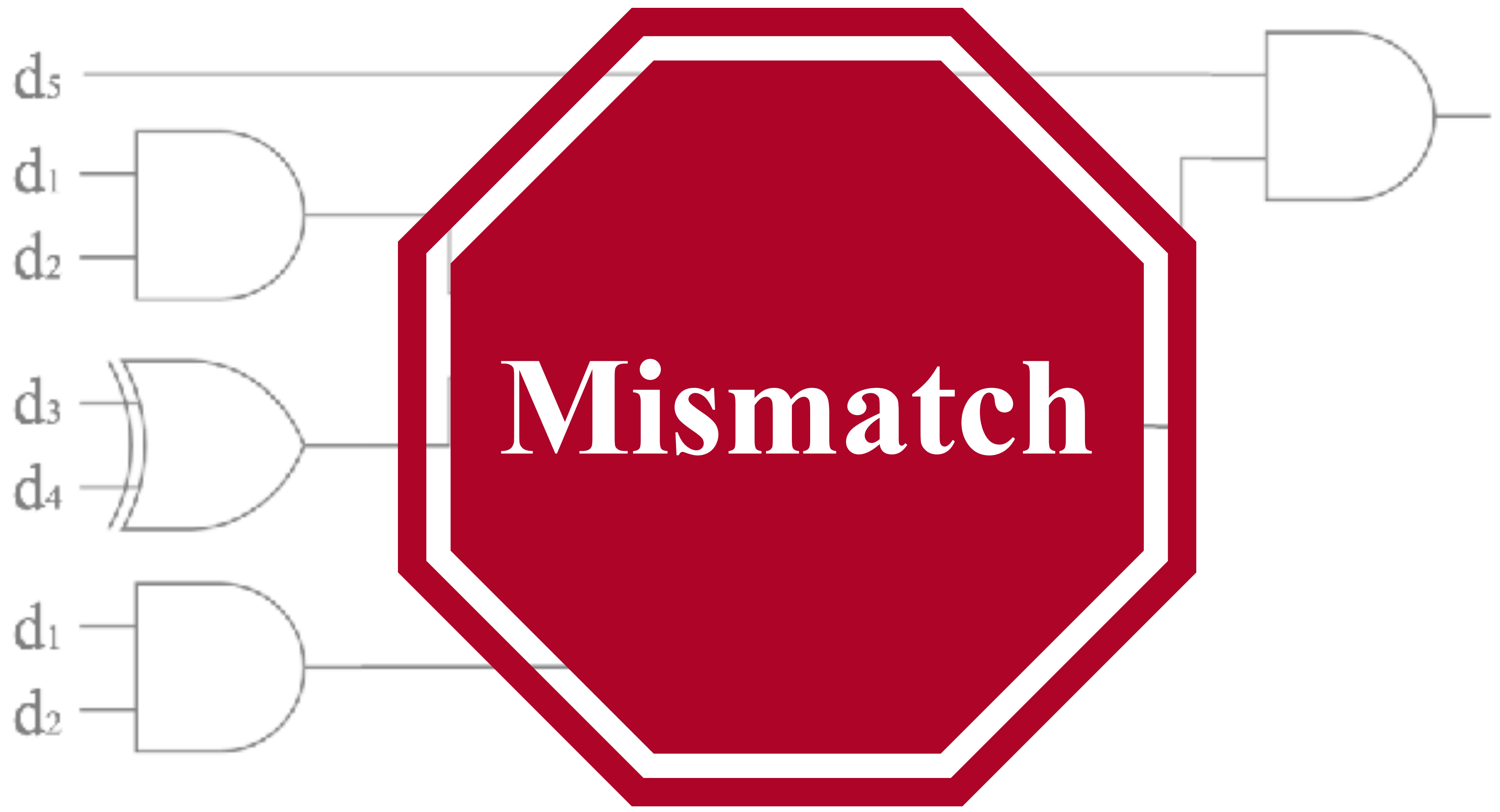
Applying Learned Optimization Patterns (1/2)

Syntactic Matching is Not Effective



Learned Opt. Patterns



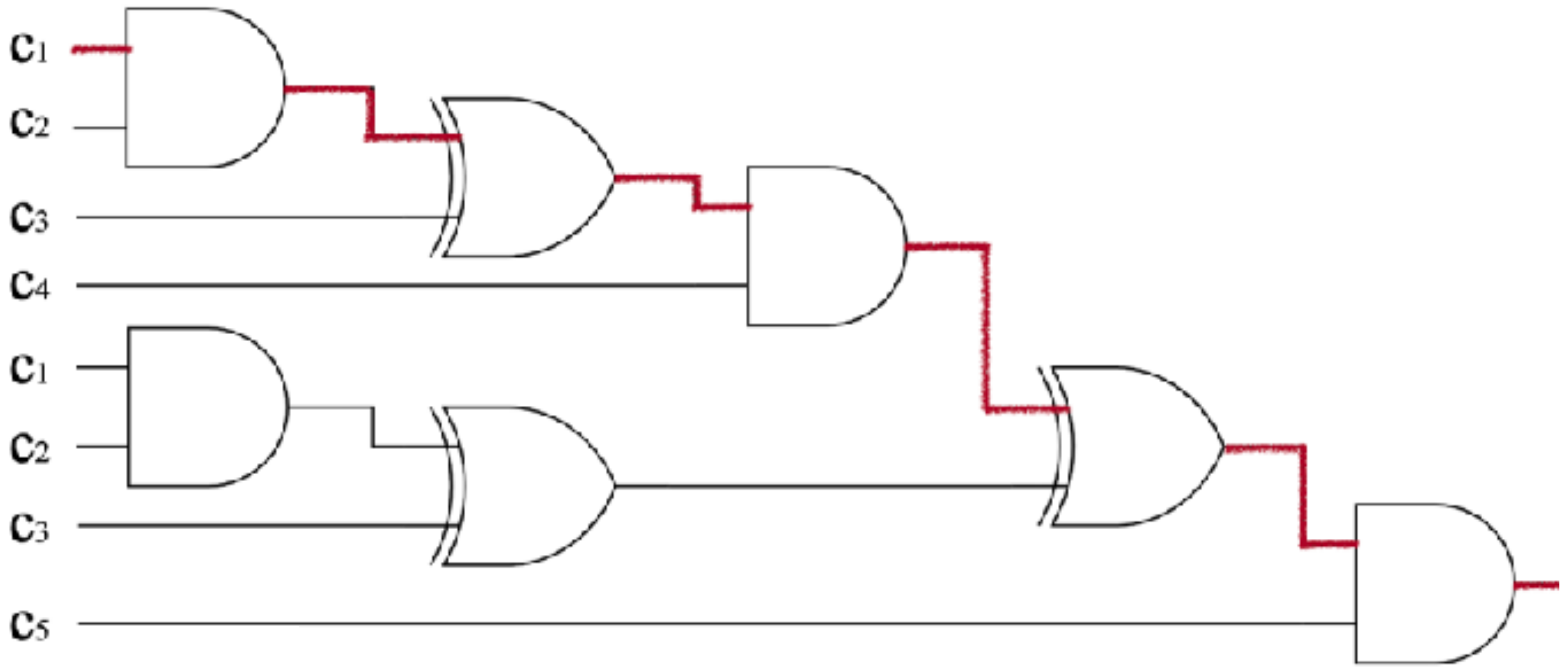


New Input Circuit Optimization

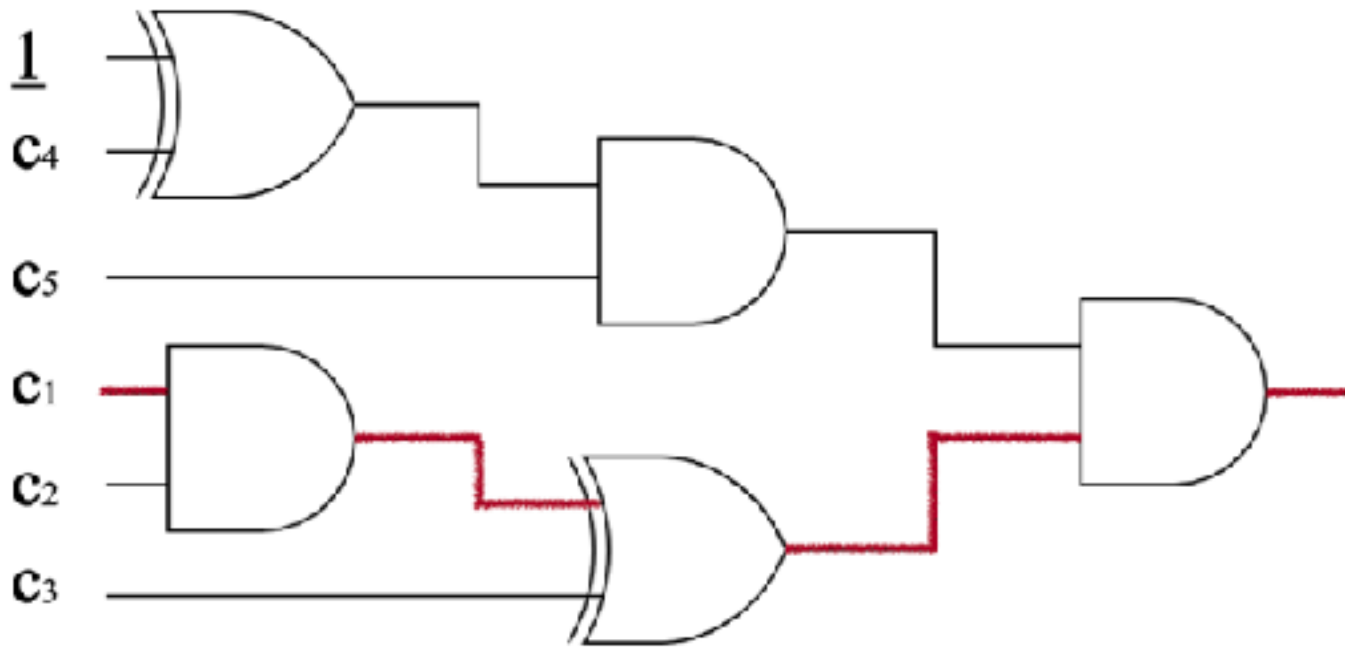
?

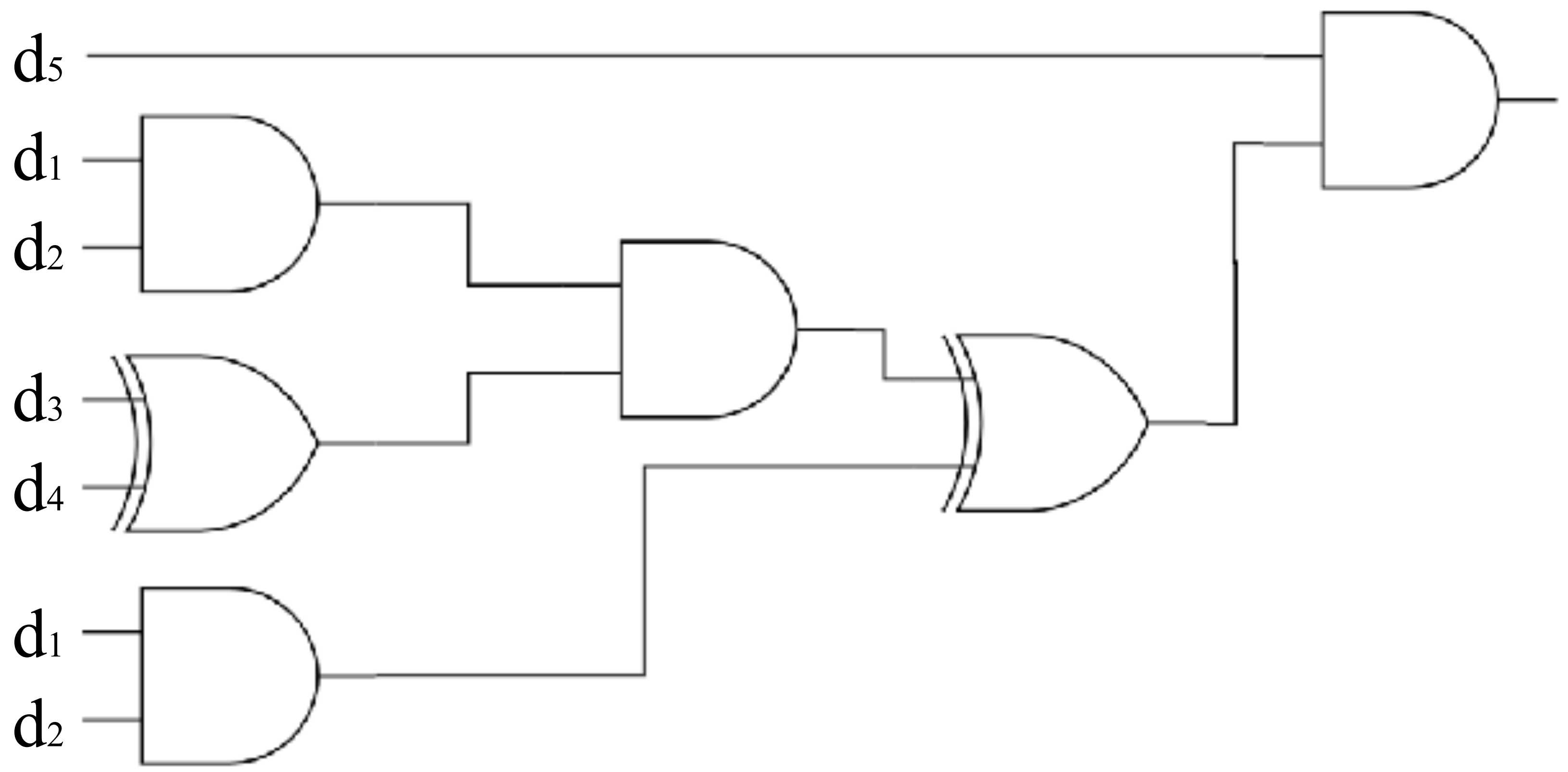
Applying Learned Optimization Patterns (2/2)

Normalization + Equational Matching



Learned Opt. Patterns



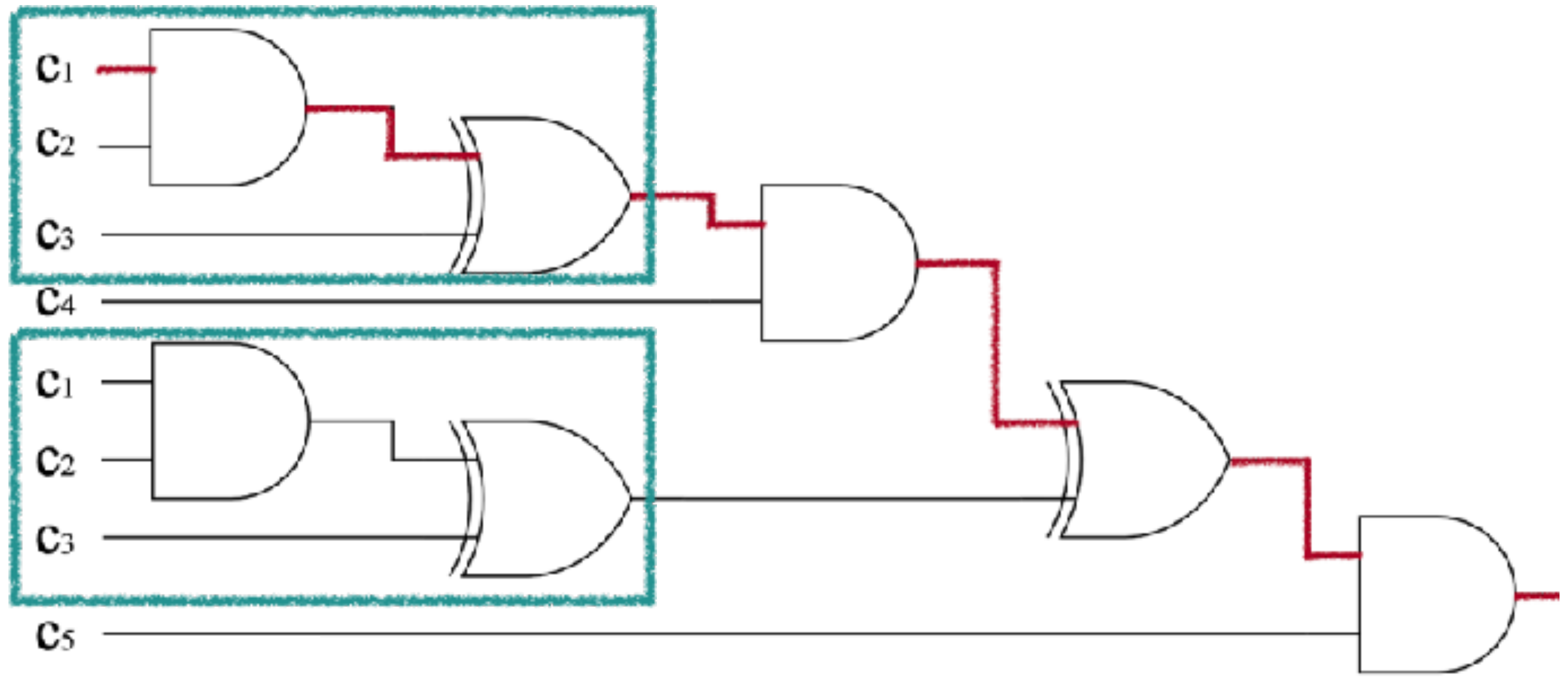


New Input Circuit Optimization

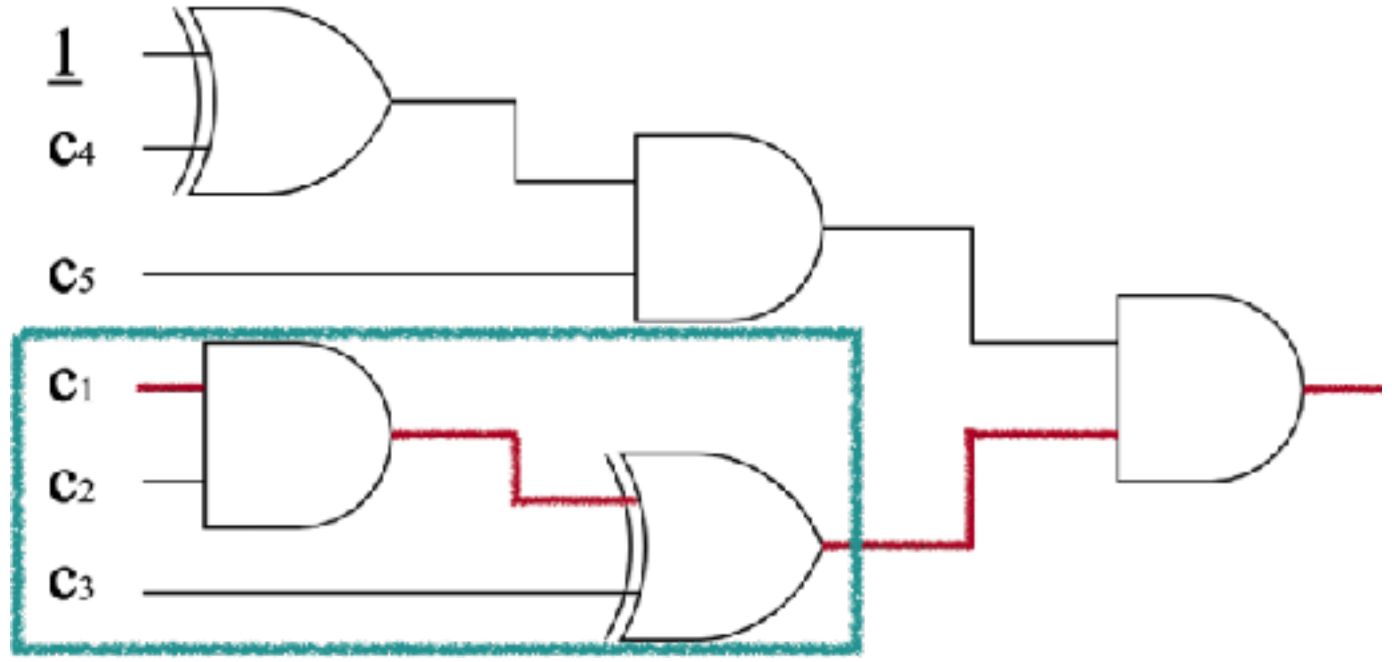
?

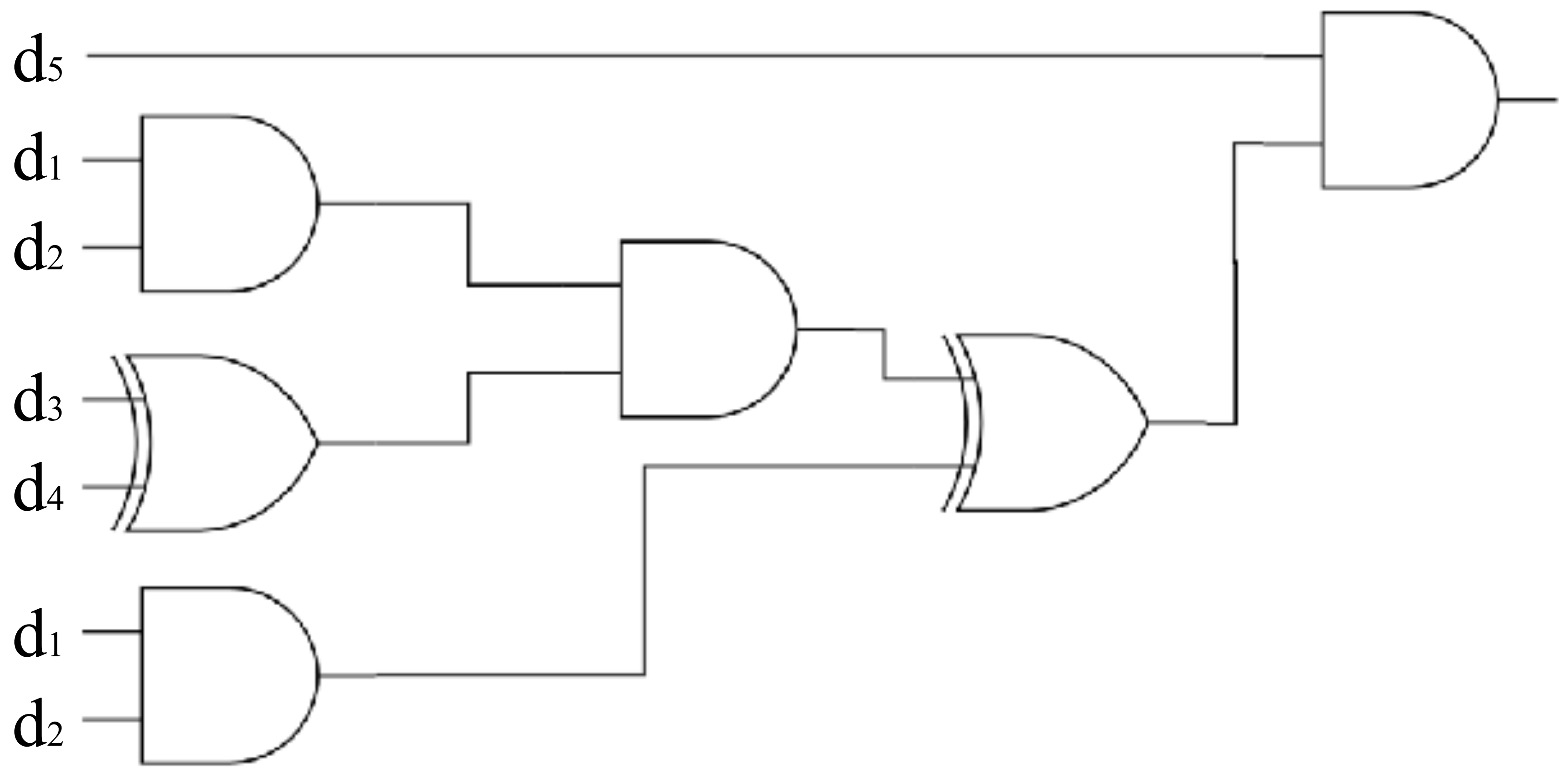
Applying Learned Optimization Patterns (2/2)

Normalization + Equational Matching



Learned Opt. Patterns



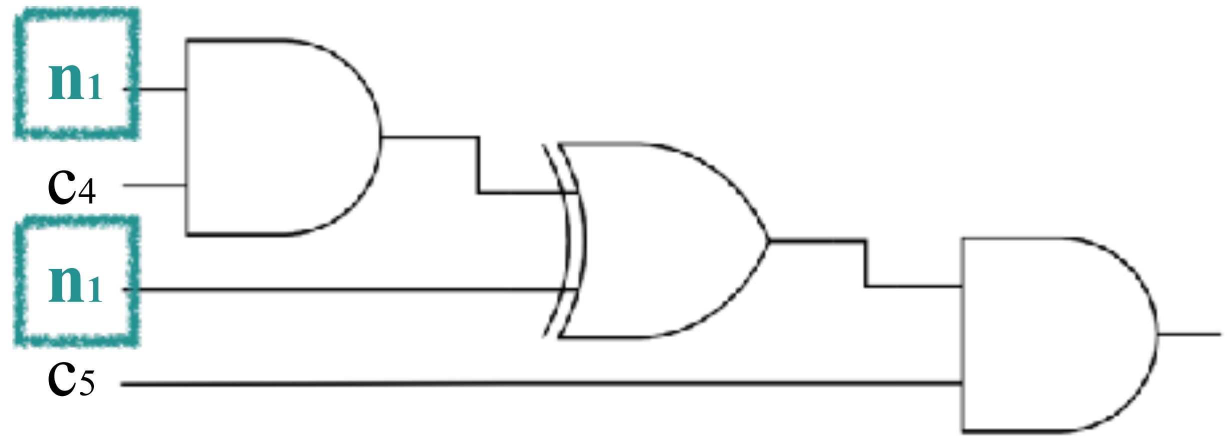


New Input Circuit Optimization

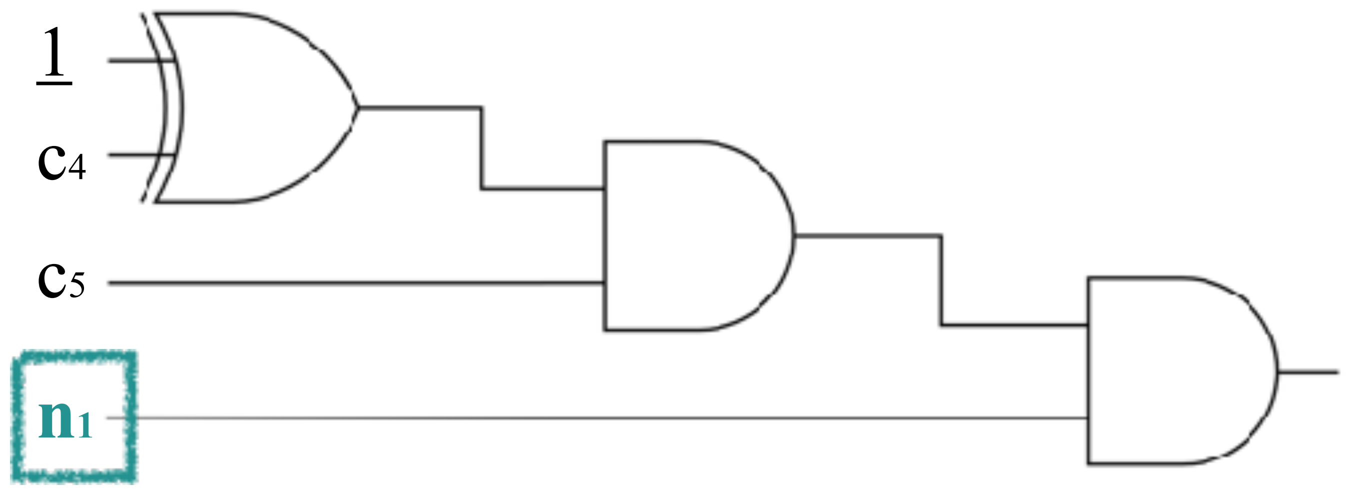
?

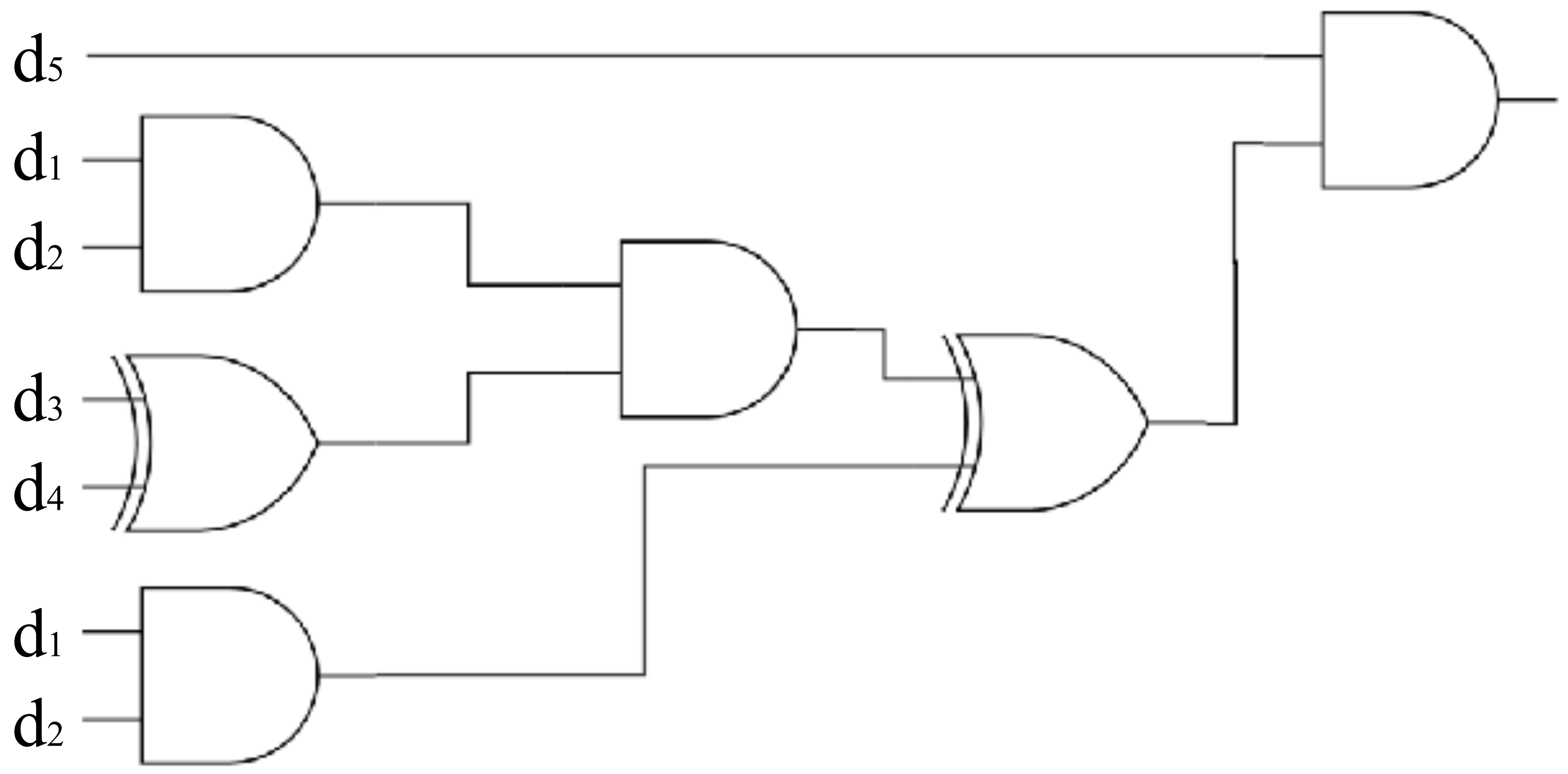
Applying Learned Optimization Patterns (2/2)

Normalization + Equational Matching



Normalized Opt. Patterns





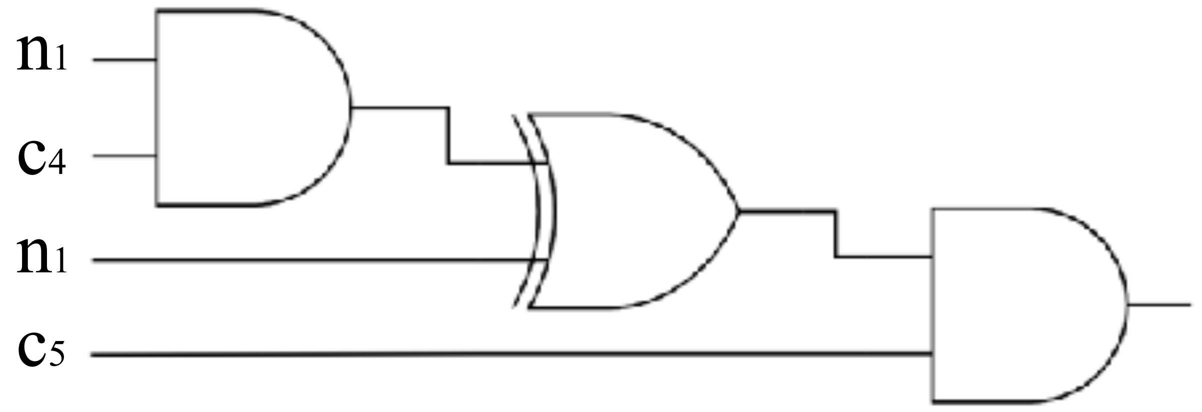
New Input Circuit Optimization

?

Applying Learned Optimization Patterns (2/2)

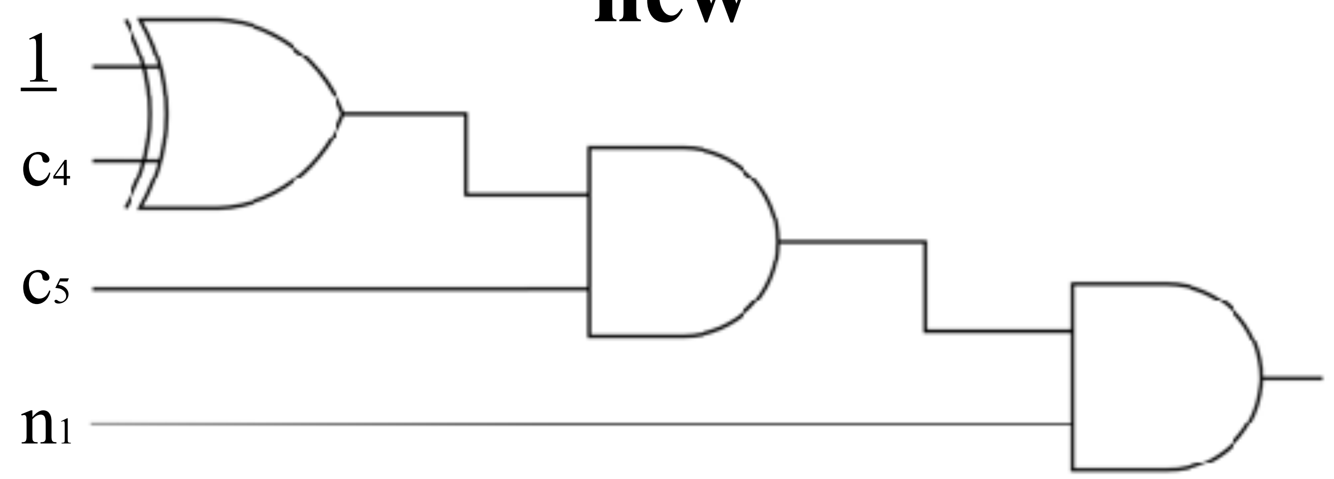
Normalization + Equational Matching

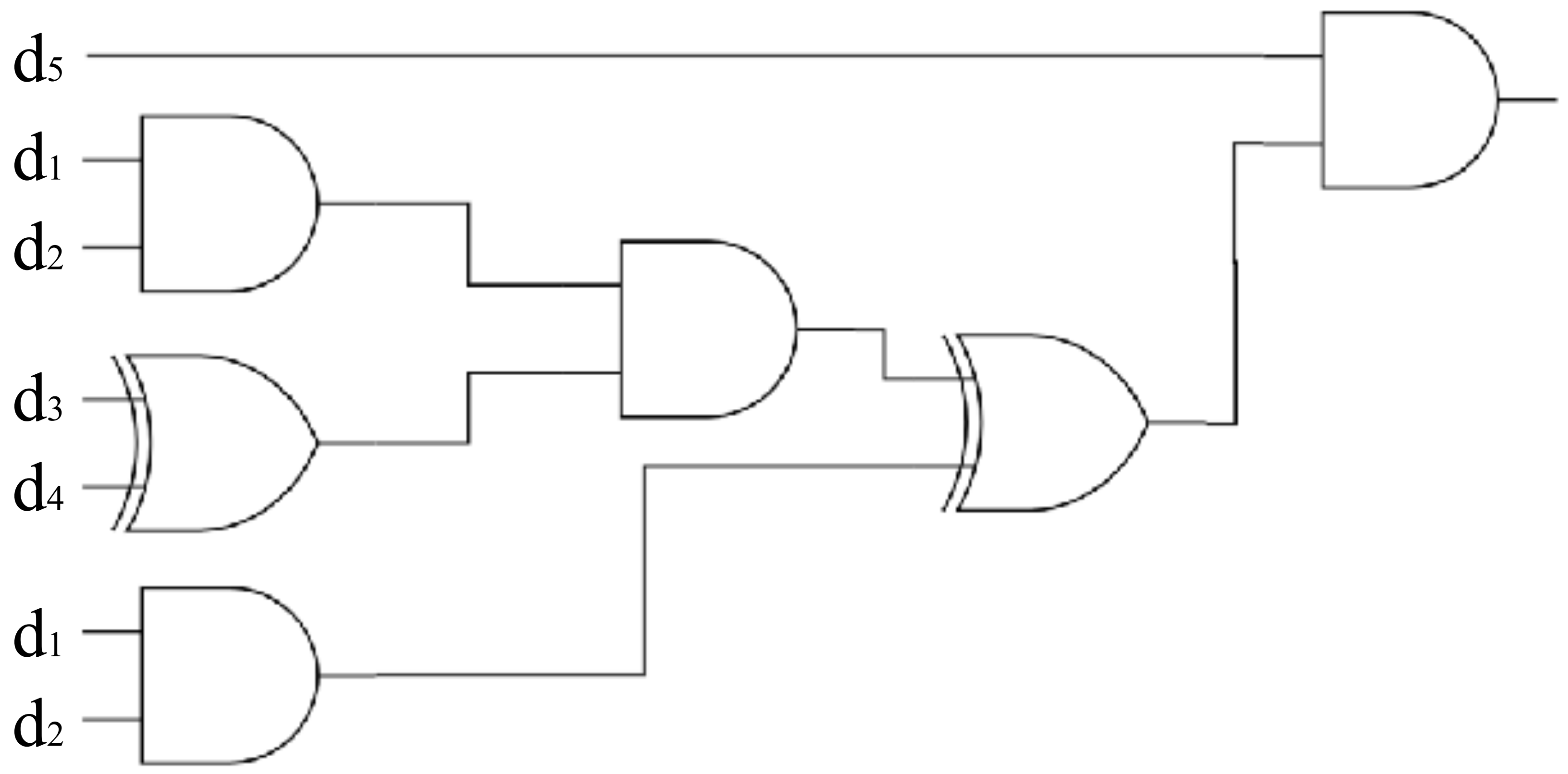
old



Normalized Opt. Patterns

new





target

New Input Circuit Optimization

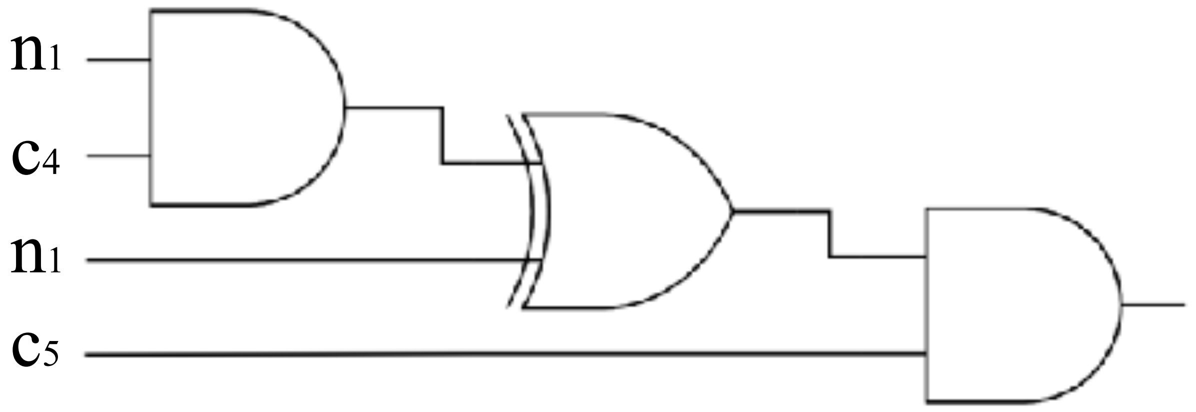
?

target'

Applying Learned Optimization Patterns (2/2)

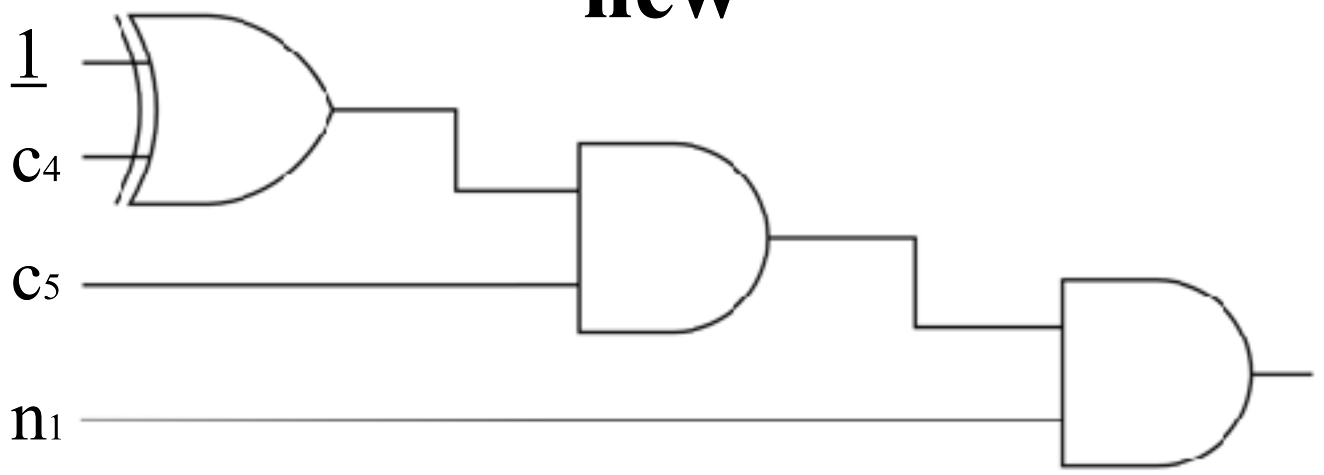
Normalization + Equational Matching

old

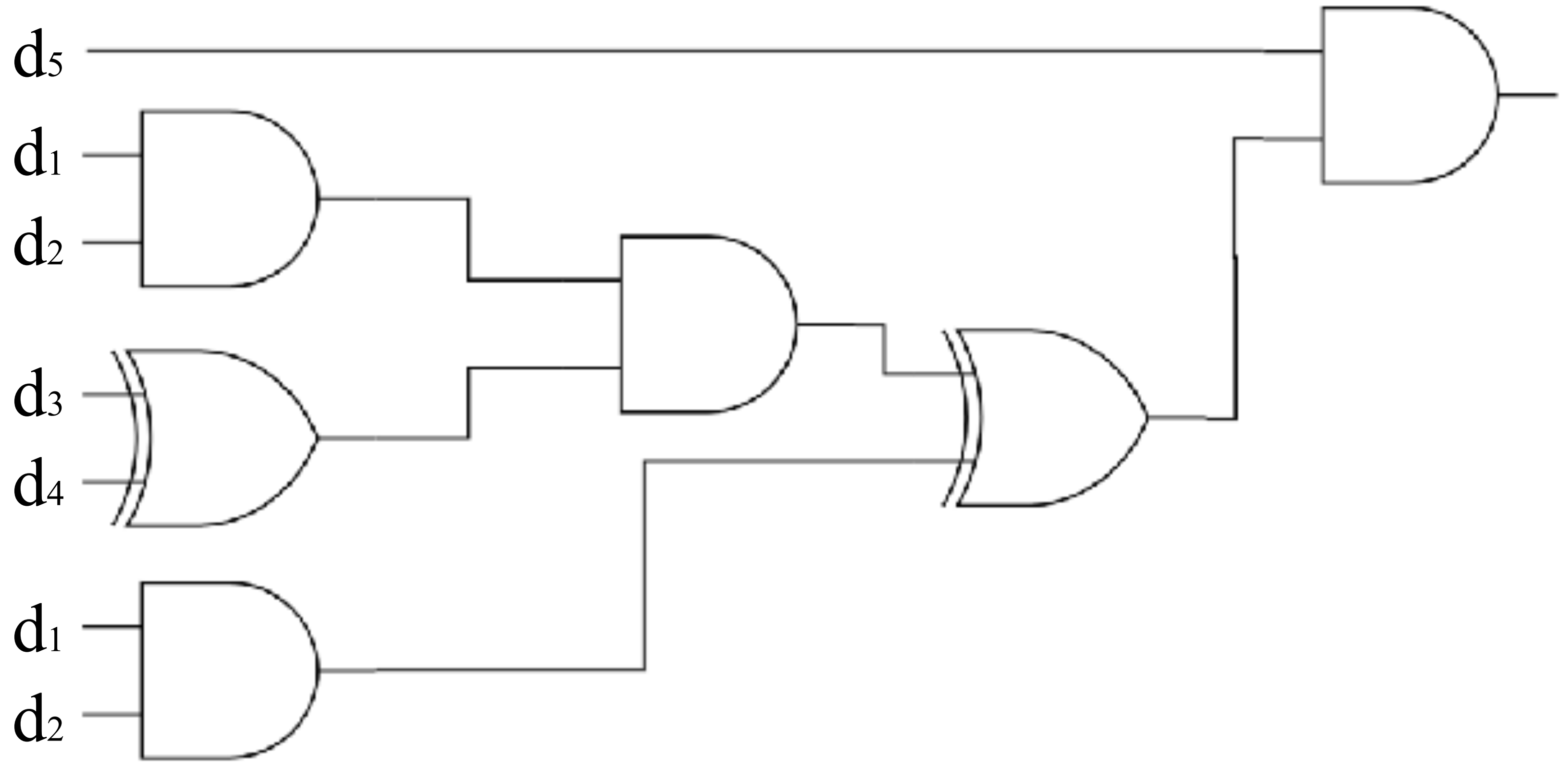


Normalized Opt. Patterns

new



Find substitution σ (considering commutativity)



target

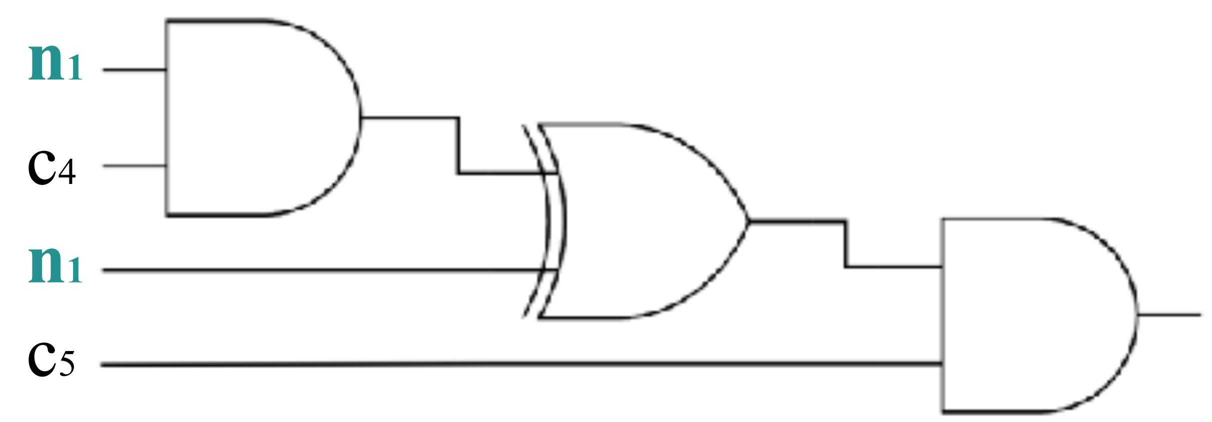
New Input Circuit Optimization

?

Applying Learned Optimization Patterns (2/2)

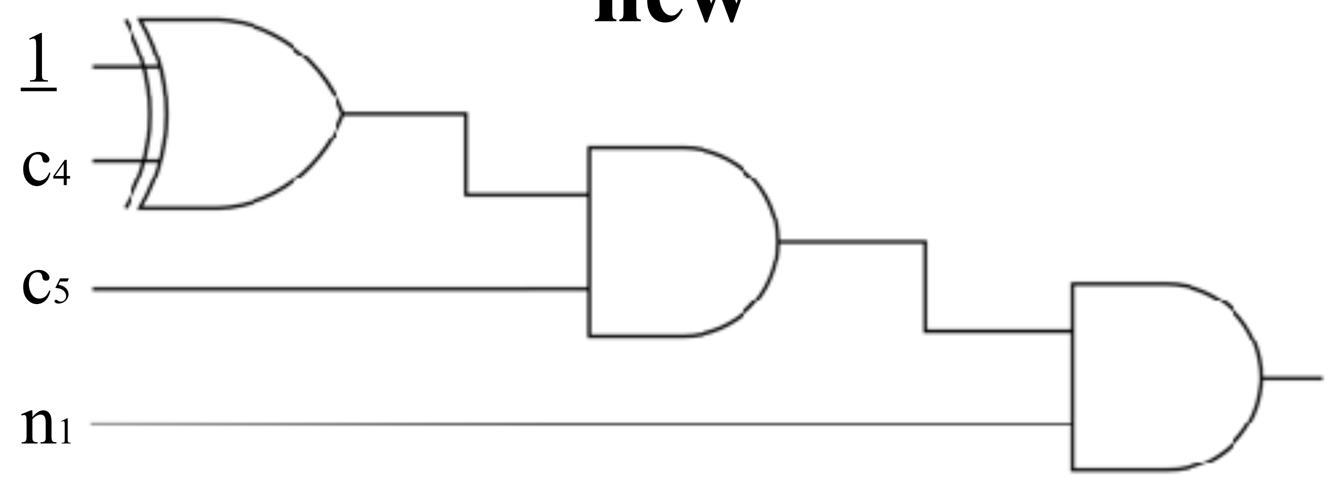
Normalization + Equational Matching

old



Normalized Opt. Patterns

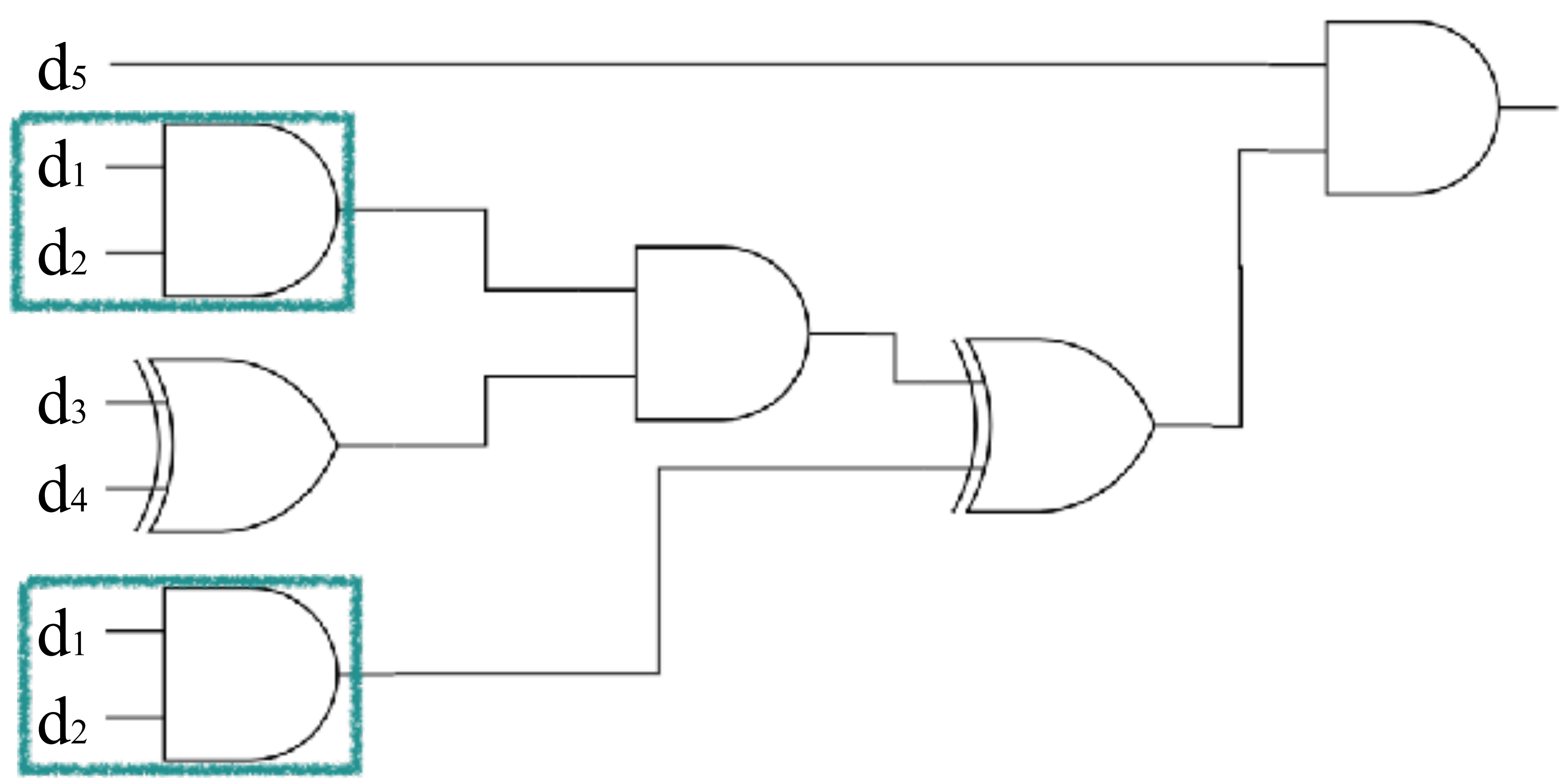
new



Find substitution σ

(considering commutativity)

σ = {n1 ↦ d1 and d2,



target

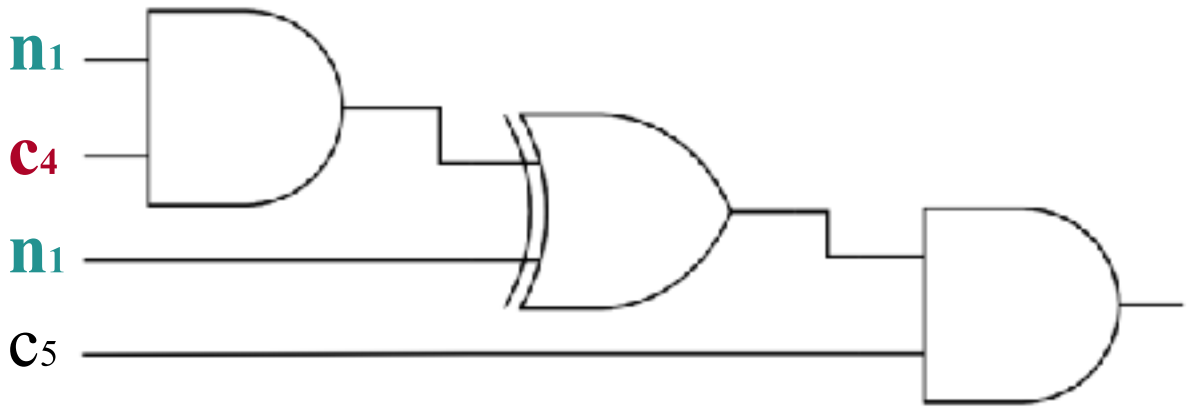
New Input Circuit Optimization

?

Applying Learned Optimization Patterns (2/2)

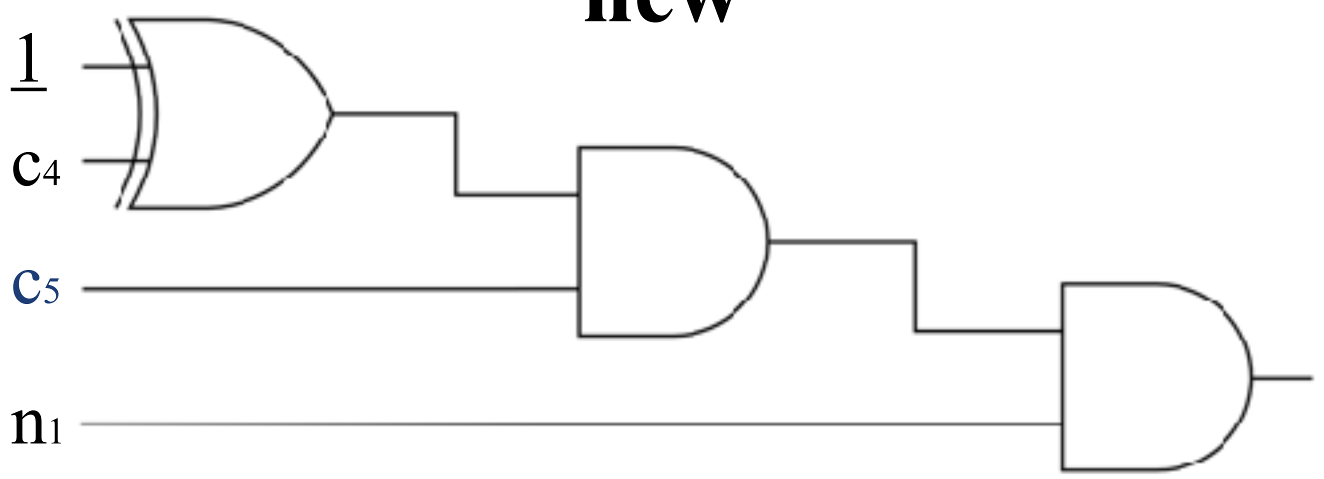
Normalization + Equational Matching

old



Normalized Opt. Patterns

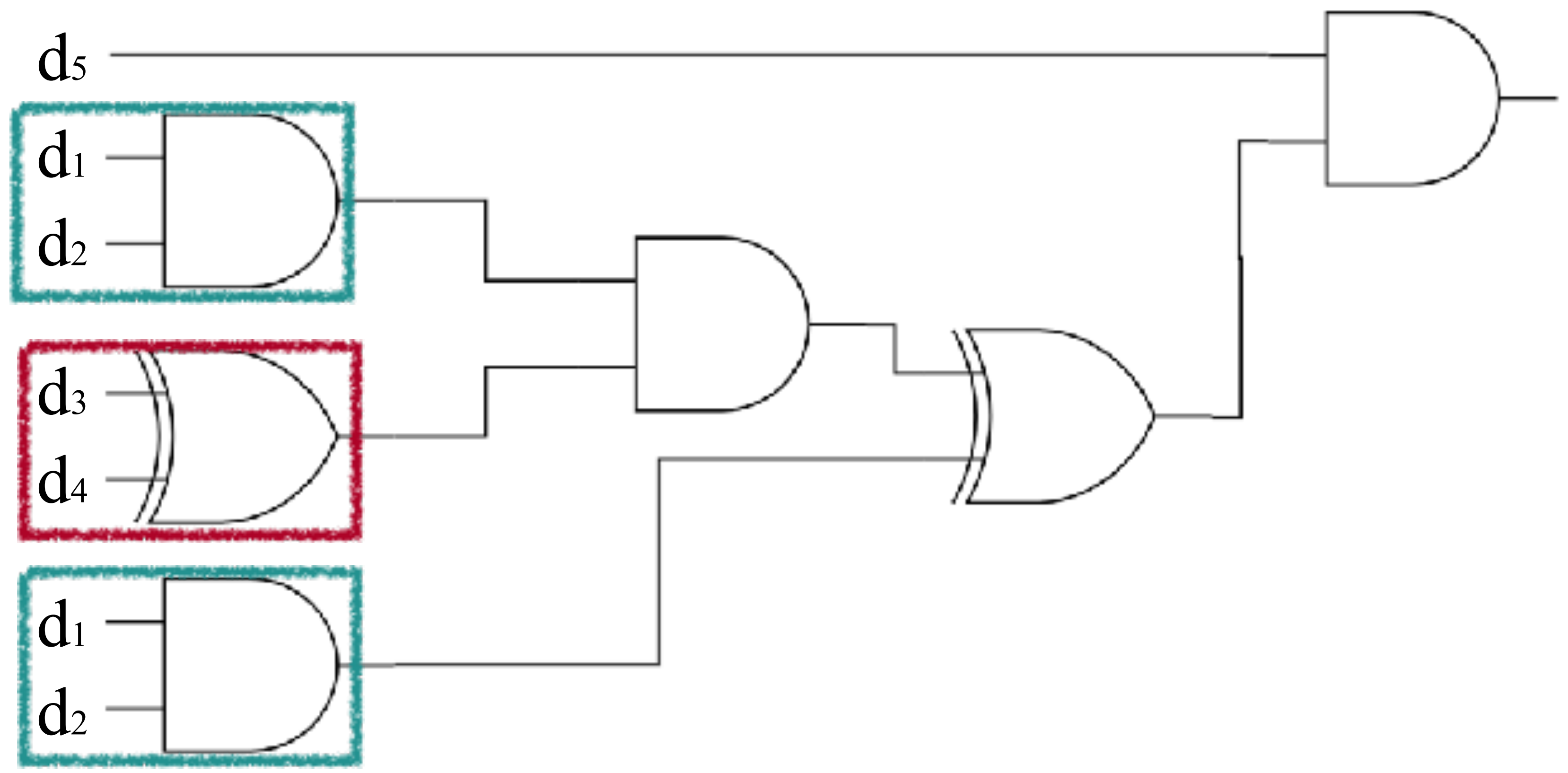
new



Find substitution σ

(considering commutativity)

$\sigma = \{n1 \mapsto d1 \text{ and } d2,$
$c4 \mapsto d3 \text{ xor } d4,$



target

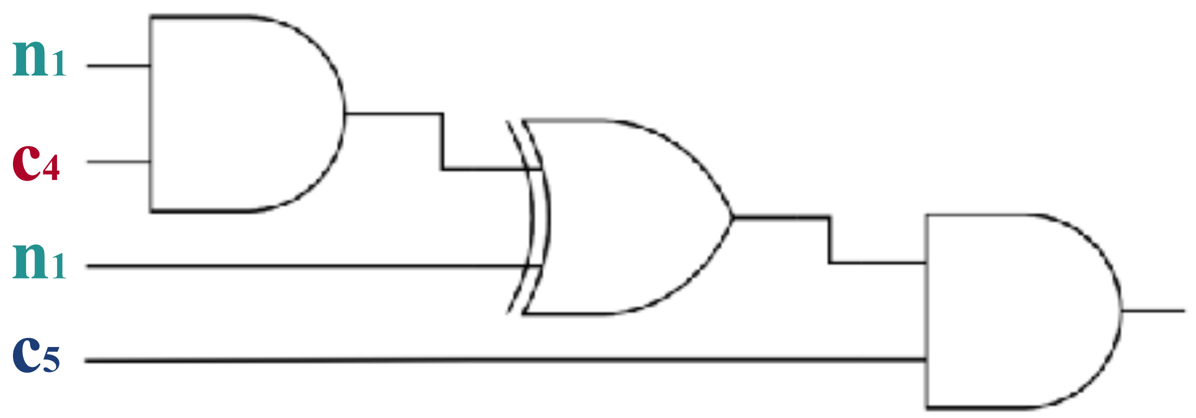
New Input Circuit Optimization

?

Applying Learned Optimization Patterns (2/2)

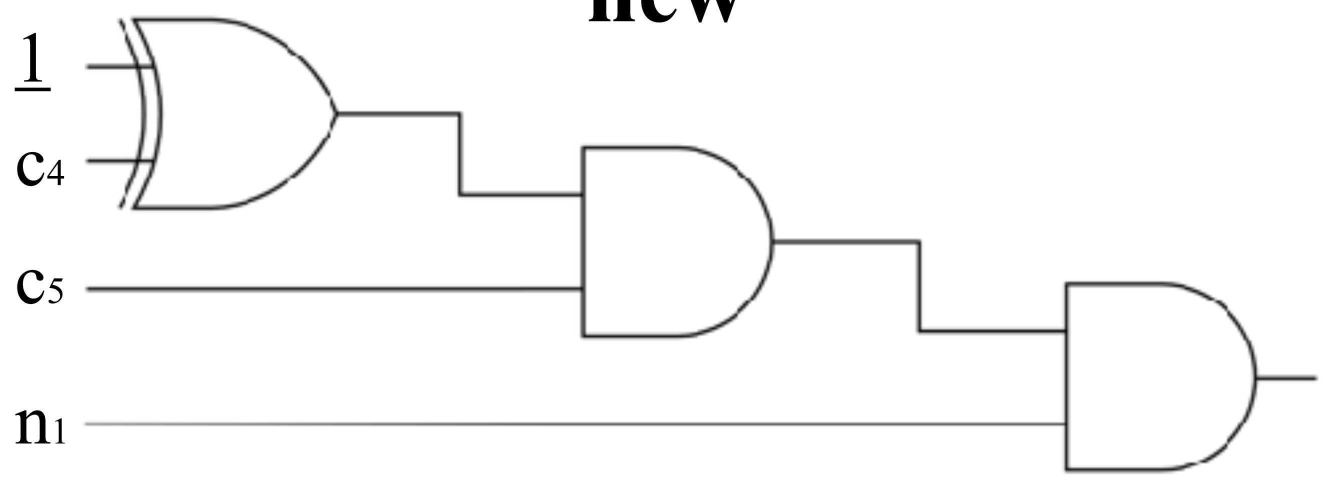
Normalization + Equational Matching

old



Normalized
Opt. Patterns

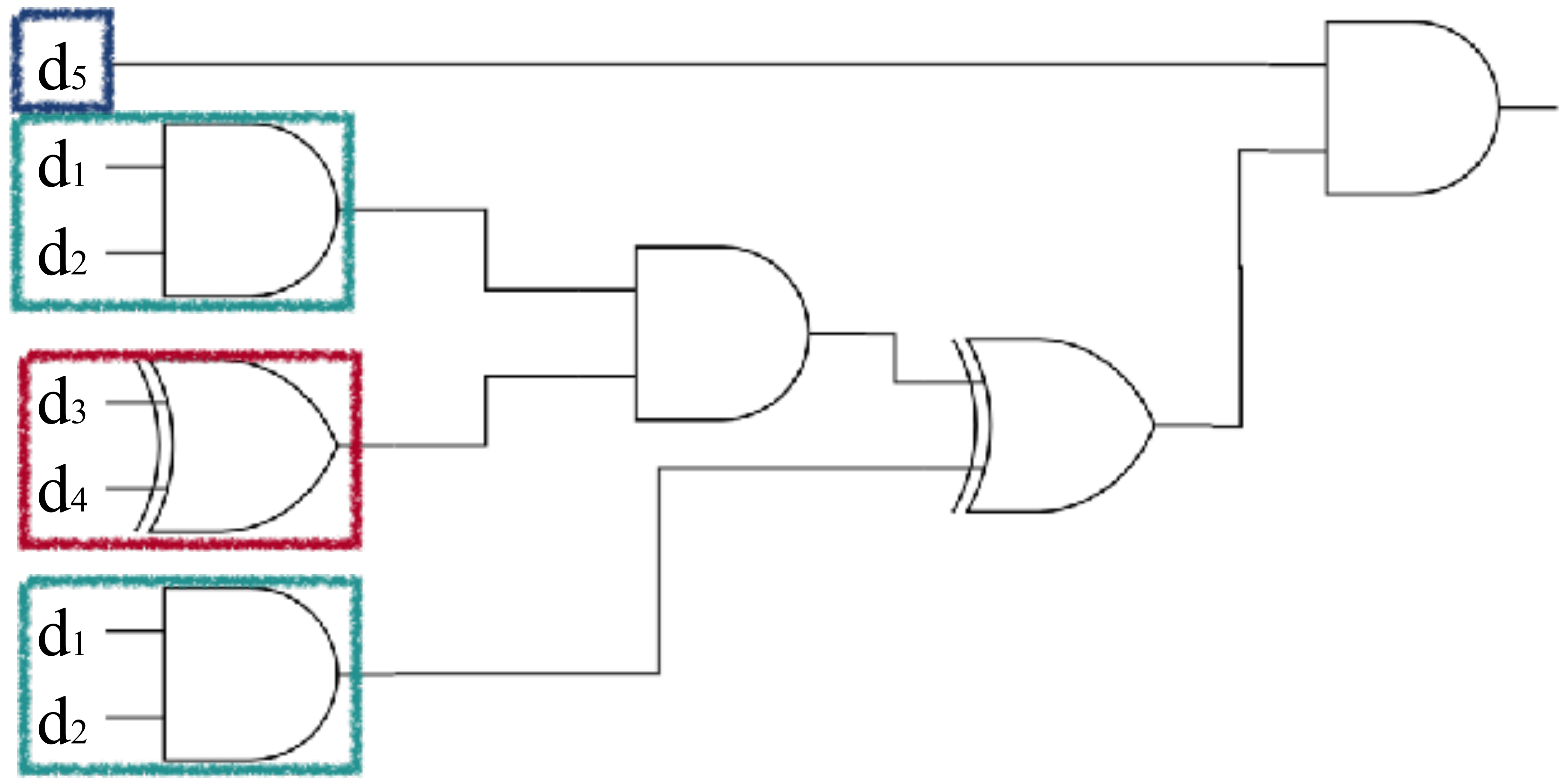
new



Find substitution σ

(considering commutativity)

$\sigma = \{n1 \mapsto d1 \text{ and } d2,$
$c4 \mapsto d3 \text{ xor } d4,$
$c5 \mapsto d5\}$



target

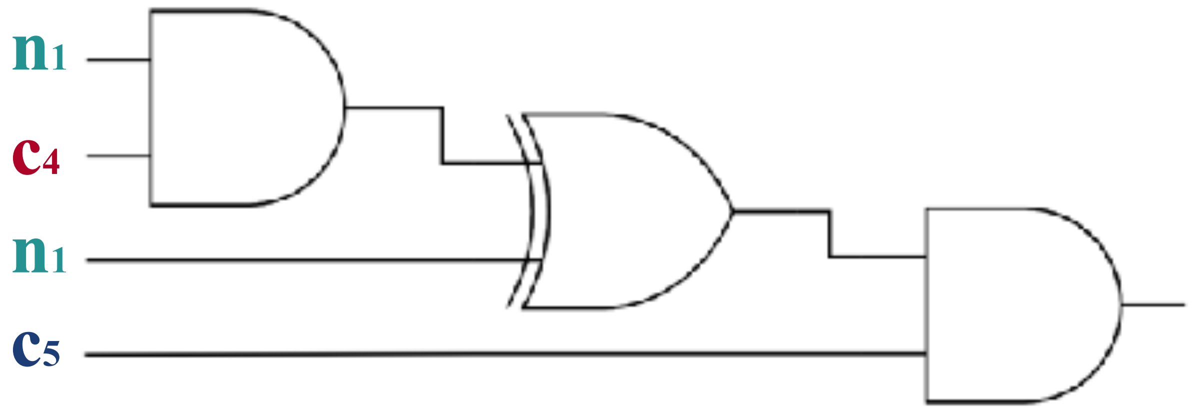
New Input Circuit
Optimization

?

Applying Learned Optimization Patterns (2/2)

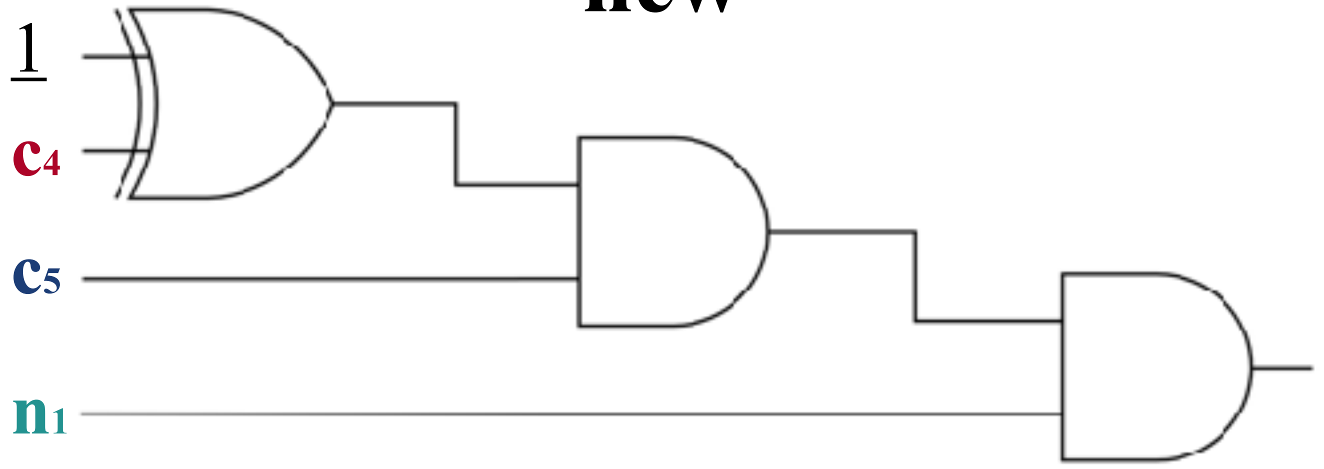
Normalization + Equational Matching

old



Normalized
Opt. Patterns

new

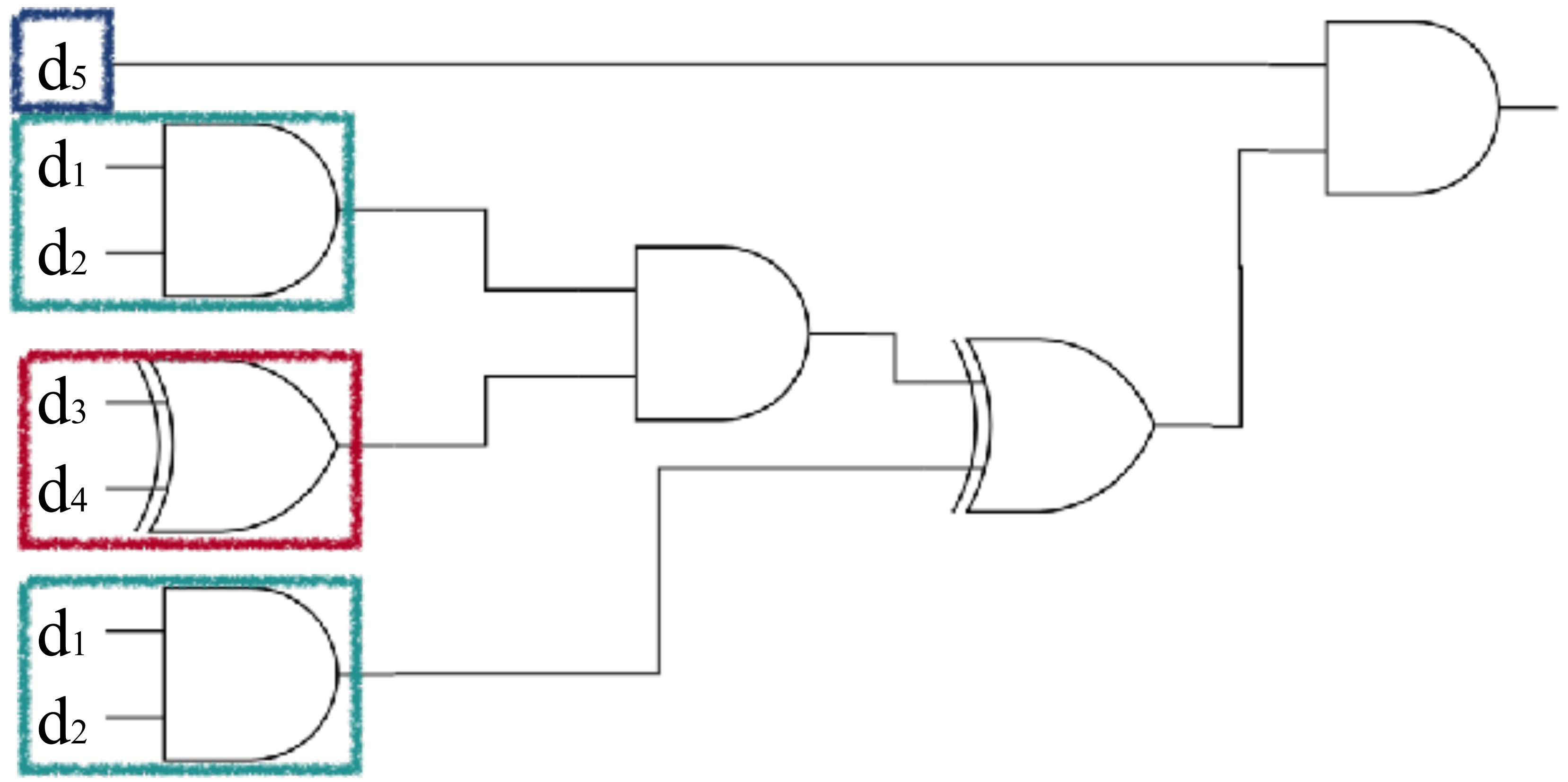


Find substitution σ

(considering commutativity)

Apply substitution σ

σ = {n1 ↦ d1 and d2,
c4 ↦ d3 xor d4,
c5 ↦ d5}



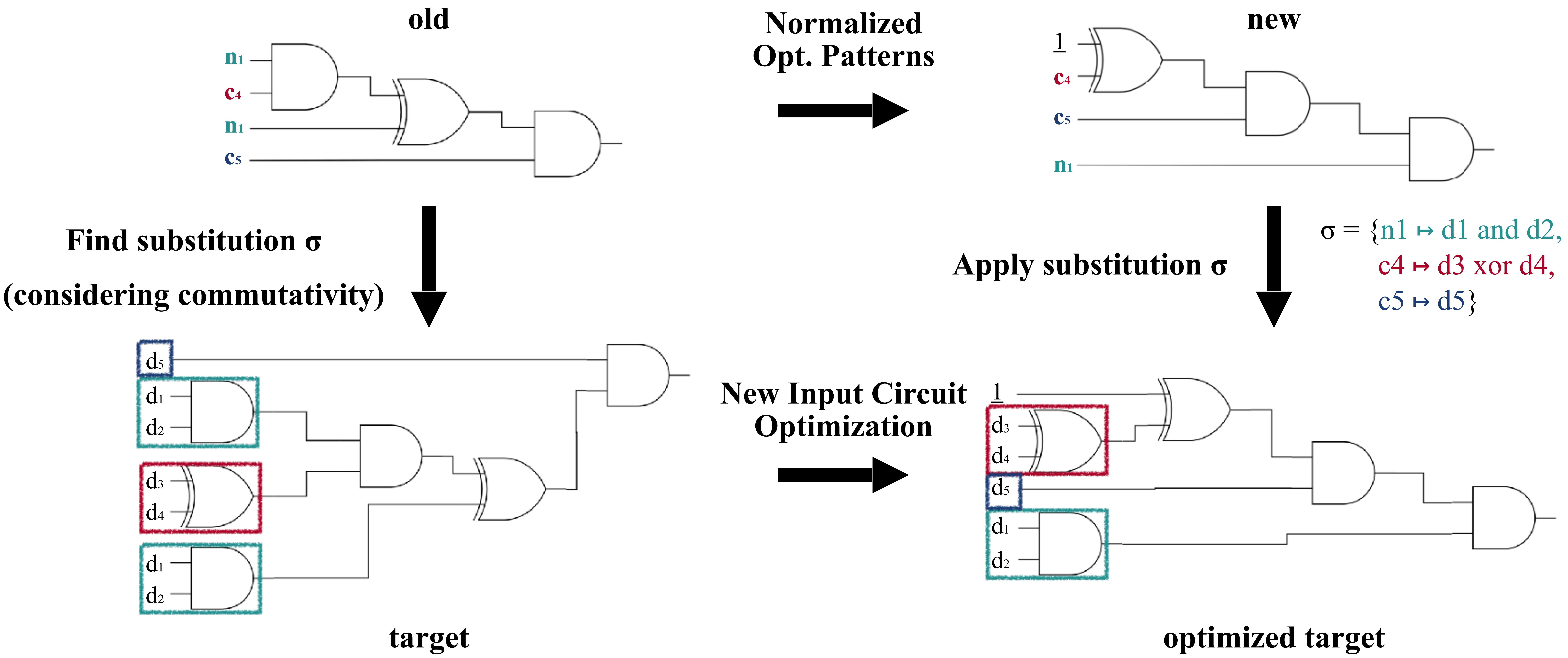
target

New Input Circuit
Optimization

?

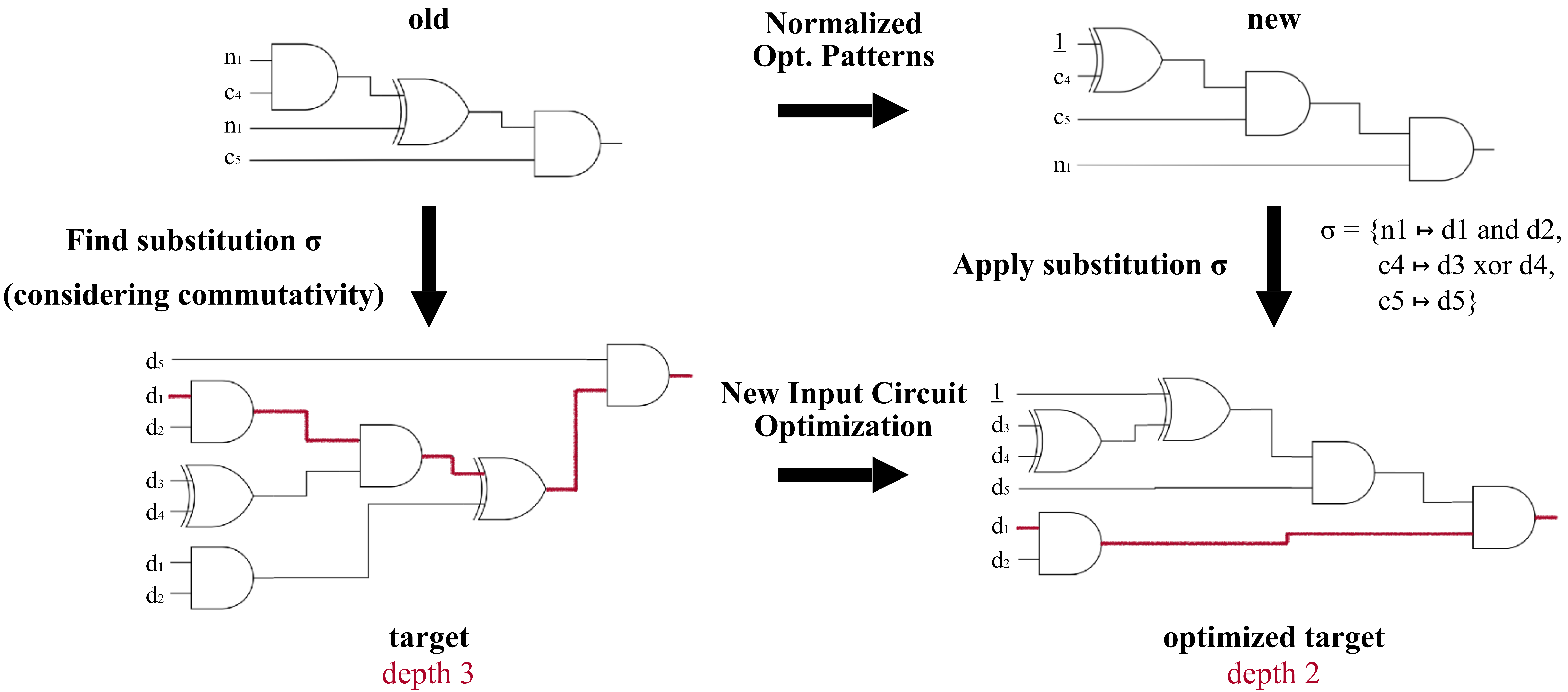
Applying Learned Optimization Patterns (2/2)

Normalization + Equational Matching



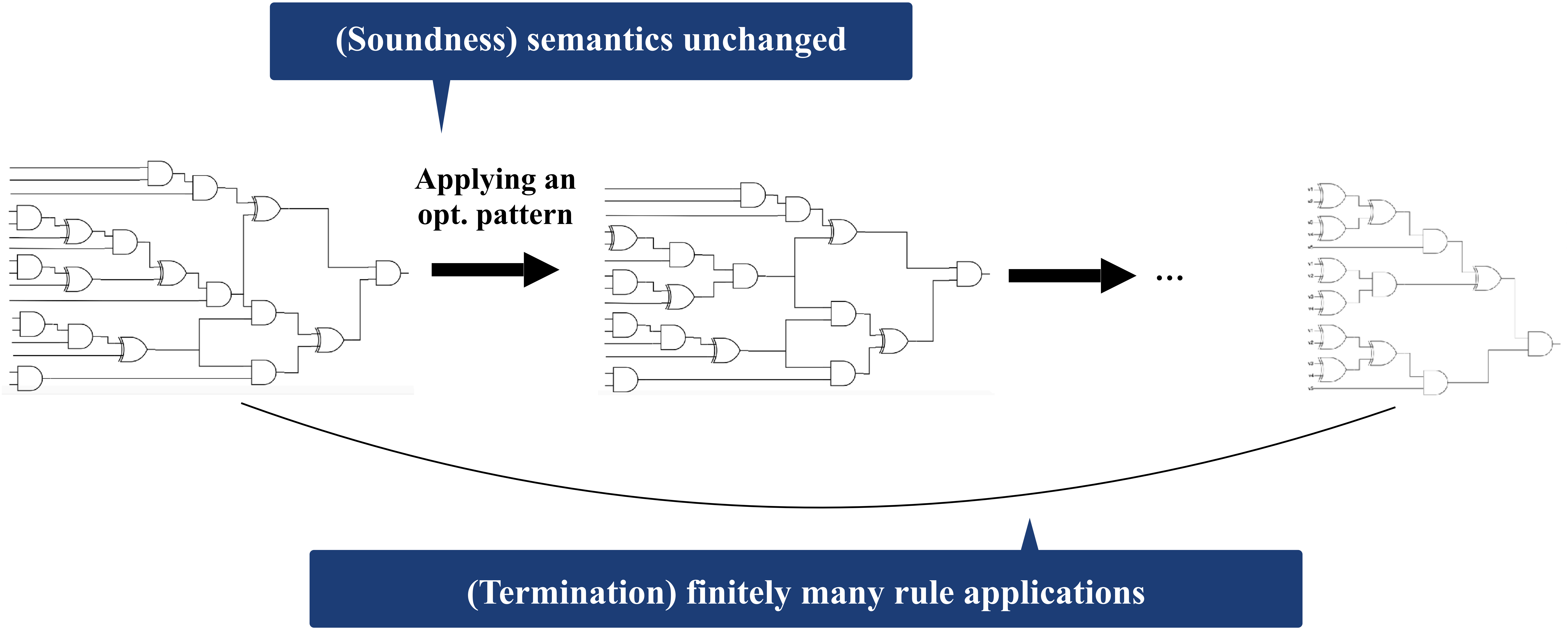
Applying Learned Optimization Patterns (2/2)

Normalization + Equational Matching



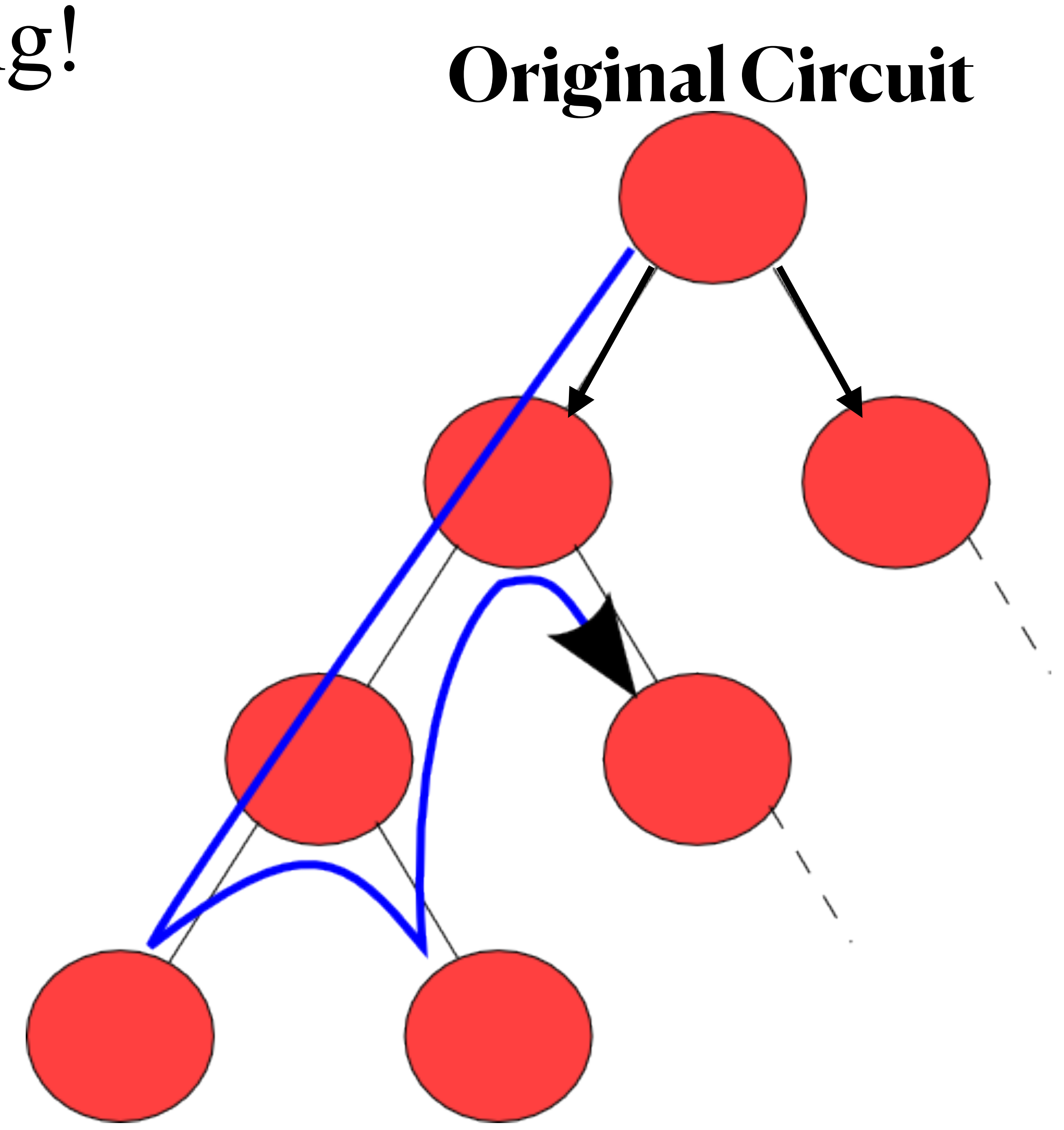
Applying Learned Optimization Patterns

Formal properties



Can We Do Better?

- The term rewriting approach may converge to a sub-optimal result.
- Different optimization rules interact by enabling or disabling opportunities for other optimization rules .
- In other words, no backtracking!



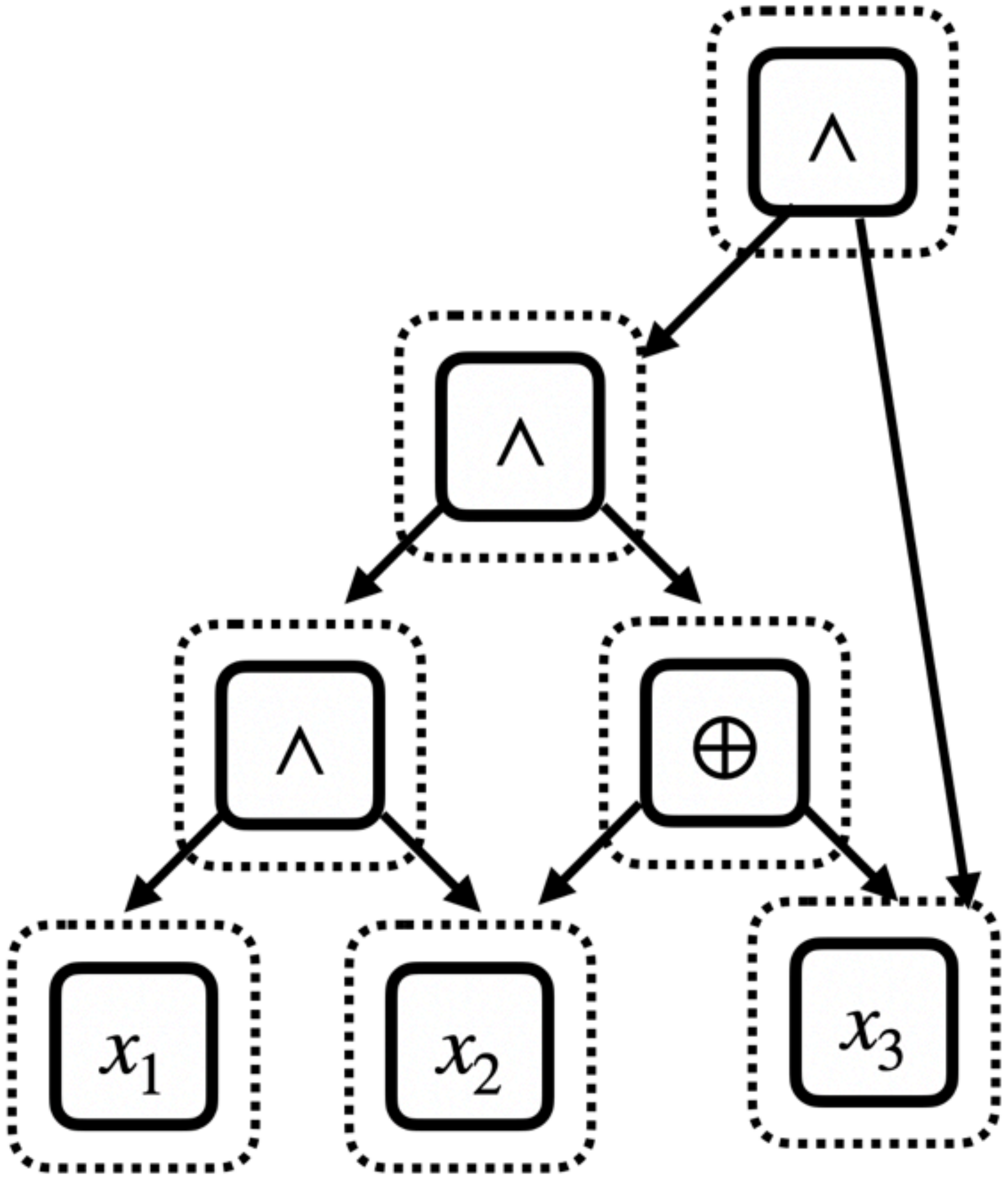
Exhaustive Search with E-graphs

- Idea: Apply the rewrite rules in all possible orders and store all the results
- Example
  - Optimizing a given circuit: $((x_1 \wedge x_2) \wedge (x_2 \oplus x_3)) \wedge x_3$
  - Usable rewrite rules:

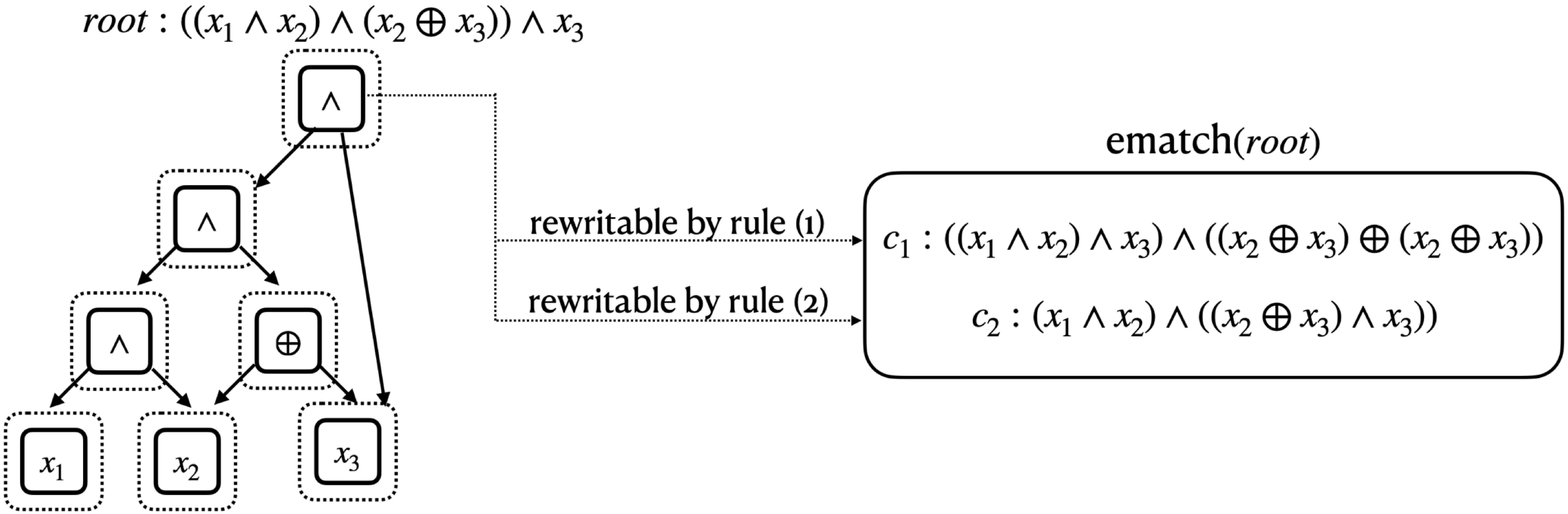
$$
\begin{aligned}
\text{rule (1)}: \quad ((v_1 \wedge v_2) \wedge v_3) \wedge v_4 &\quad\longrightarrow\quad ((v_1 \wedge v_2) \wedge v_4) \wedge ((v_2 \oplus v_4) \oplus v_3) \\
\text{rule (2)}: \quad ((v_1 \wedge v_2) \wedge v_3) \wedge v_4 &\quad\longrightarrow\quad (v_1 \wedge v_2) \wedge (v_3 \wedge v_4) \\
\text{rule (3)}: \quad (v_1 \oplus v_1) &\quad\longrightarrow\quad 0 \\
\text{rule (4)}: \quad (v_1 \wedge 0) &\quad\longrightarrow\quad 0
\end{aligned}
$$

Exhaustive Search with E-graphs

$$root : ((x_1 \wedge x_2) \wedge (x_2 \oplus x_3)) \wedge x_3$$

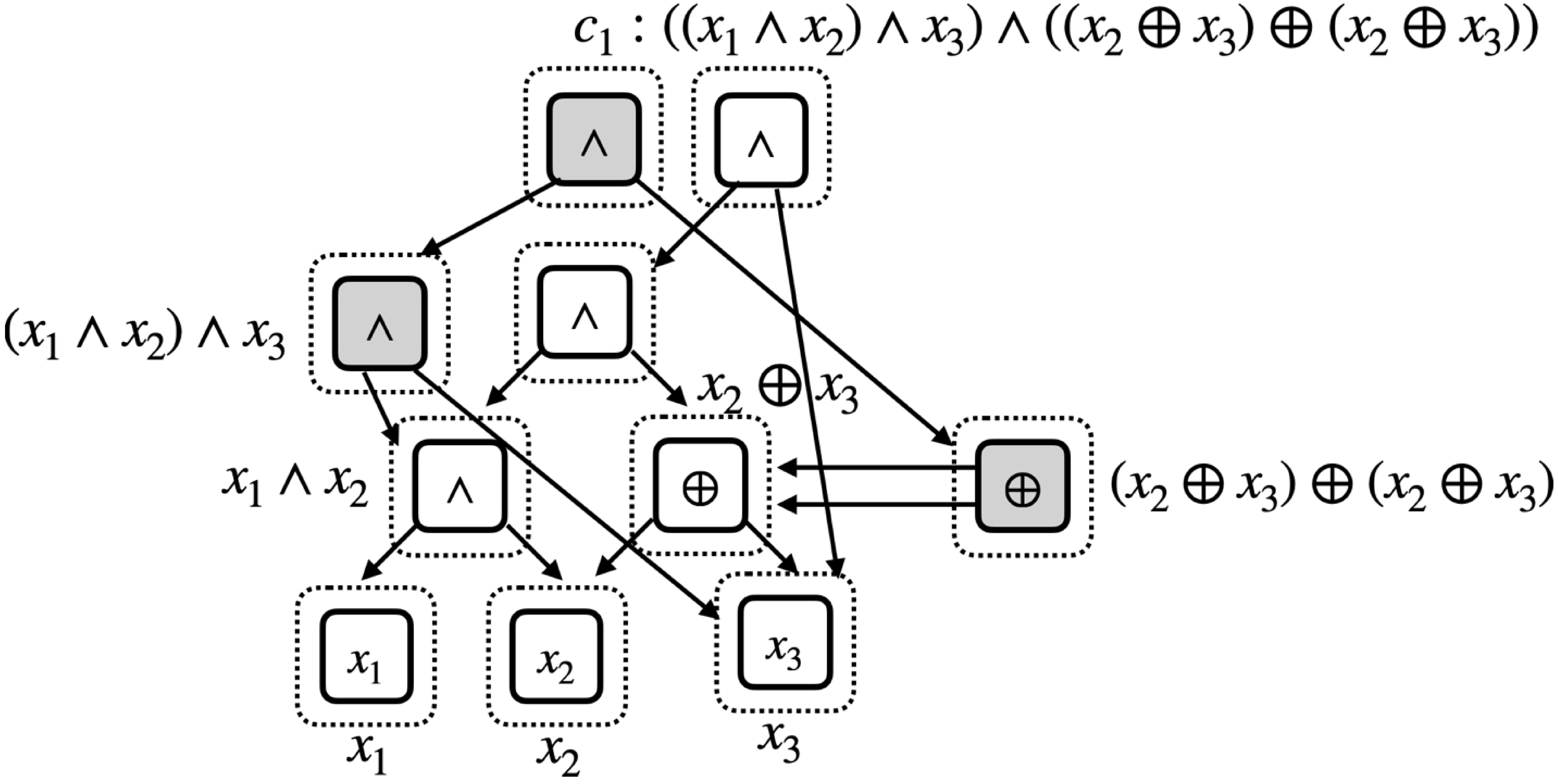


Exhaustive Search with E-graphs



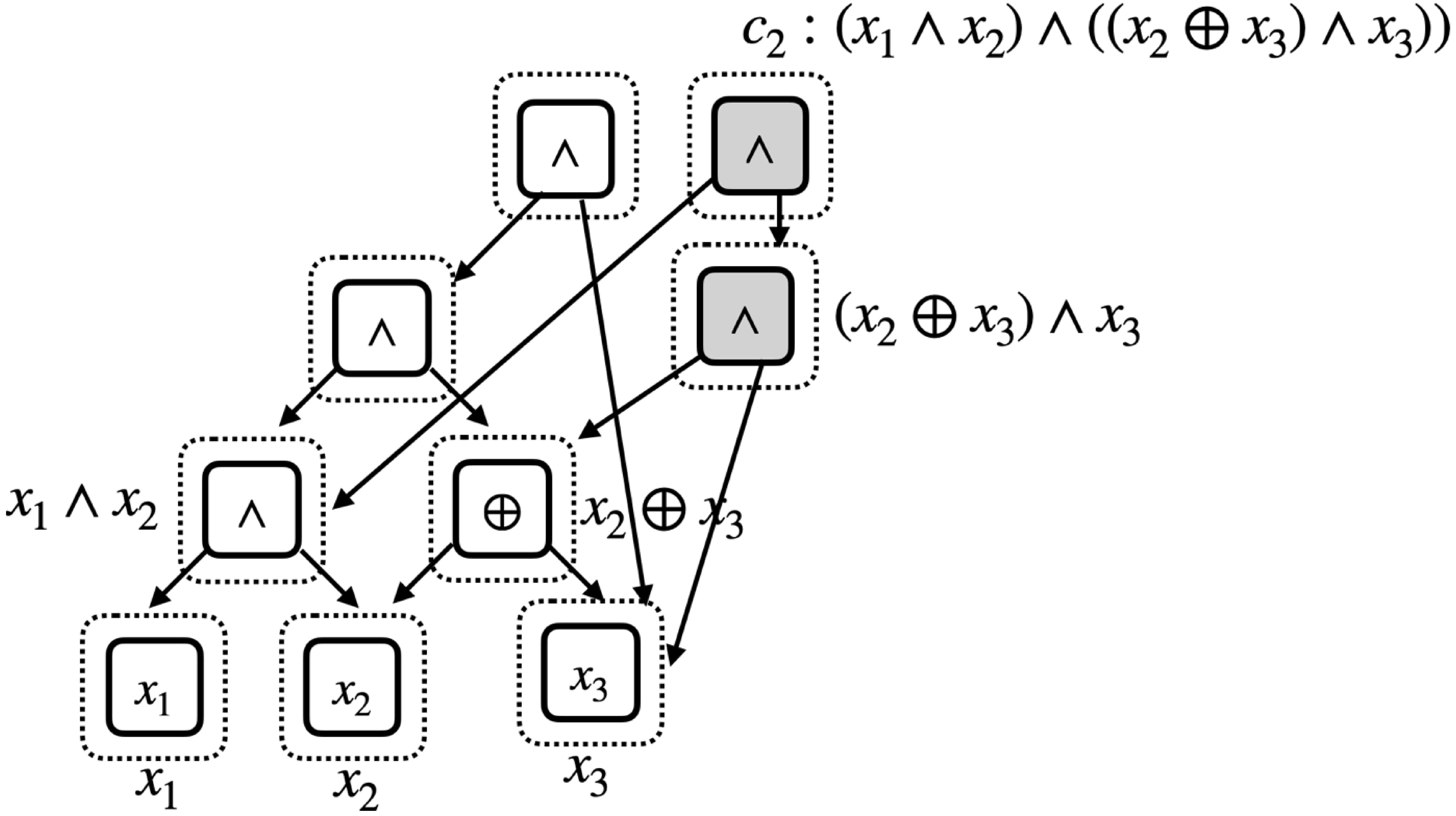
Exhaustive Search with E-graphs

- Applying rule (1)   (shaded: newly added)



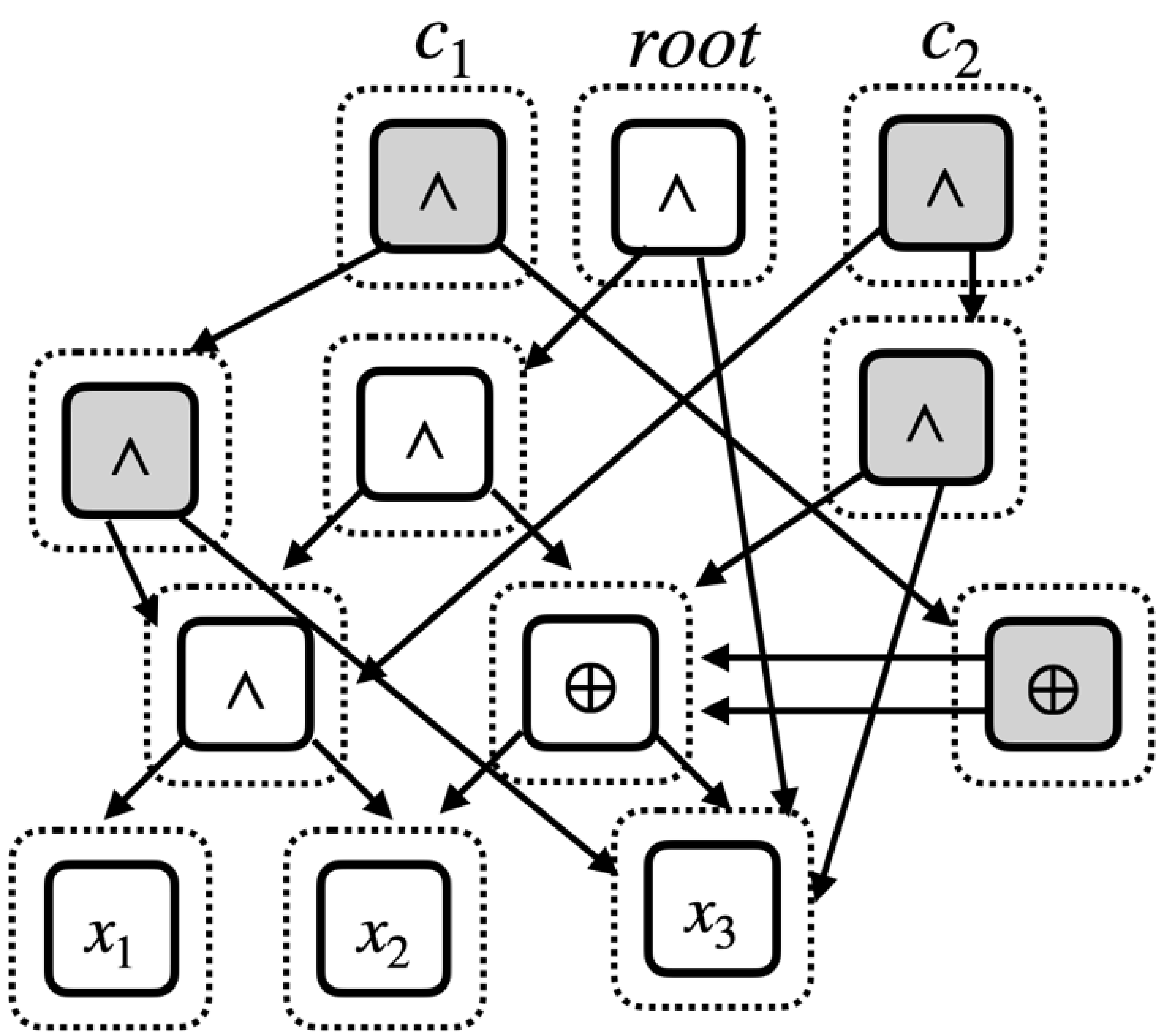
Exhaustive Search with E-graphs

- Applying rule (2)

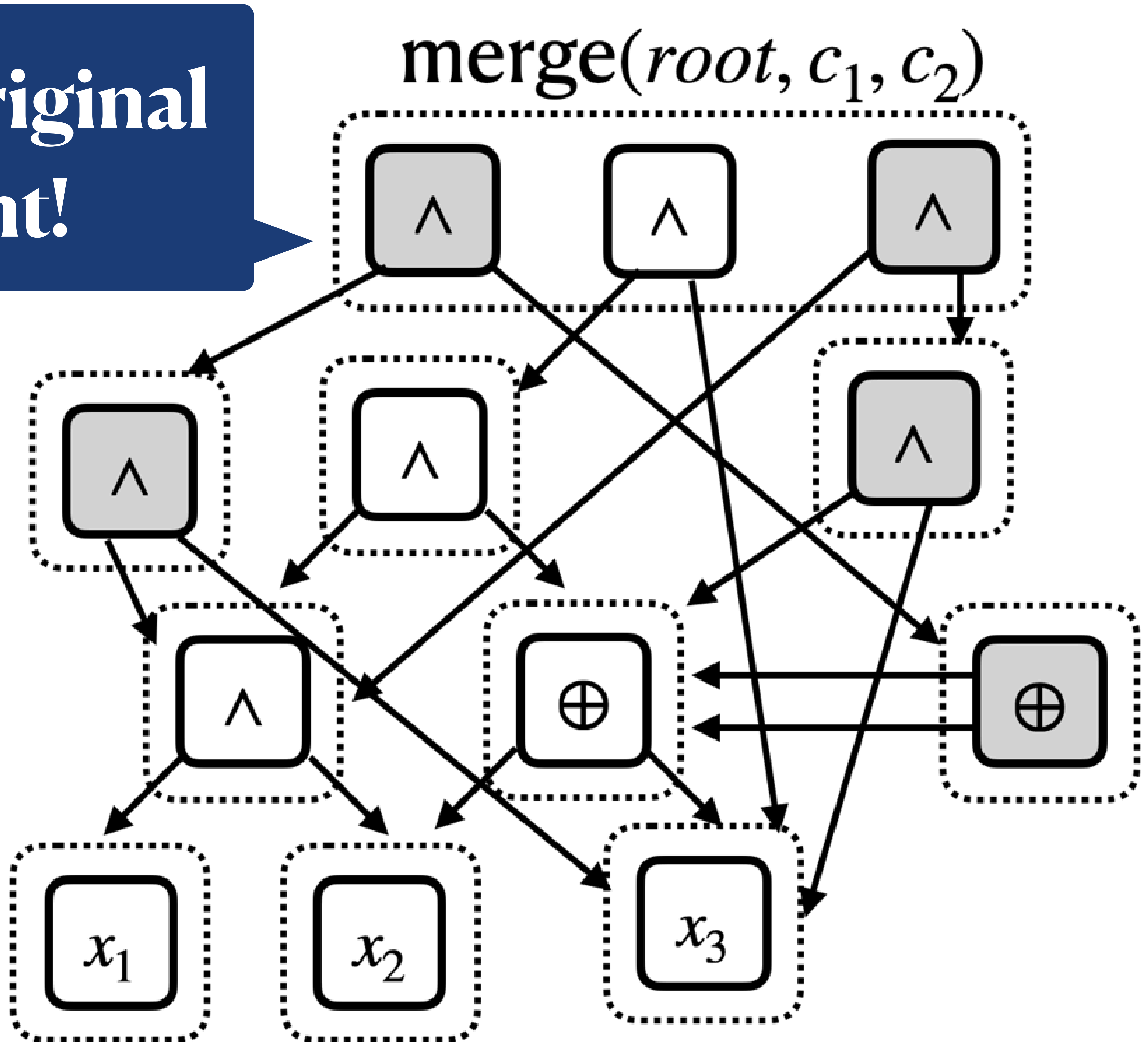


Exhaustive Search with E-graphs

- Merging the results of applying rules (1) and (2)

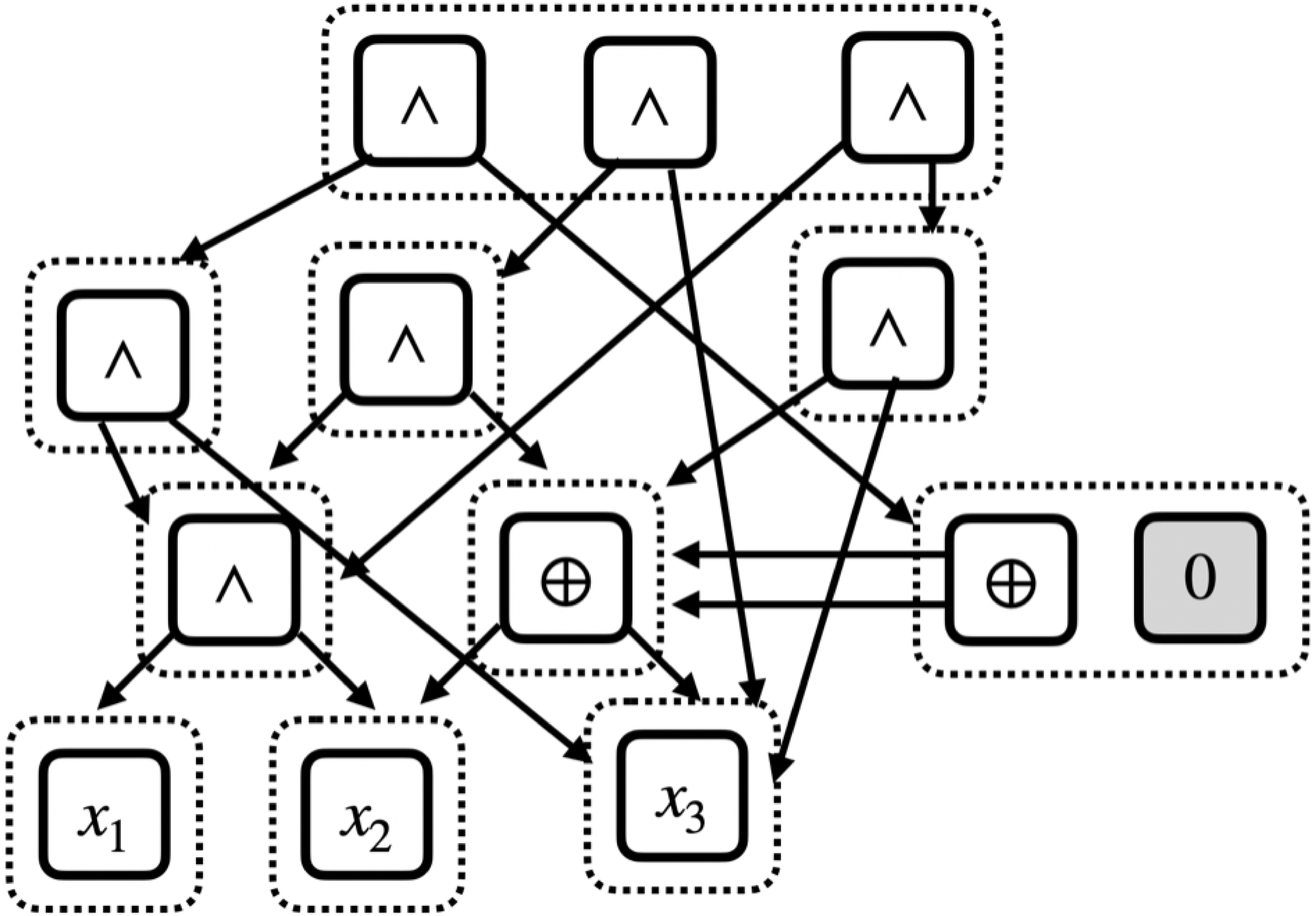


Meaning: c1, c2, and original circuit are all equivalent!



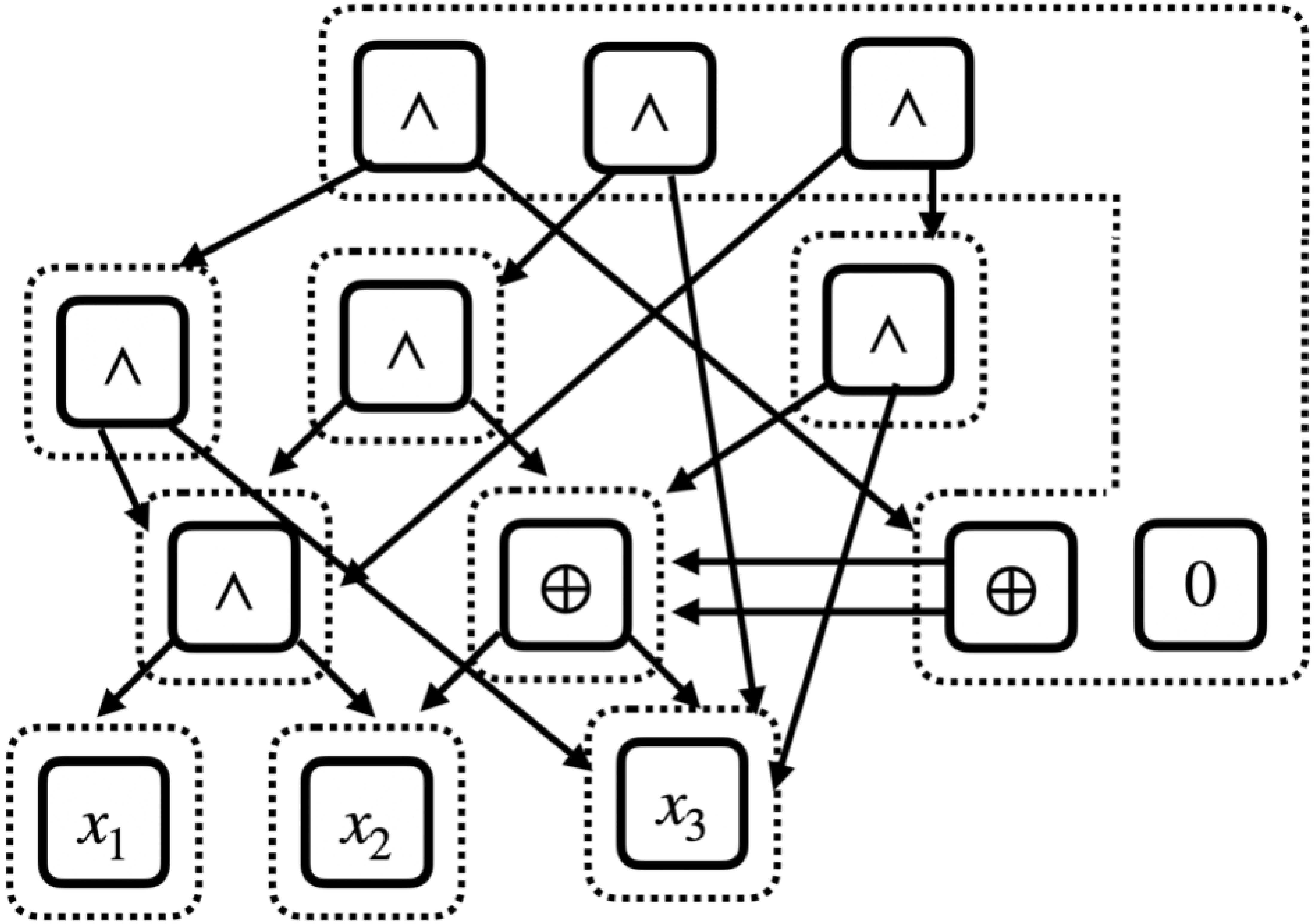
Exhaustive Search with E-graphs

- Applying rule (3) $(v_1 \oplus v_1) \quad \rightarrow \quad 0$ and merging



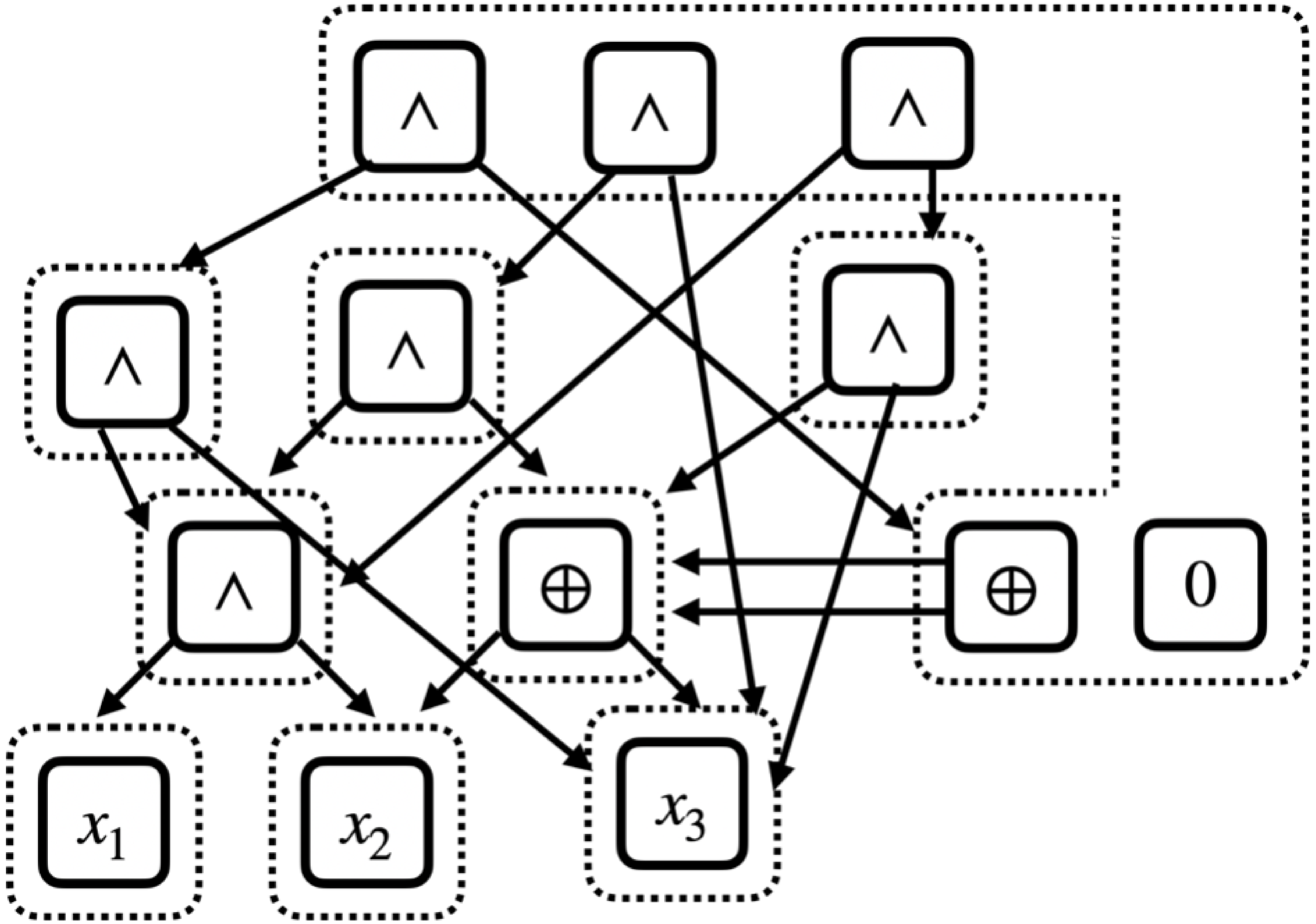
Exhaustive Search with E-graphs

- Applying rule (4) $(v_1 \wedge 0) \quad \rightarrow \quad 0$ and merging



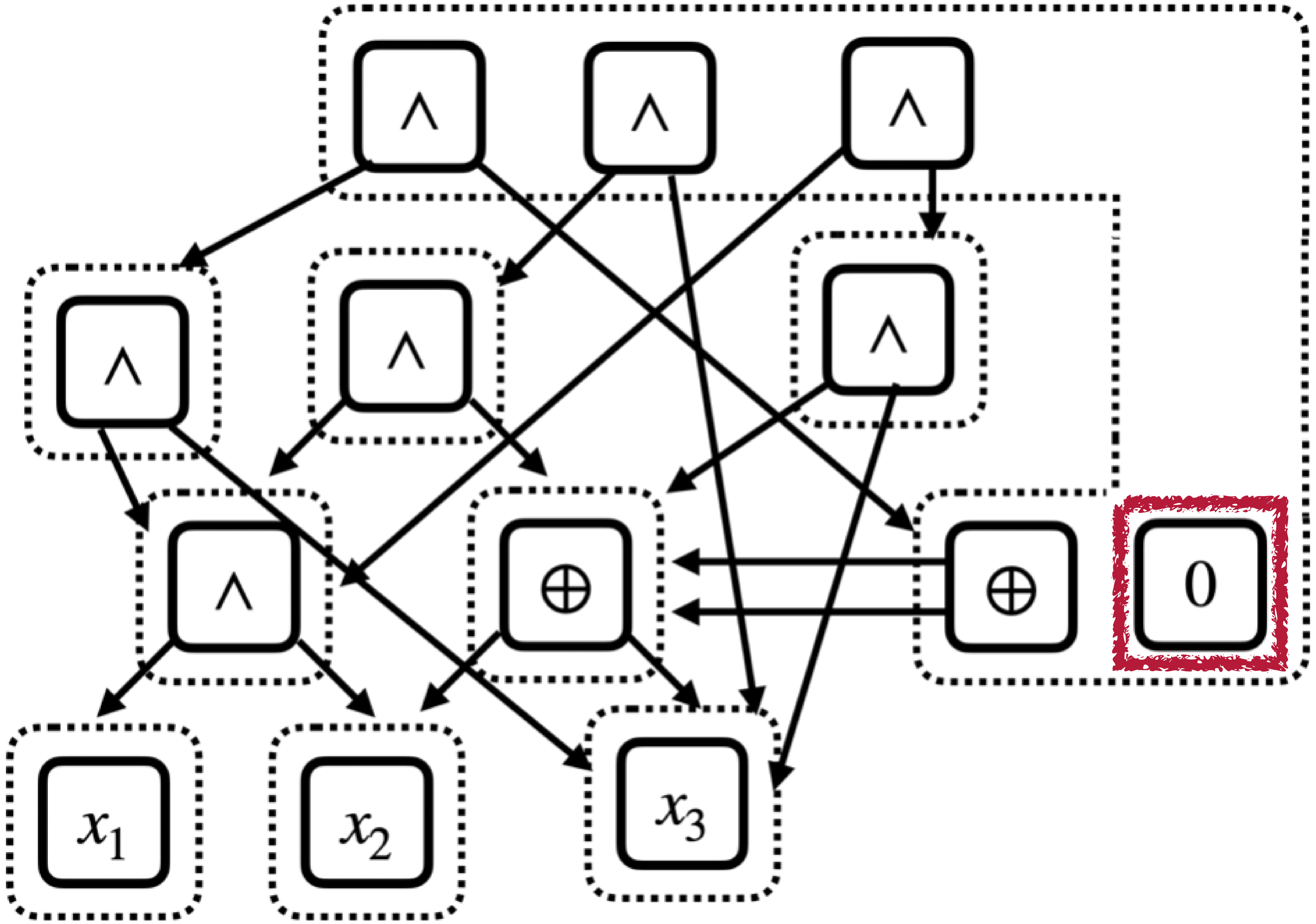
Exhaustive Search with E-graphs

- Applying any rule doesn't change the e-graph => **saturated!**



Exhaustive Search with E-graphs

- Pick the best result: **0**



Rewriting vs. Exhaustive search (aka equality saturation)

- Rewriting: scalable but may find a sub-optimal result
- Equality saturation: unscalable but guaranteed to find an optimal result†
- We use the rewriting method for large circuits and equality saturation for small circuits.

† egg: Fast and Extensible Equality Saturation, ACM POPL 2021

Lobster Performance (1/4)

Benchmarks

- 25 HE algorithms from 4 sources
  - Cingulata benchmarks
  - Sorting benchmarks
  - Hackers Delight benchmarks
  - EPFL benchmarks

2 HE friendly algorithms (medical, sorting)

4 privacy-preserving sorting algorithms (merge, insert, bubble, odd-even)

12 Homomorphic bitwise operations

7 EPFL combinational benchmark suite (to test circuit optimizer)

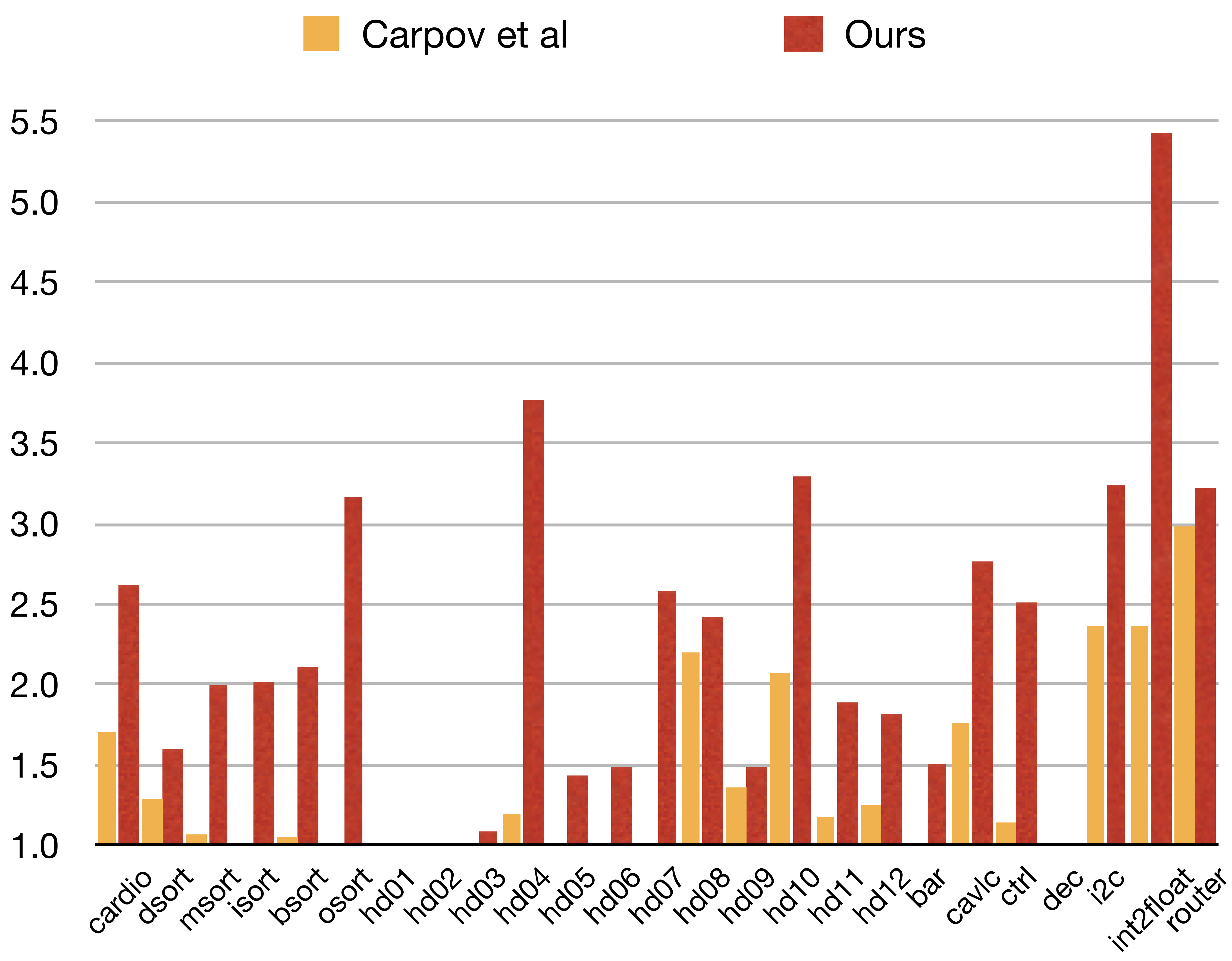
Lobster Performance (1/4)

Benchmarks

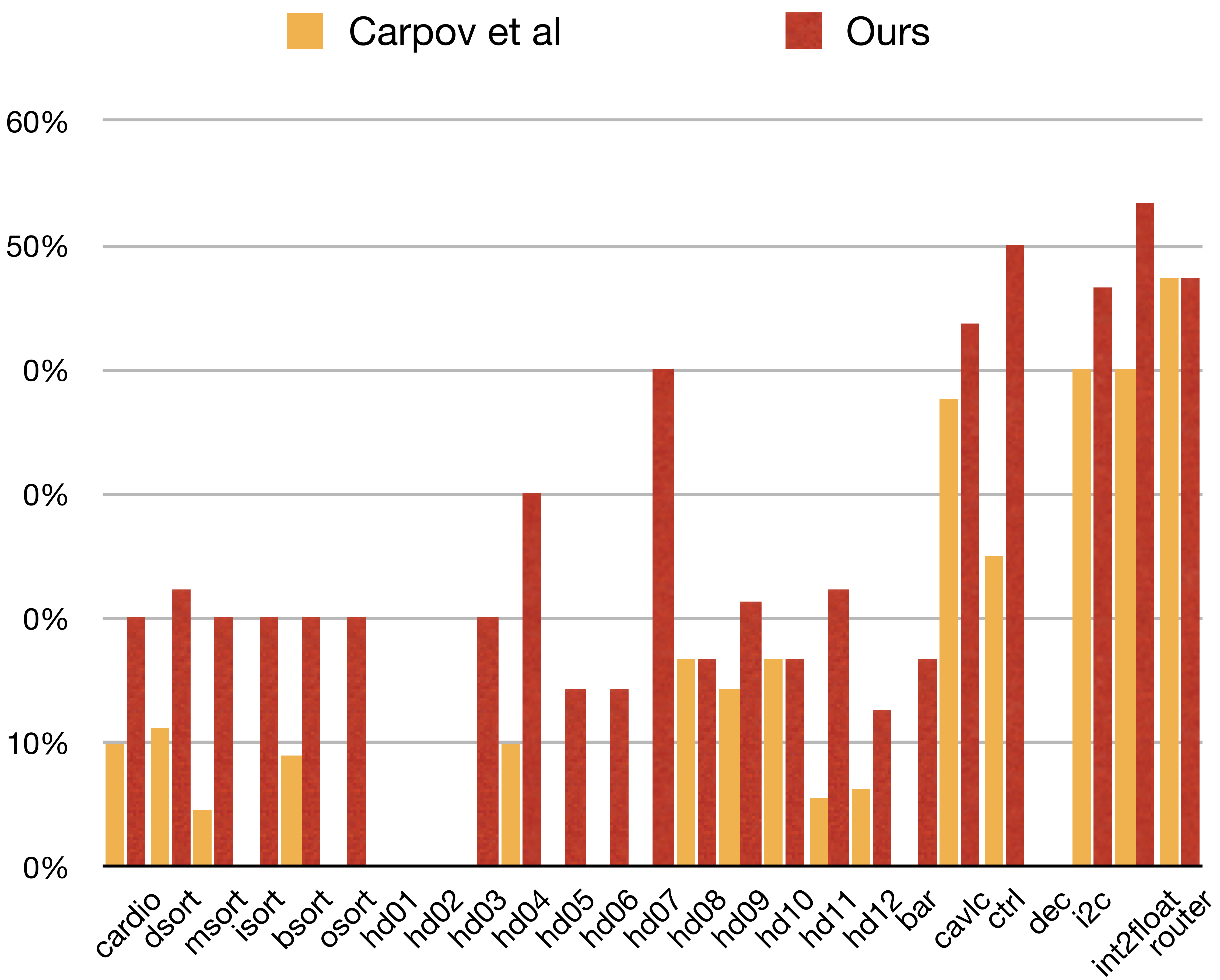
Name	Description	×Depth	#AND	Size
cardio	medical diagnostic algorithm [16]	10	109	318
dsort	FHE-friendly direct sort [17]	9	708	1464
msort	merge sort [17]	45	810	1525
isort	insertion sort [17]	45	810	1525
bsort	bubble sort [17]	45	810	1525
osort	oddeven sort [17]	25	702	1343
hd-01	isolate the rightmost 1-bit [35]	6	87	118
hd-02	absolute value [35]	6	76	229
hd-03	floor of average of two integers (a clever impl.) [35]	5	27	64
hd-04	floor of average of two integers (a naive impl.) [52]	10	75	159
hd-05	max of two integers [35]	7	121	295
hd-06	min of two integers [35]	7	121	295
hd-07	turn off the rightmost contiguous string of 1-bits [35]	5	17	32
hd-08	determine if an integer is a power of 2 [35]	6	18	37
hd-09	round up to the next highest power of 2 [35]	14	134	236
hd-10	find first 0-byte [52]	6	35	73
hd-11	the longest length of contiguous string of 1-bits [52]	18	391	652
hd-12	number of leading 0-bits [52]	16	116	232
bar	barrel shifter [1]	12	3141	5710
cavlc	coding-cavlc [1]	16	655	1219
ctrl	ALU control unit [1]	8	107	180
dec	decoder [1]	3	304	312
i2c	i2c controller [1]	15	1157	1987
int2float	int to float converter [1]	15	213	386
router	lookahead XY router [1]	19	170	277

Lobster Performance (2/4)

Optimization Results of Lobster and the baseline



Speedup

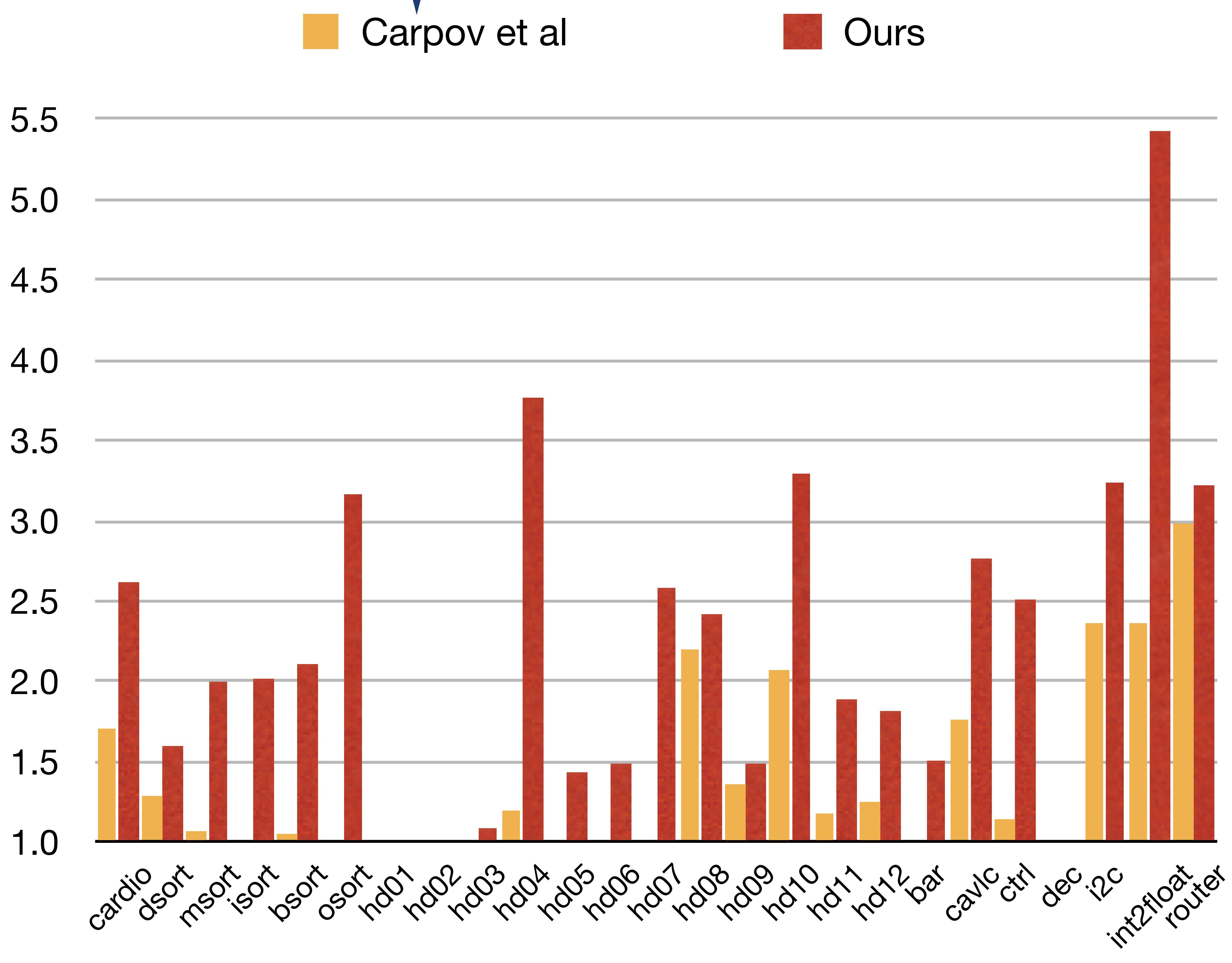


Depth Reduction

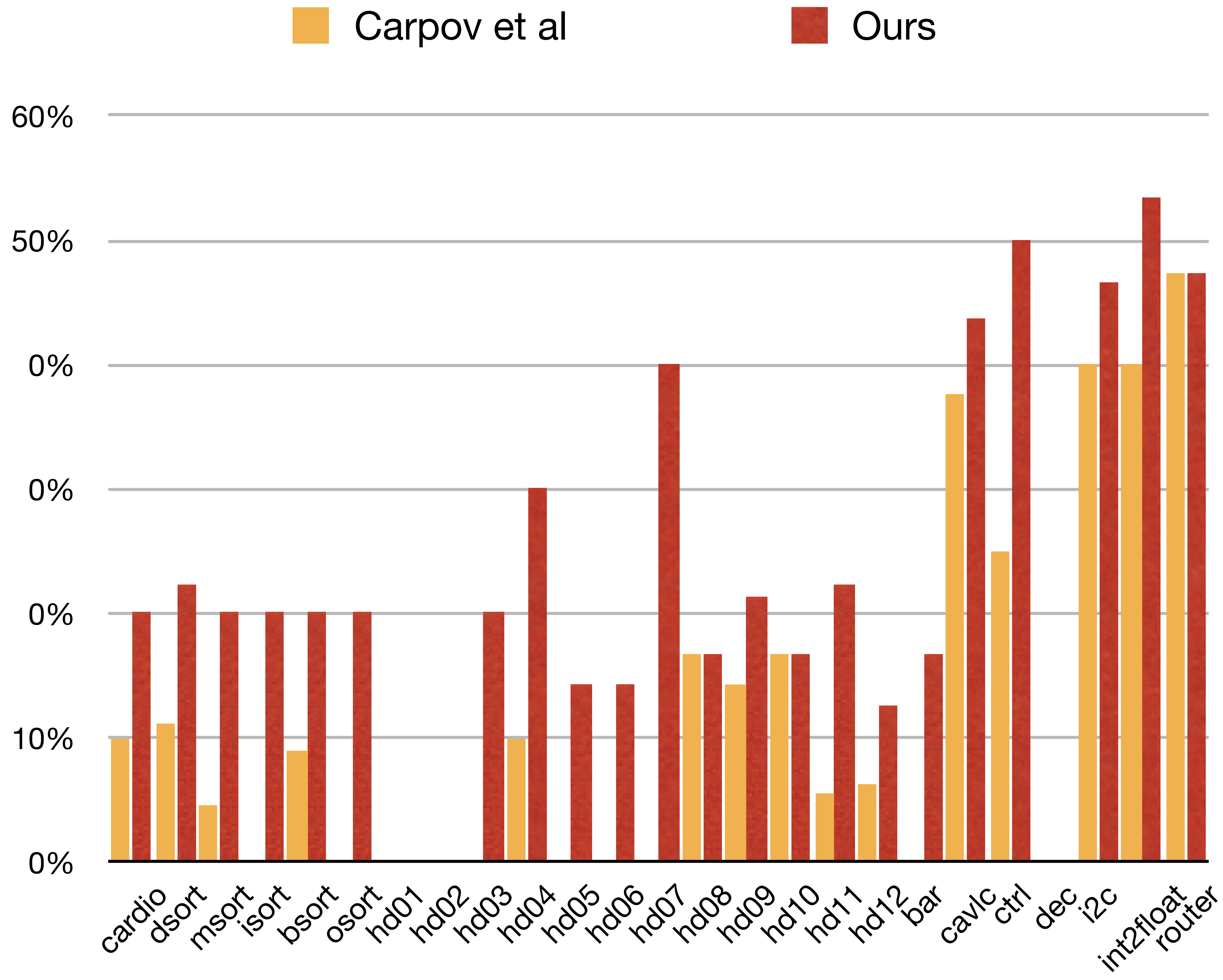
Lobster Performance (2/4)

Hand-written-rule based HE circuit optimizer

Optimization Results of Lobster and the baseline



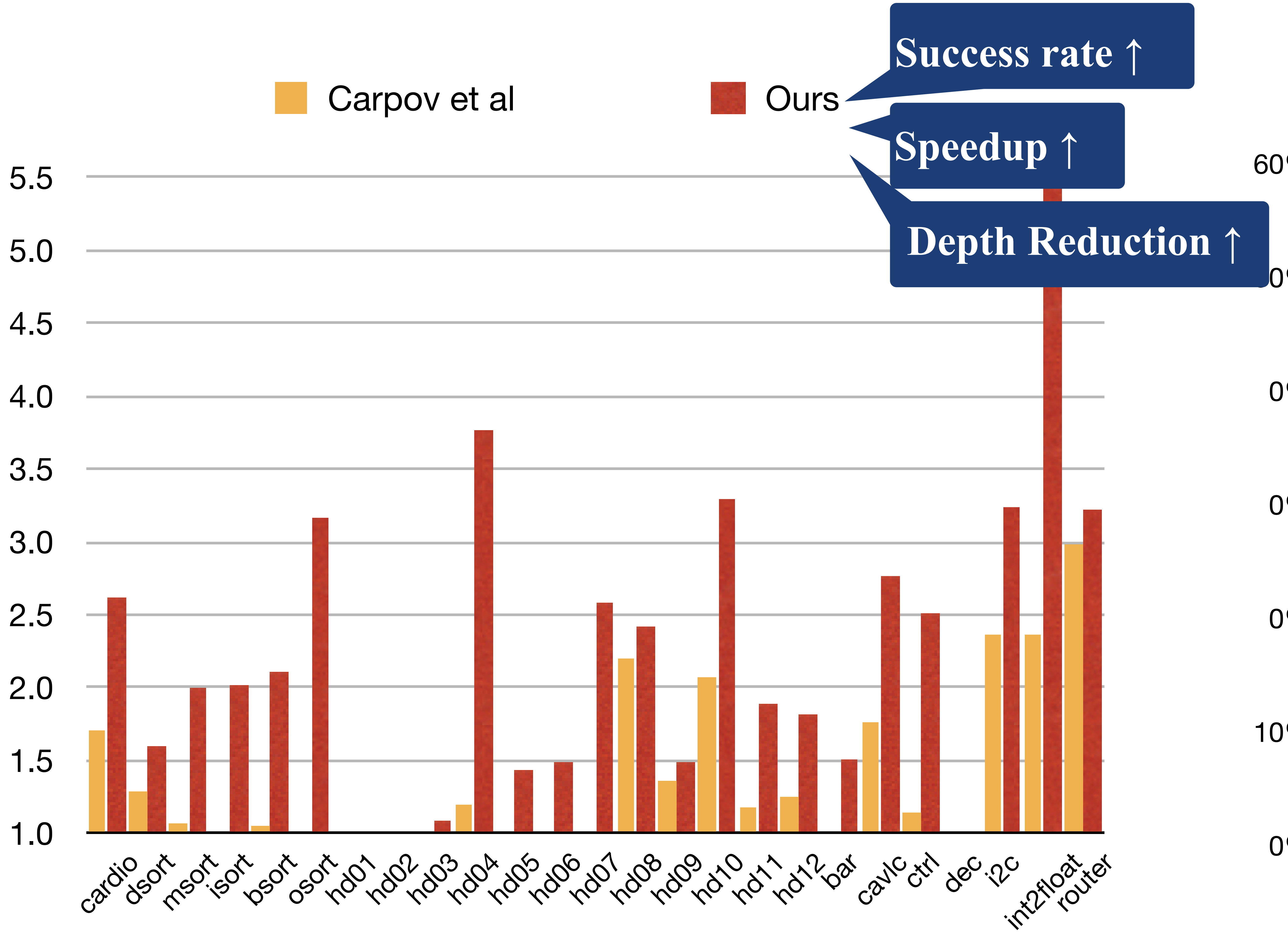
Speedup



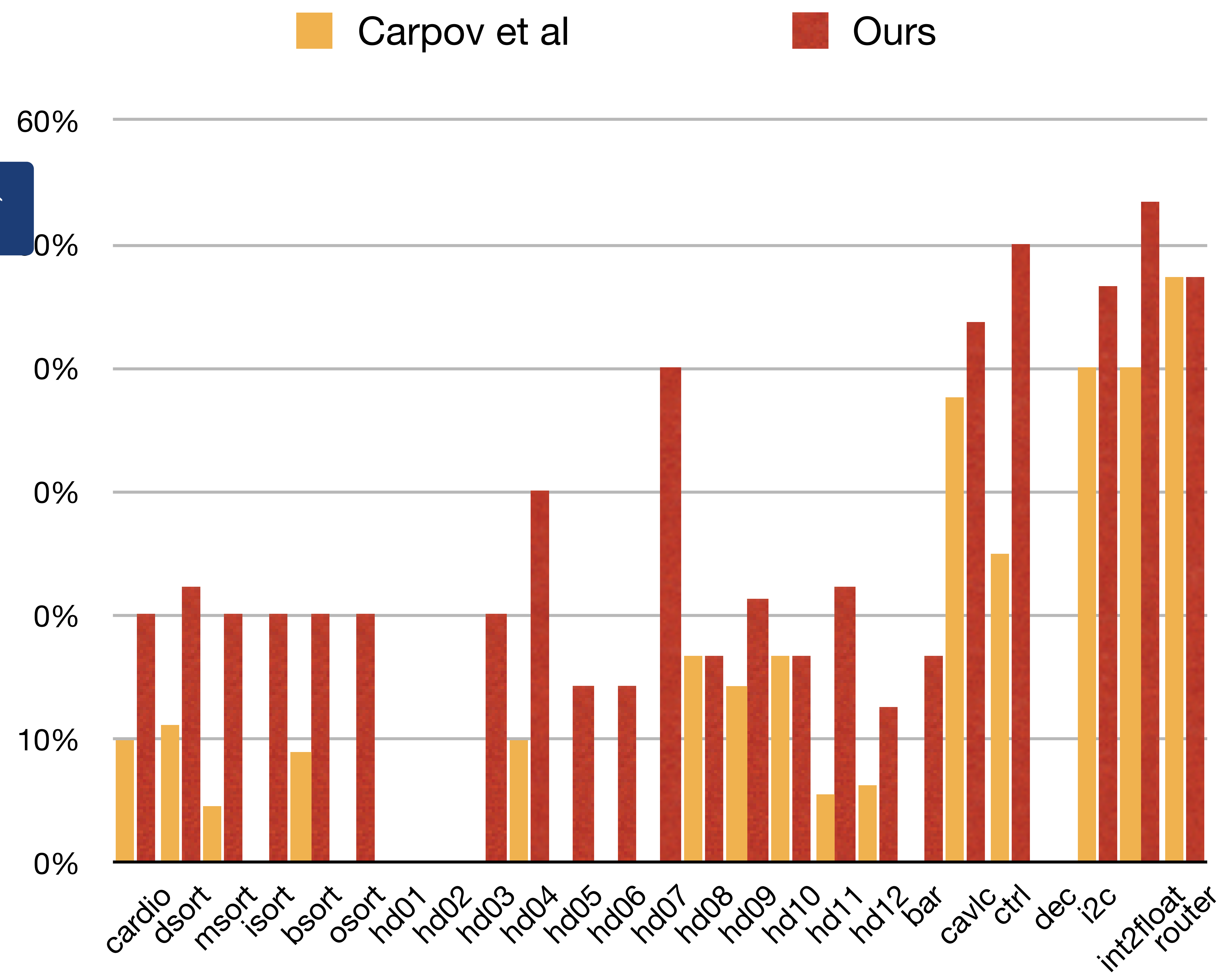
Depth Reduction

Lobster Performance (2/4)

Optimization Results of Lobster and the baseline



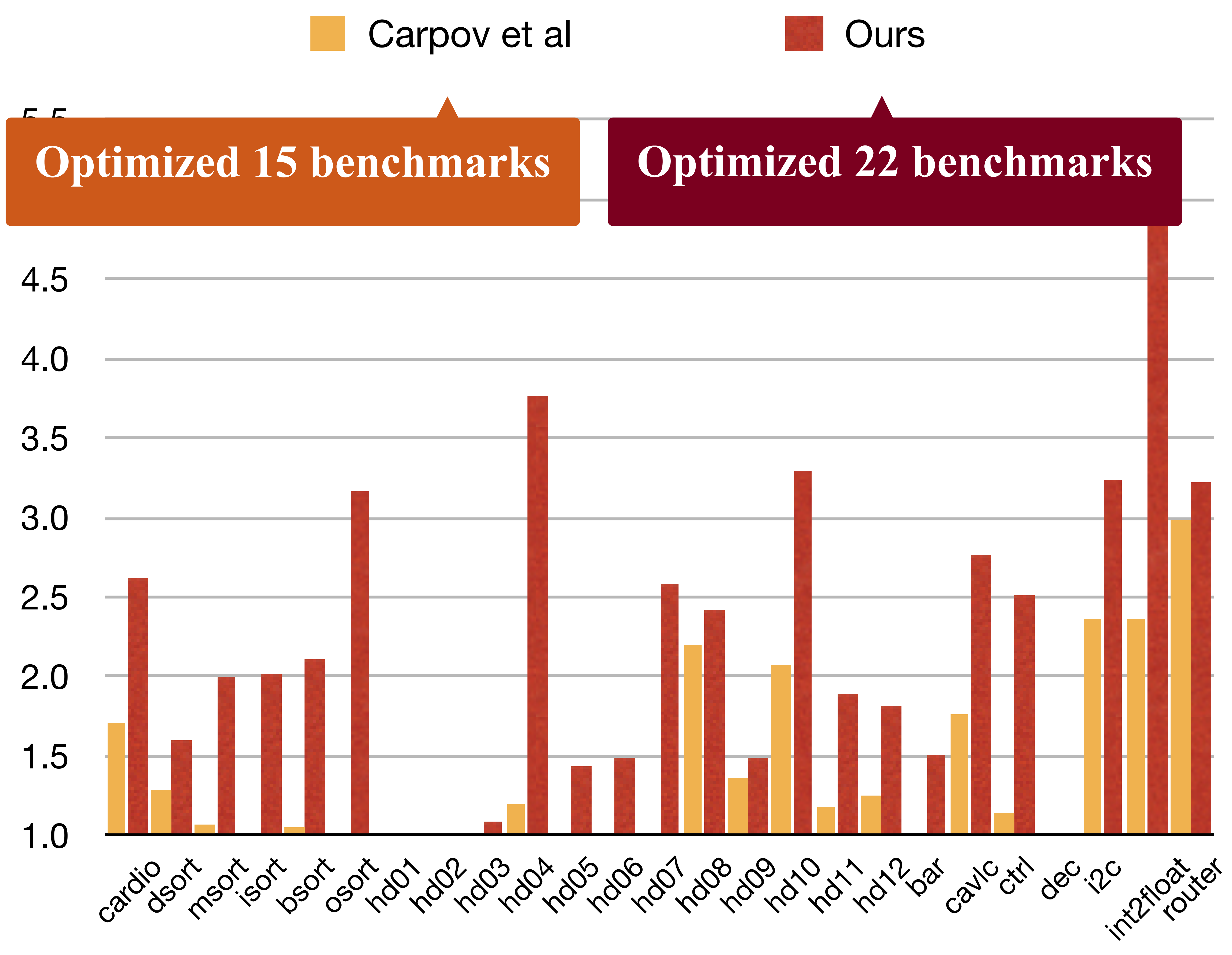
Speedup



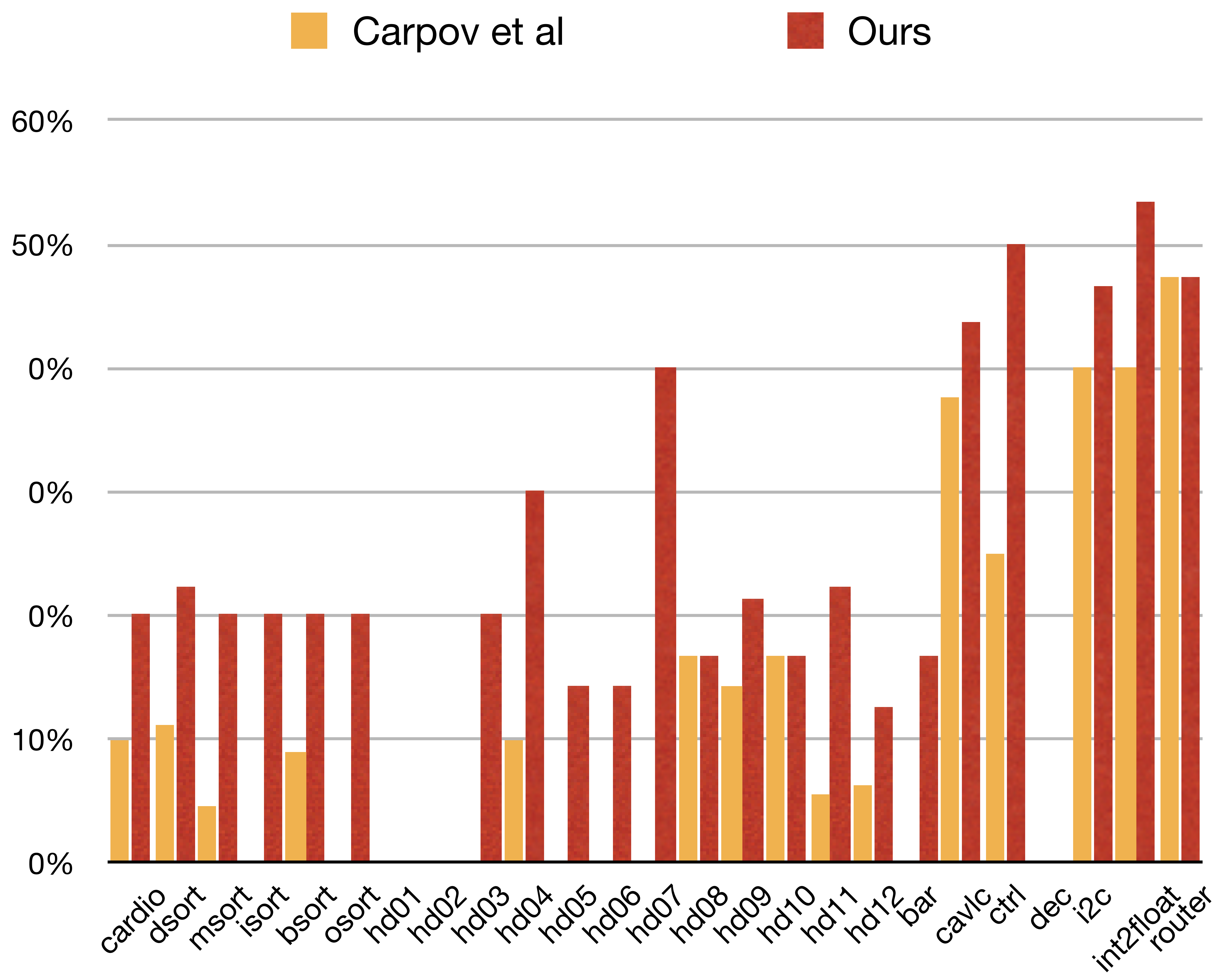
Depth Reduction

Lobster Performance (2/4)

Optimization Results of Lobster and the baseline



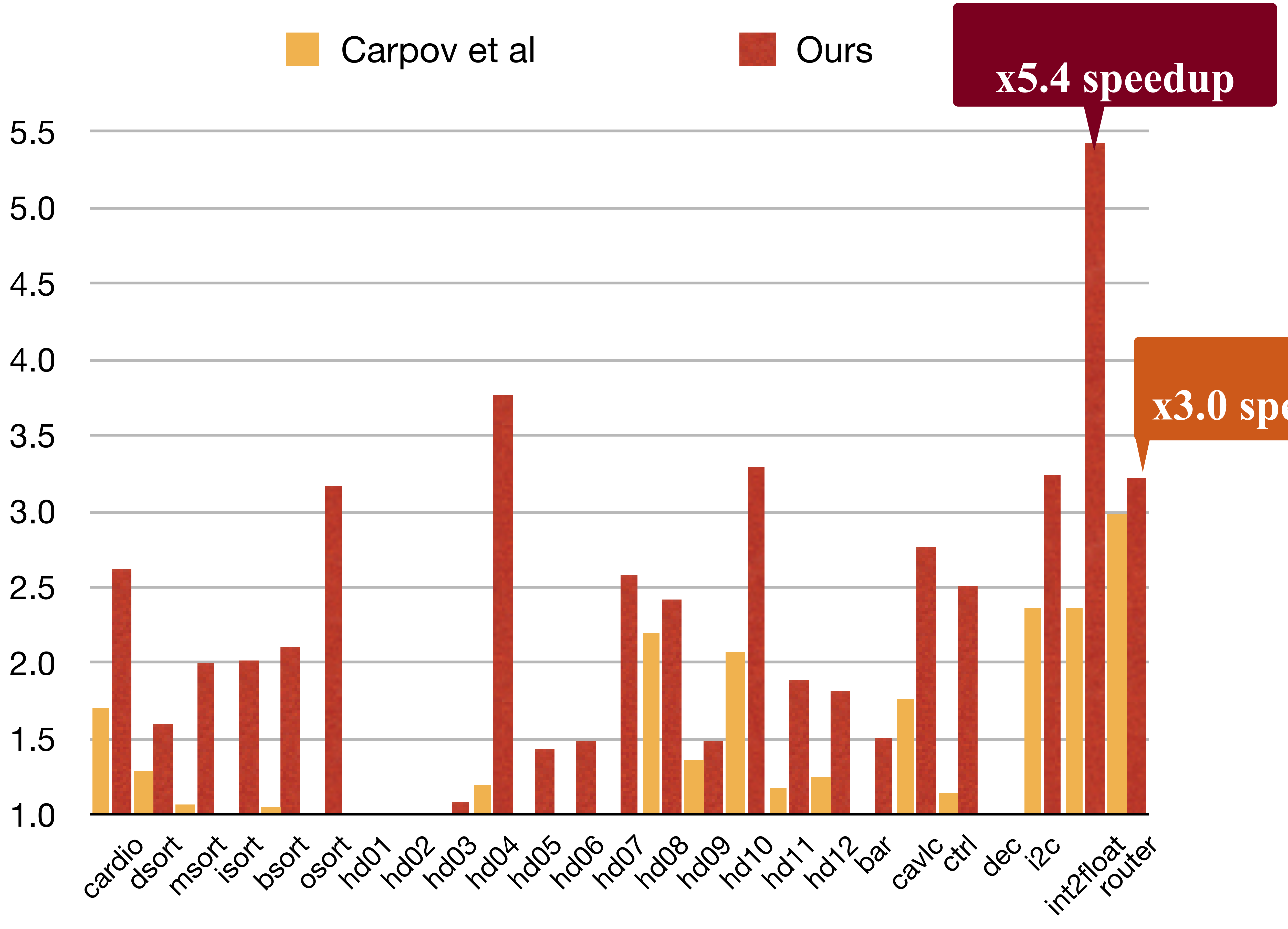
Speedup



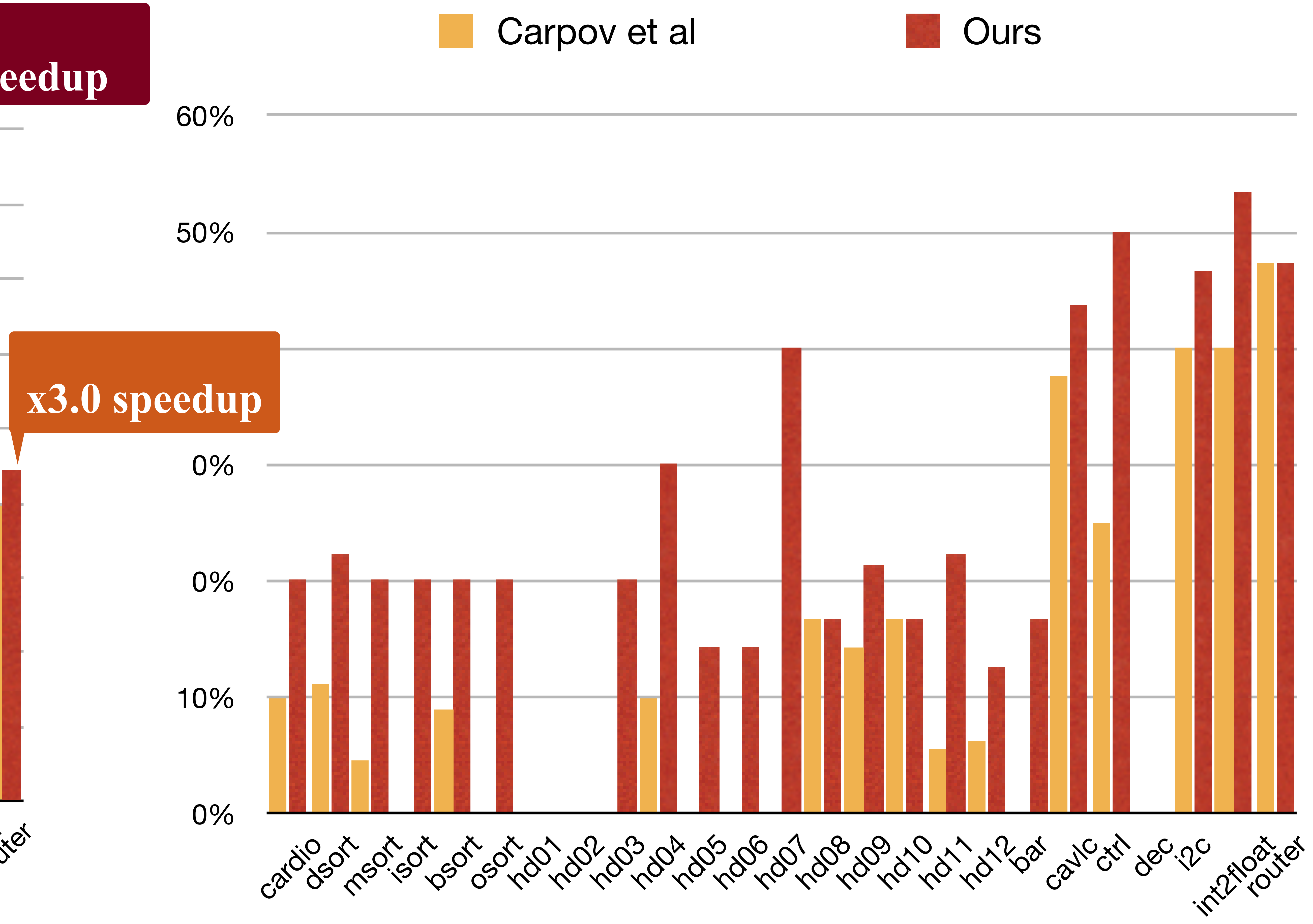
Depth Reduction

Lobster Performance (2/4)

Optimization Results of Lobster and the baseline



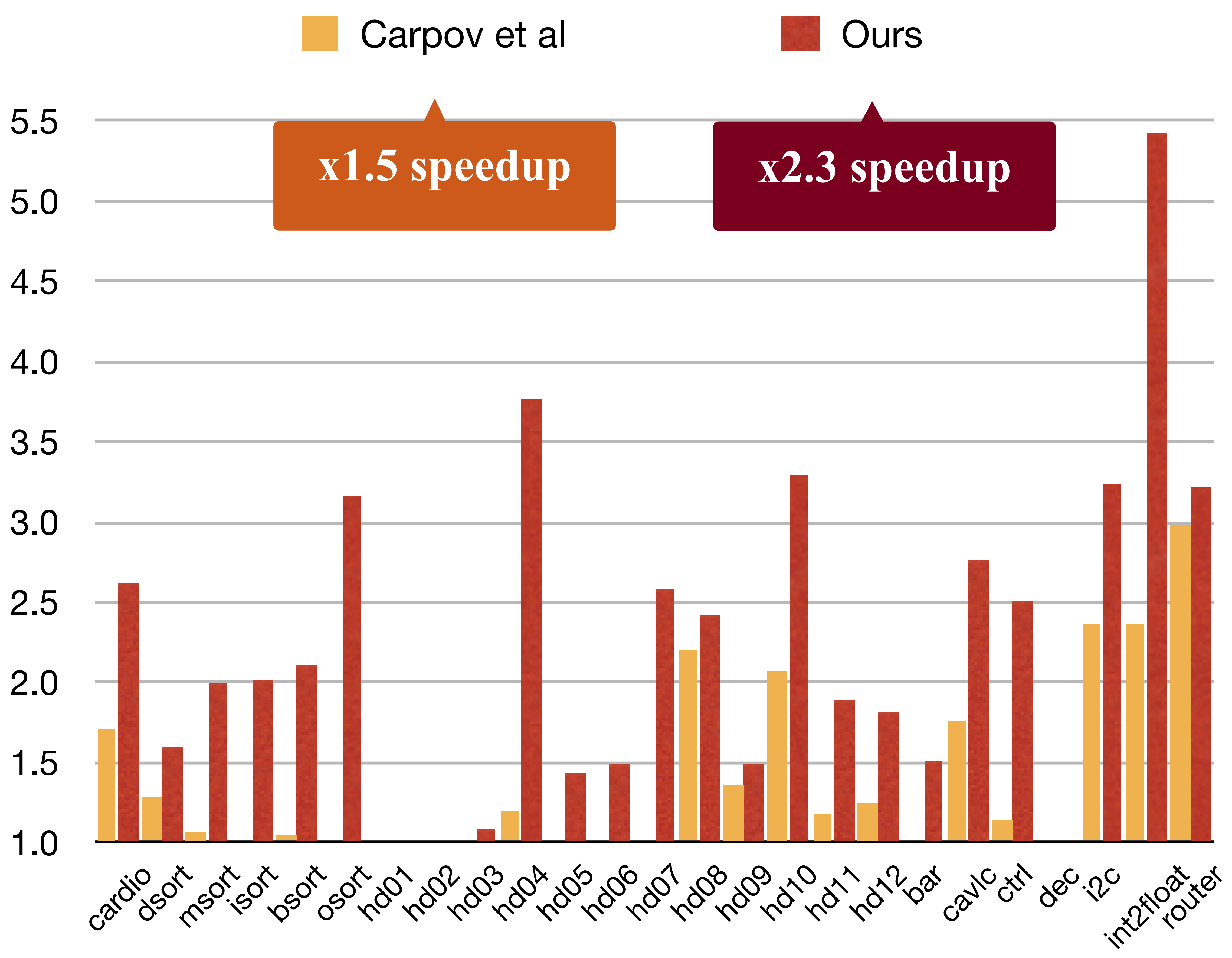
Speedup



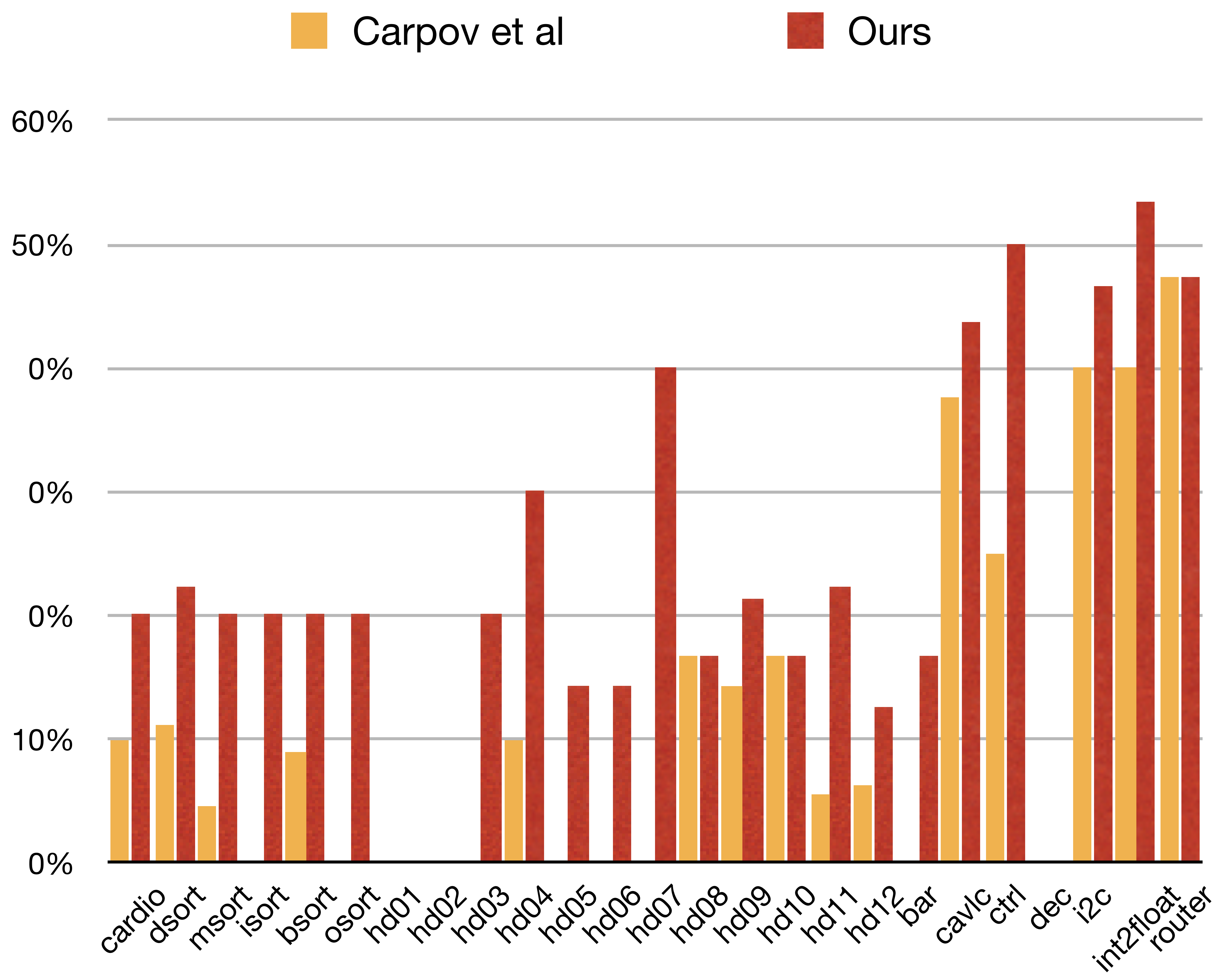
Depth Reduction

Lobster Performance (2/4)

Optimization Results of Lobster and the baseline



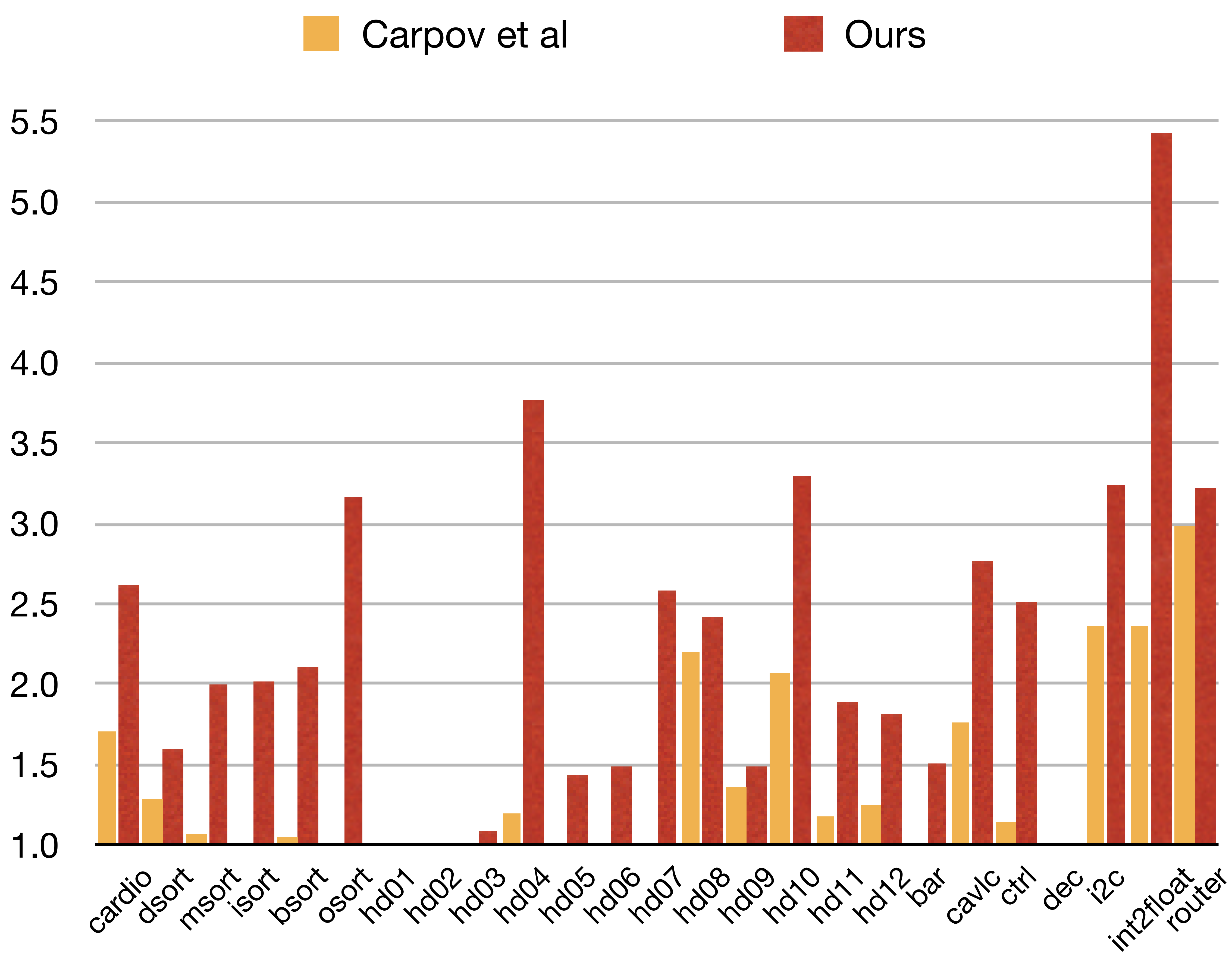
Speedup



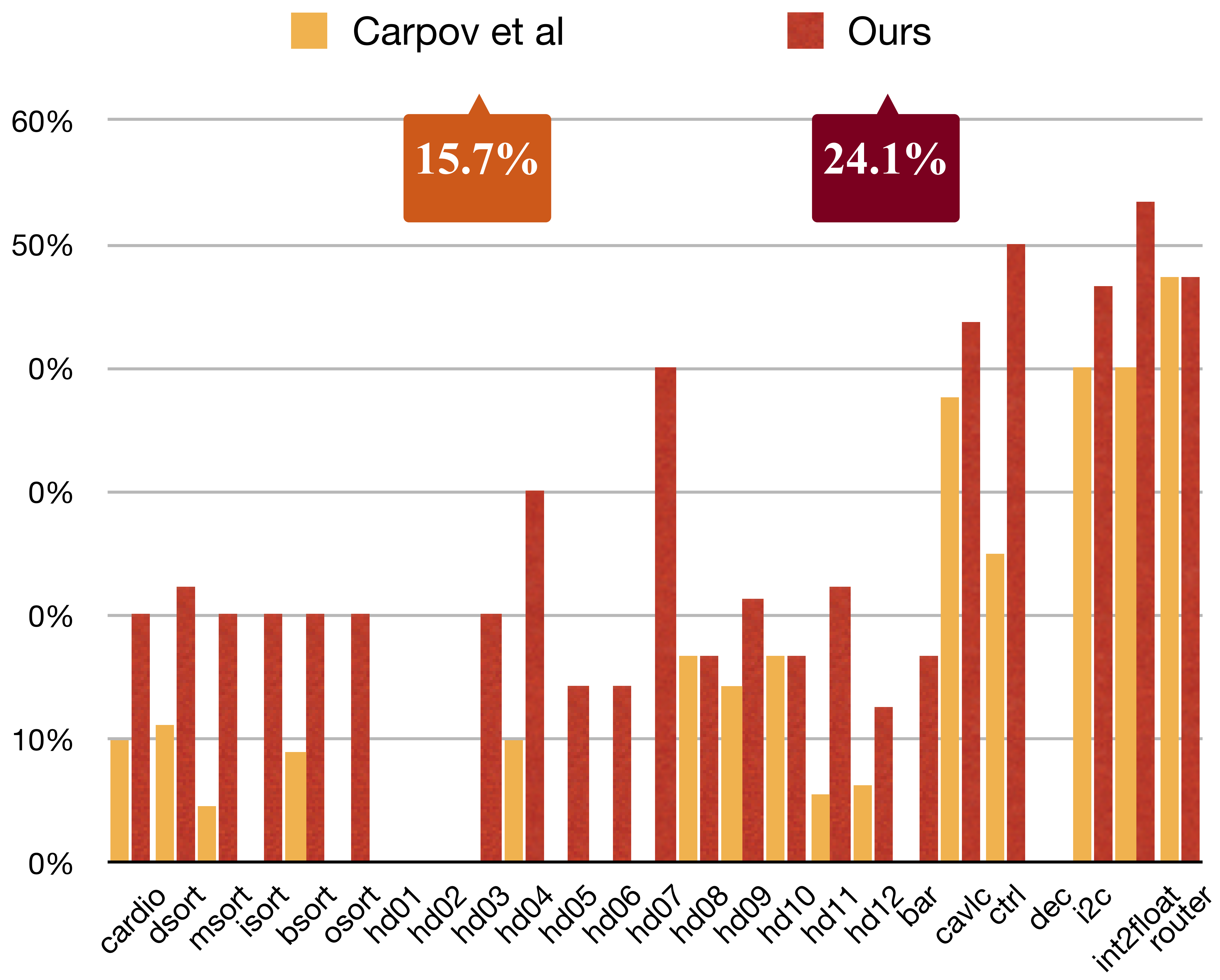
Depth Reduction

Lobster Performance (2/4)

Optimization Results of Lobster and the baseline



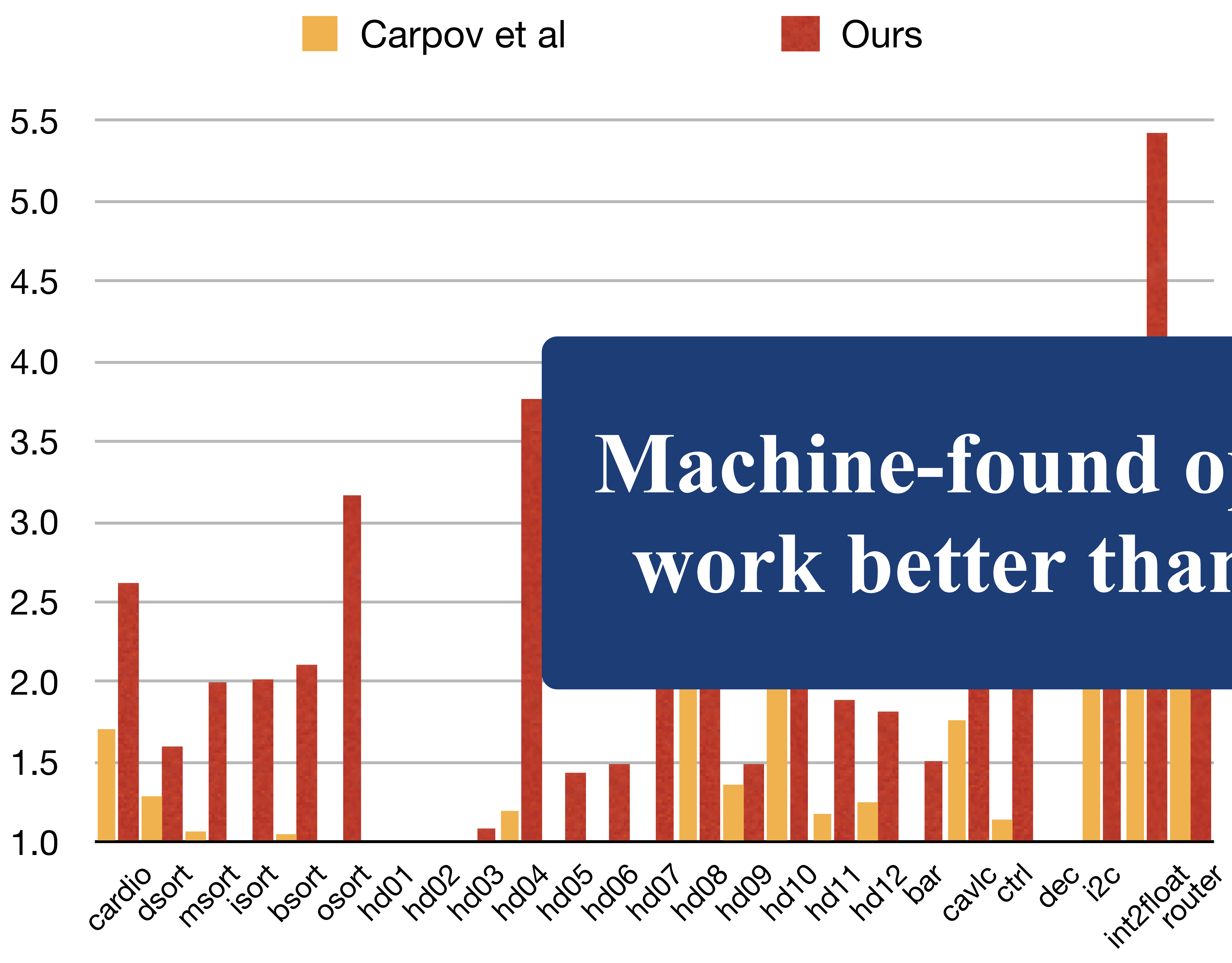
Speedup



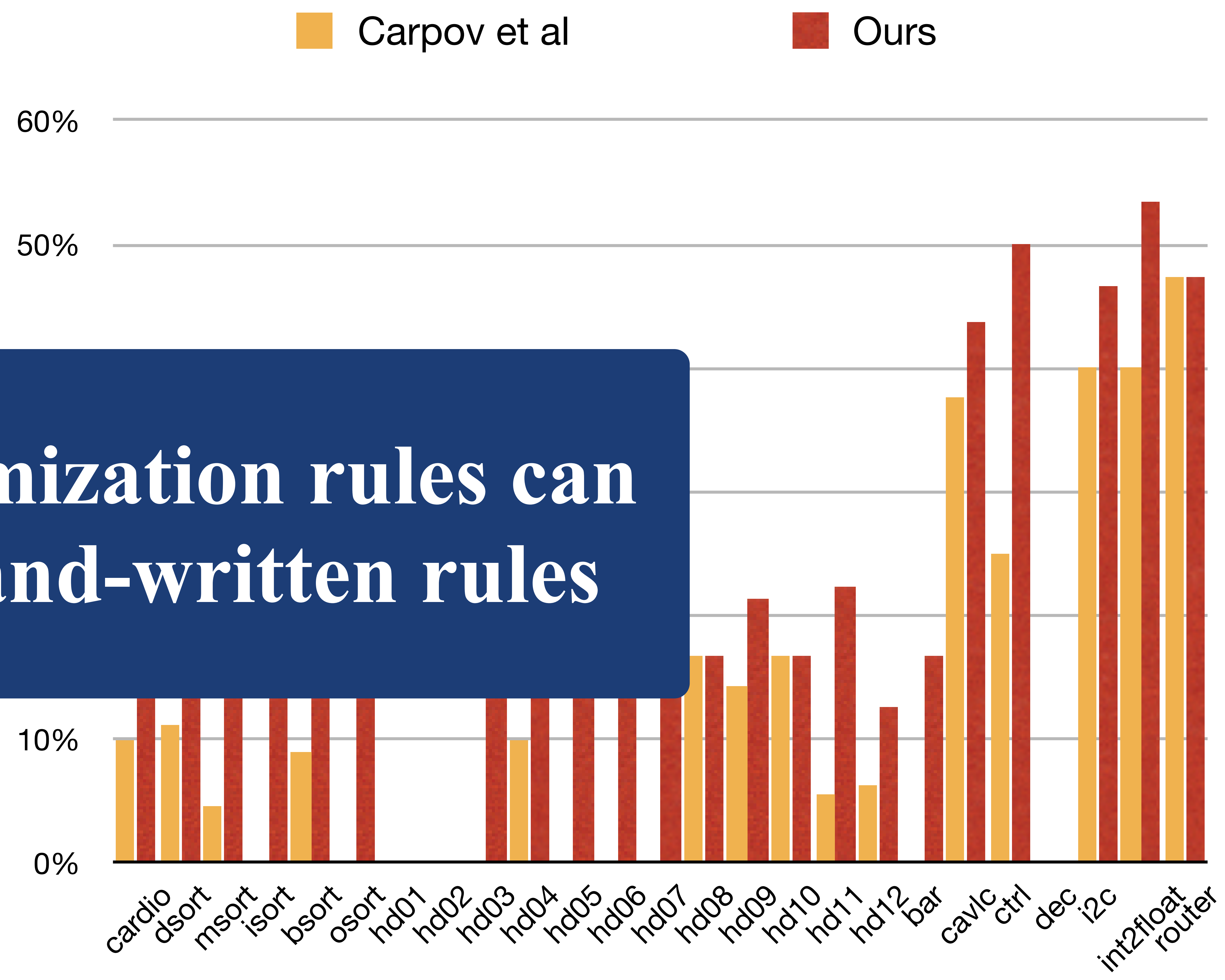
Depth Reduction

Lobster Performance (2/4)

Optimization Results of Lobster and the baseline



Speedup

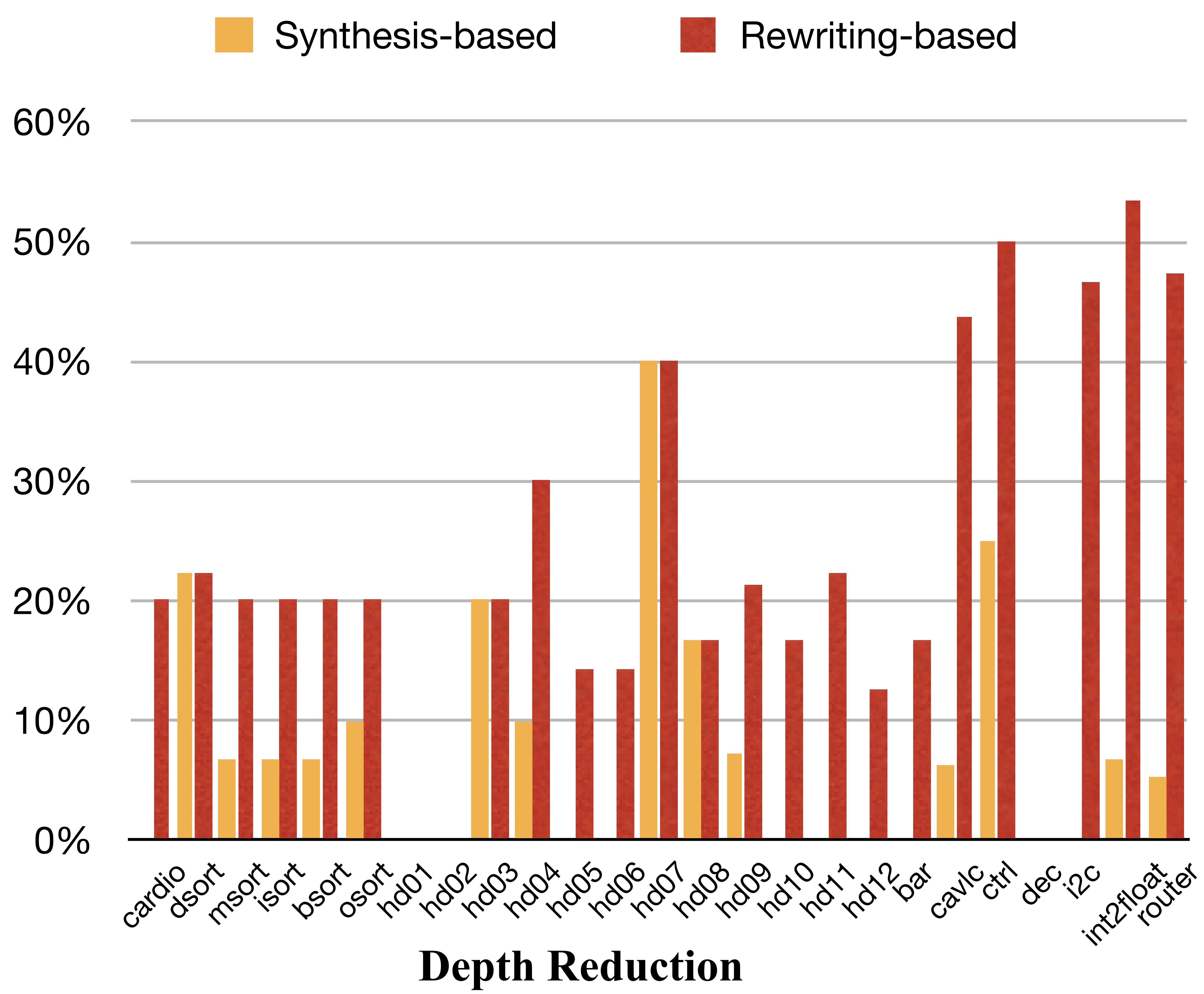


Depth Reduction

Machine-found optimization rules can work better than hand-written rules

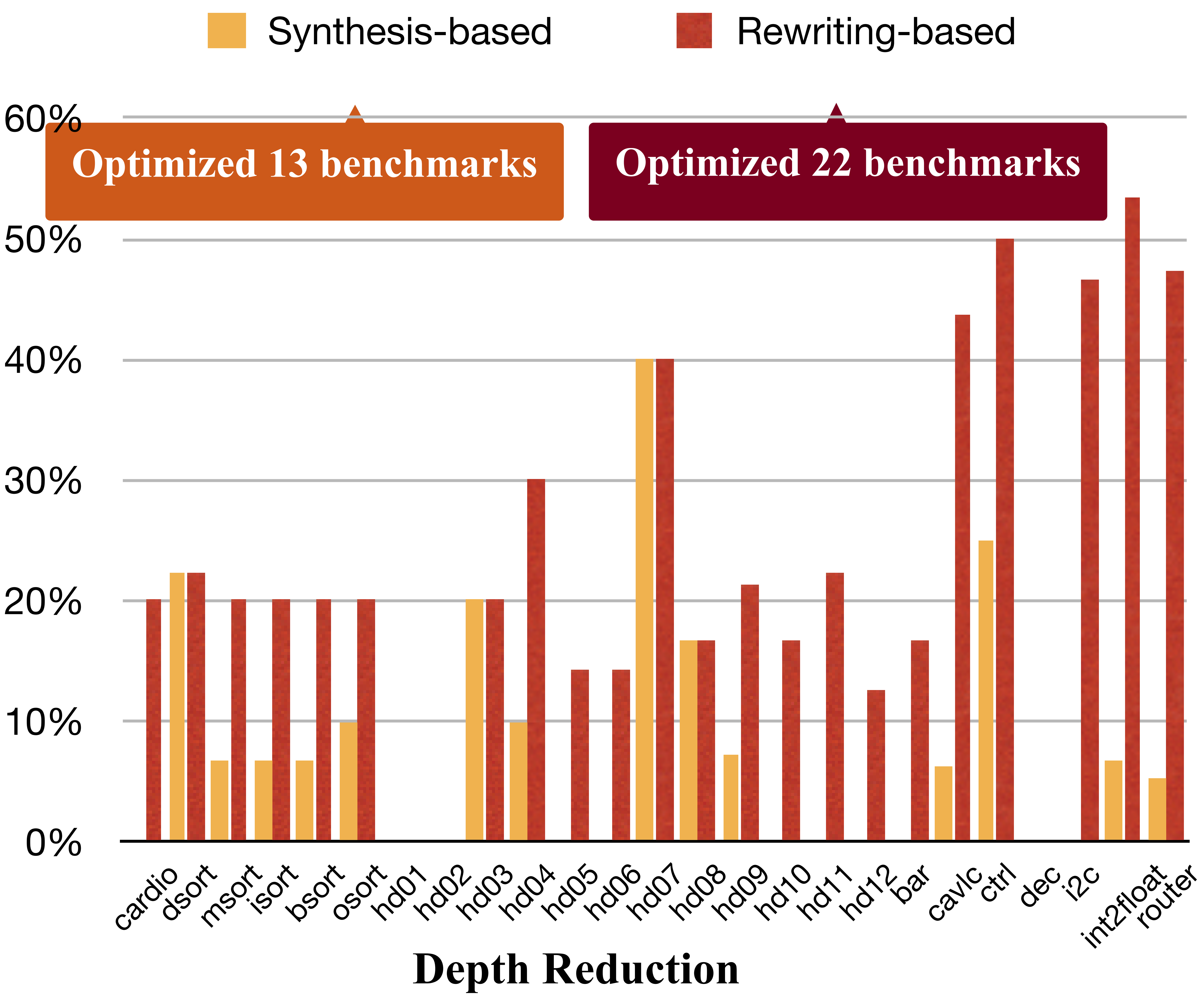
Lobster Performance (3/4)

Efficacy of Reusing Learned Optimization Patterns



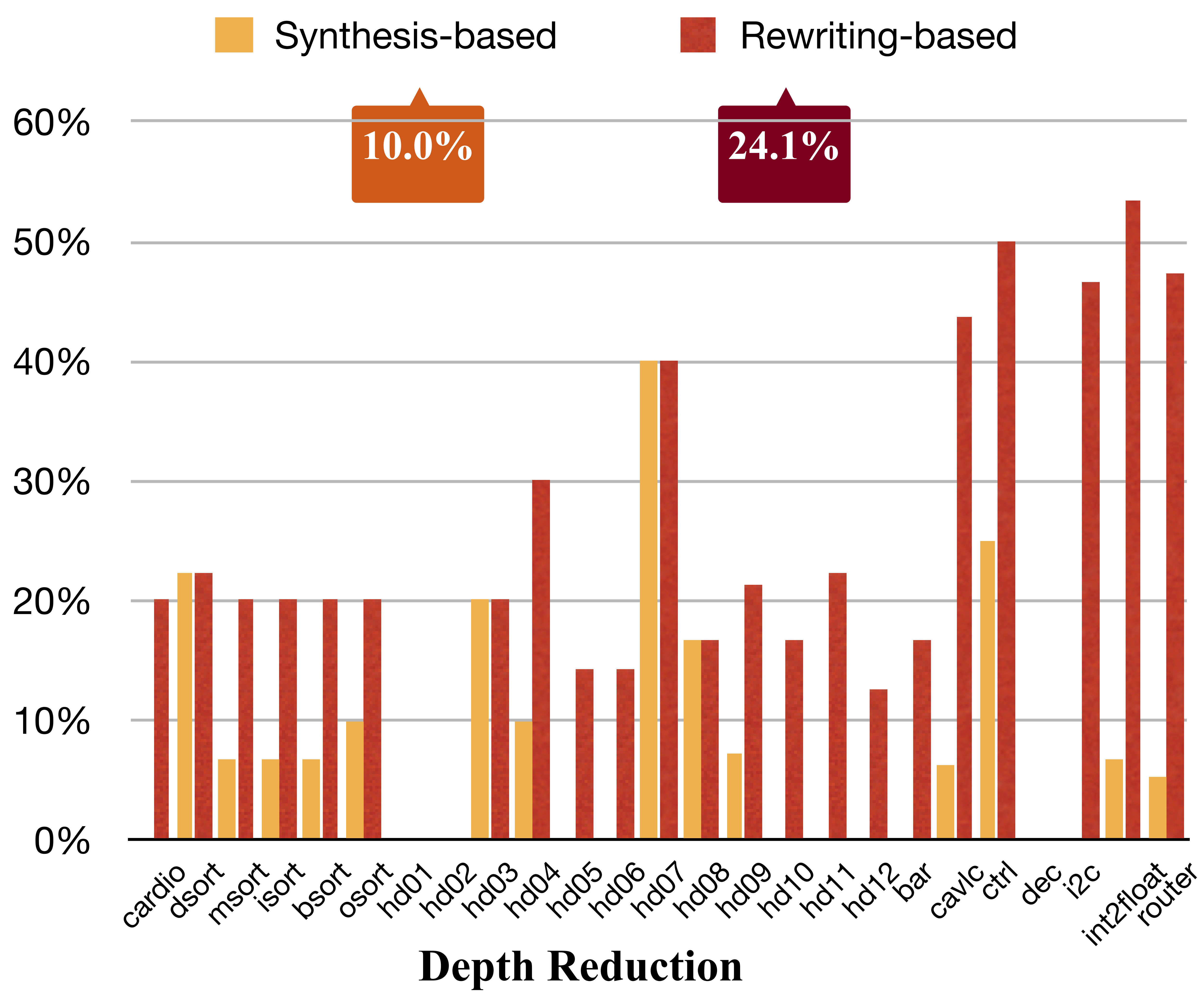
Lobster Performance (3/4)

Efficacy of Reusing Learned Optimization Patterns



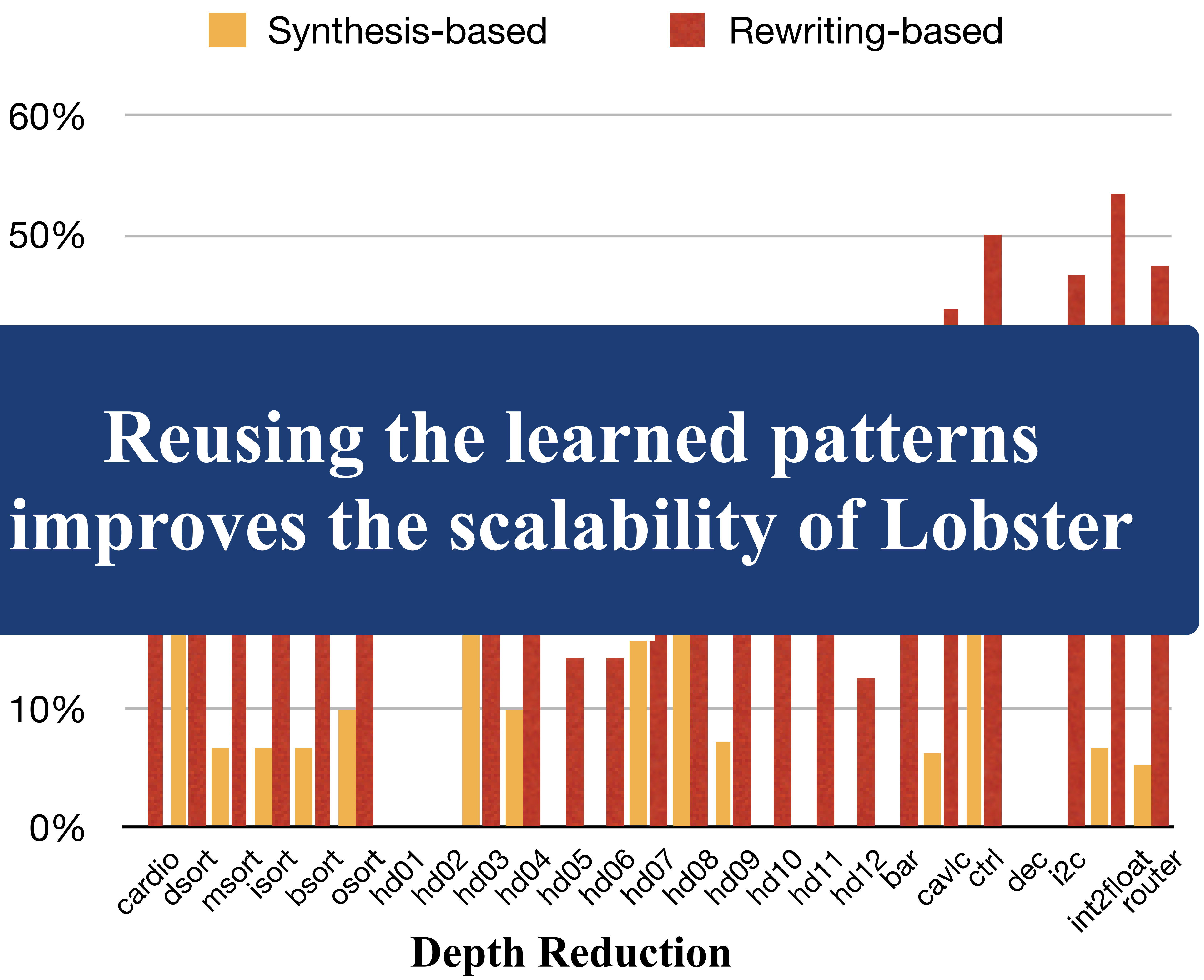
Lobster Performance (3/4)

Efficacy of Reusing Learned Optimization Patterns



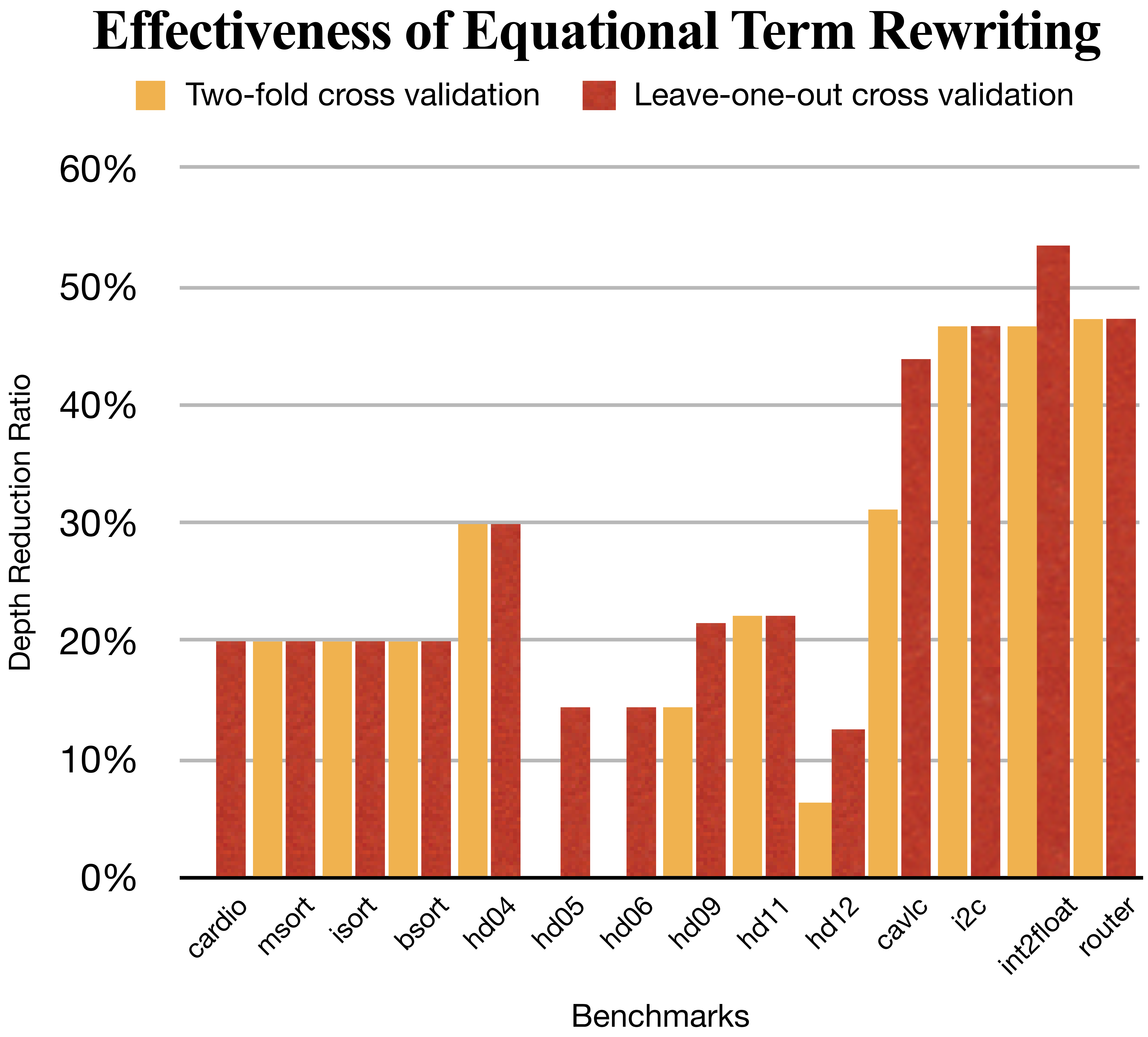
Lobster Performance (3/4)

Efficacy of Reusing Learned Optimization Patterns



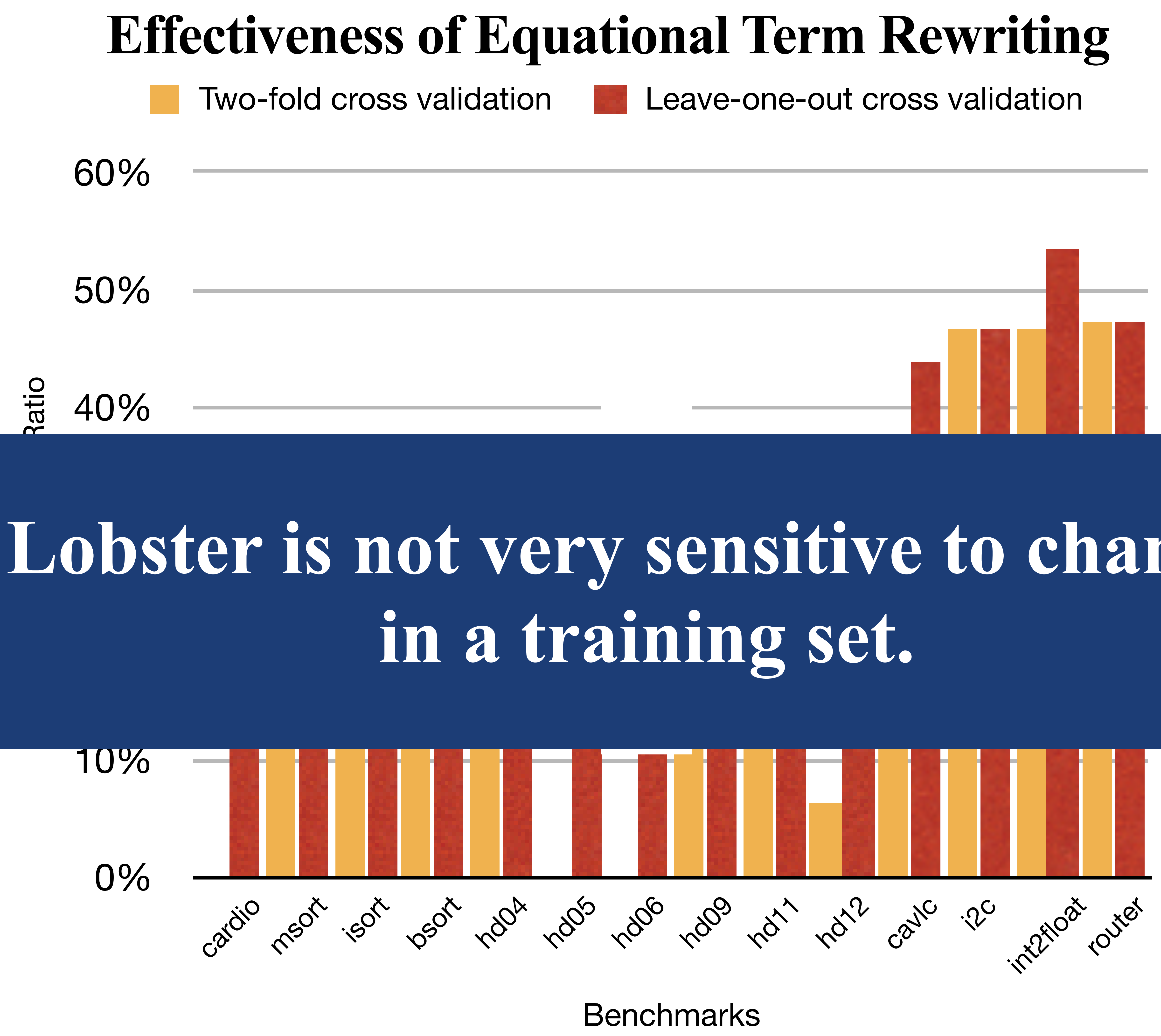
Reusing the learned patterns improves the scalability of Lobster

Lobster Performance (4/4)



Depth Reduction

Lobster Performance (4/4)



Depth Reduction

Lobster is not very sensitive to changes in a training set.

Related Work

- Hardware synthesis (e.g., ABC)
  - For decreasing circuit area and circuit depth (latency), not for multiplicative depth reduction
- General-purpose FHE compilers (e.g., Cingulata, Ramparts, Alchemy)
  - Optimization rules are hand-written, which requires manual efforts and often sub-optimal.
- Domain-specific FHE compilers (e.g., CHET)
  - Optimizations specialized for specific tasks (e.g., secure neural-network inference)
- Synthesis-based program optimization (e.g., STOKE, Optgen, Souper)
  - Optimization rules are also automatically learned, and applied via *syntactic* matching
  - We use *equational* matching to maximize generalization.

In the Paper…

- Detailed description of synthesis via localization
- Formalized Equational Term Rewriting
- Detailed description of experiment results

Thank you!