universität freiburg

Verifying Multipliers with Symbolic Computer Algebra by Order and Phase Optimization

University of Freiburg
Department of Computer Science
Alexander Konrad, Christoph Scholl
Freiburg, 05. September 2024

Verification of Arithmetic Circuits

Motivation

- Circuit design containing arithmetic not only by processor vendors, but also by suppliers of special-purpose hardware
- **Fully automatic formal verification** of arithmetic circuits needed
- Methods based on BDDs or SAT usually fail for multipliers
- **Great progress** for gate-level multipliers during last years based on **Symbolic Computer Algebra**

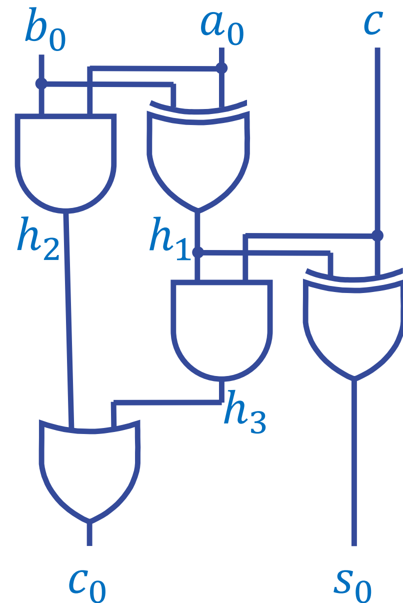
Verification of Arithmetic Circuits
Symbolic Computer Algebra (SCA)

- Symbolic Computer Algebra to verify **integer arithmetic**:
  - Exposition can be **simplified** by considering replacements of variables by gate polynomials („backward rewriting“)
  - Based on the fact that the **polynomial** for a pseudo-Boolean function $\boldsymbol{f}: \{\mathbf{0}, \mathbf{1}\}^{\boldsymbol{n}} \rightarrow \boldsymbol{Z}$ **is unique** (up to reordering of terms)
  - Illustrated by a simple example: …

Verification of Arithmetic Circuits

Symbolic Computer Algebra – Example of Backward Rewriting

- **Full-Adder with specification $2c_0 + s_0 = a_0 + b_0 + c$:**

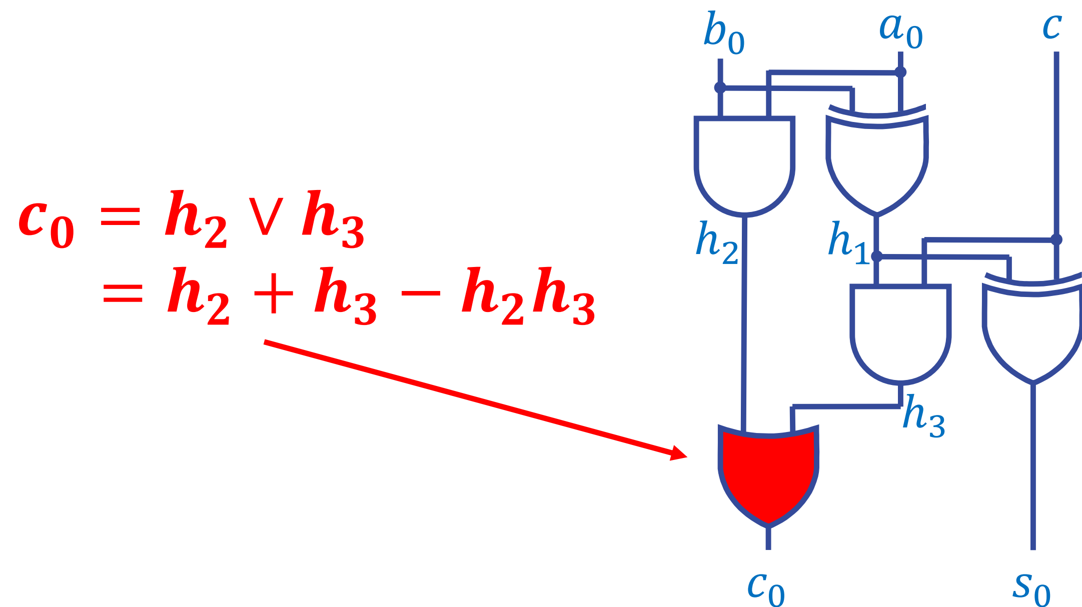


- **Start with output word $2c_0 + s_0$**

Verification of Arithmetic Circuits

Symbolic Computer Algebra – Example of Backward Rewriting

- **Full-Adder with specification $2c_0 + s_0 = a_0 + b_0 + c$:**



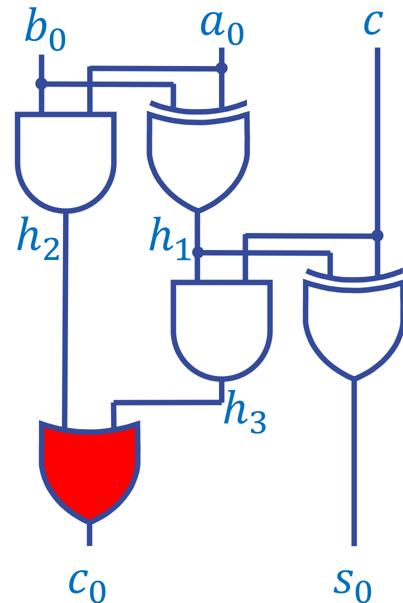
- **Start with output word $2c_0 + s_0$**

$\Rightarrow 2h_2 + 2h_3 - 2h_2h_3 + s_0$

Verification of Arithmetic Circuits

Symbolic Computer Algebra – Example of Backward Rewriting

- **Full-Adder with specification $2c_0 + s_0 = a_0 + b_0 + c$:**



- **Start with output word $2c_0 + s_0$**

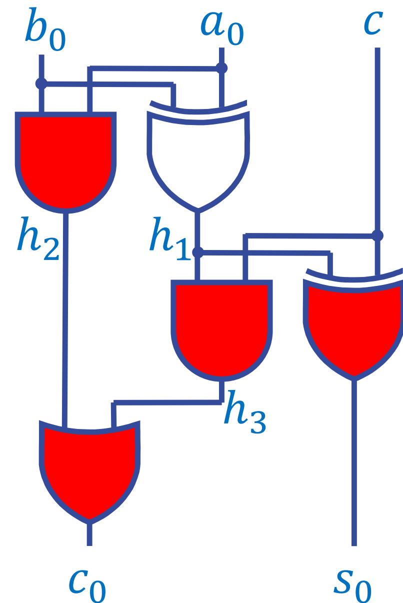
$\Rightarrow 2h_2 + 2h_3 - 2h_2h_3 + s_0$

$\Rightarrow \ldots$

Verification of Arithmetic Circuits

Symbolic Computer Algebra – Example of Backward Rewriting

- **Full-Adder with specification $2c_0 + s_0 = a_0 + b_0 + c$:**



- **Start with output word $2c_0 + s_0$**

$\Rightarrow 2h_2 + 2h_3 - 2h_2h_3 + s_0$

$\Rightarrow 2h_2 + 2h_3 - 2h_2h_3 + s_0$

$\Rightarrow 2h_2 + 2ch_1 - 2ch_1h_2 + s_0$

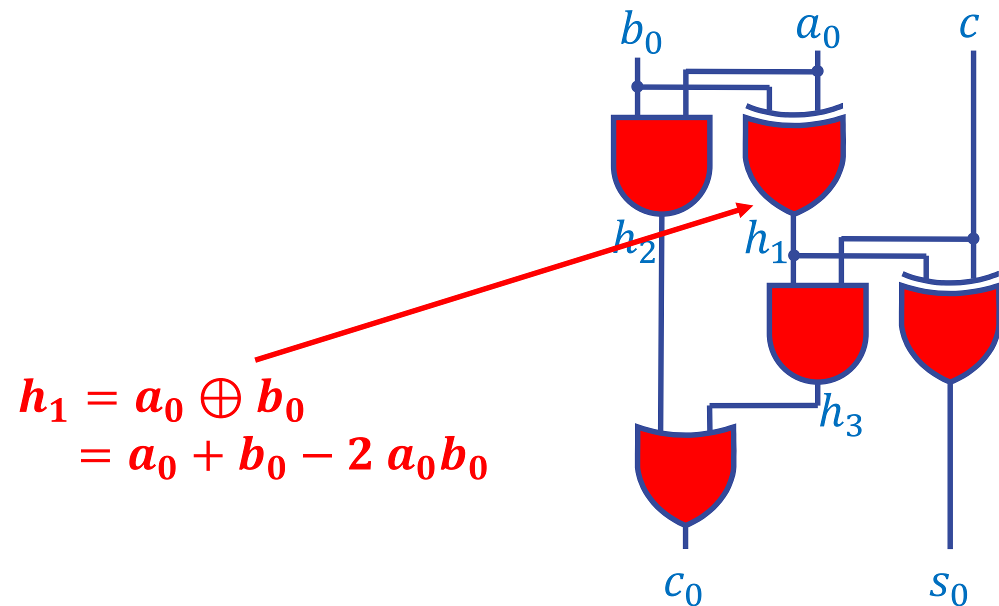
$\Rightarrow 2h_2 - 2ch_1h_2 + c + h_1$

$\Rightarrow 2a_0b_0 - 2a_0b_0ch_1 + c + h_1$

Verification of Arithmetic Circuits

Symbolic Computer Algebra – Example of Backward Rewriting

- **Full-Adder with specification** $2c_0 + s_0 = a_0 + b_0 + c$:



- **Start with output word** $2c_0 + s_0$

$\Rightarrow 2h_2 + 2h_3 - 2h_2h_3 + s_0$

$\Rightarrow 2h_2 + 2h_3 - 2h_2h_3 + s_0$

$\Rightarrow 2h_2 + 2ch_1 - 2ch_1h_2 + s_0$

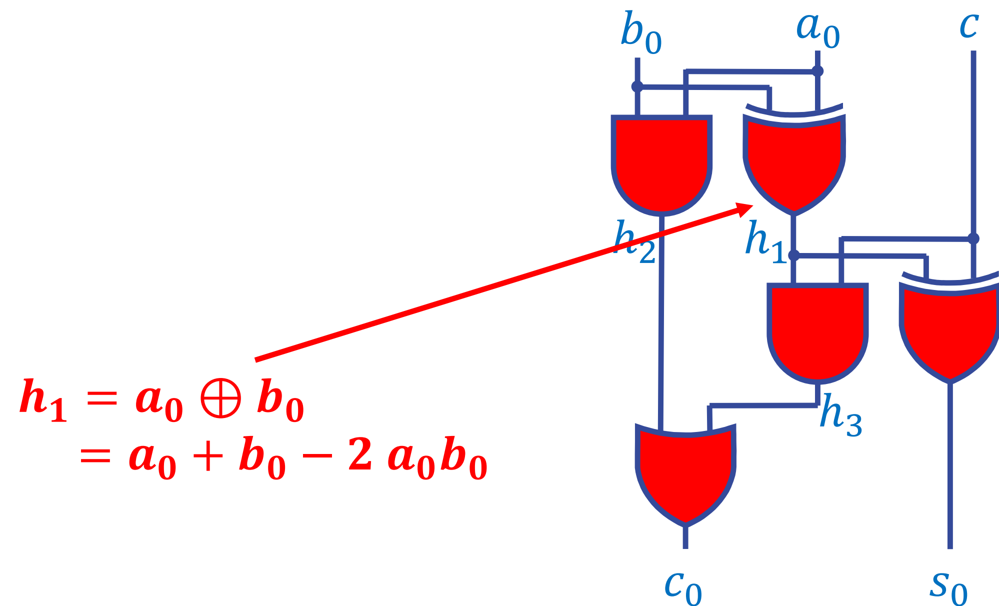
$\Rightarrow 2h_2 - 2ch_1h_2 + c + h_1$

$\Rightarrow 2a_0b_0 - 2a_0b_0ch_1 + c + h_1$

Verification of Arithmetic Circuits

Symbolic Computer Algebra – Example of Backward Rewriting

- **Full-Adder with specification** $2c_0 + s_0 = a_0 + b_0 + c$:



- **Start with output word** $2c_0 + s_0$

$\Rightarrow 2h_2 + 2h_3 - 2h_2h_3 + s_0$

$\Rightarrow 2h_2 + 2h_3 - 2h_2h_3 + s_0$

$\Rightarrow 2h_2 + 2ch_1 - 2ch_1h_2 + s_0$

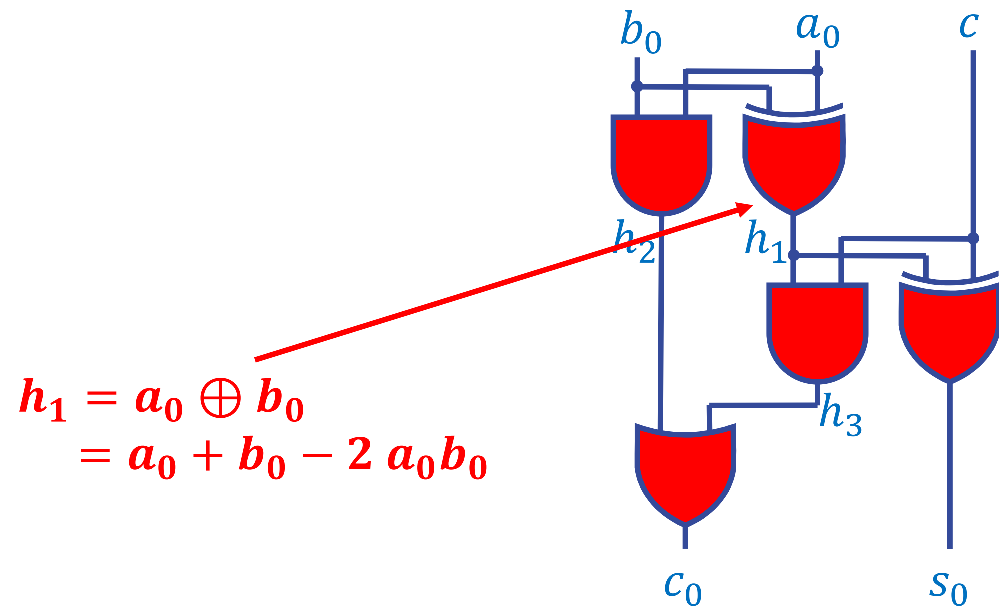
$\Rightarrow 2h_2 - 2ch_1h_2 + c + h_1$

$\Rightarrow 2a_0b_0 - 2a_0b_0ch_1 + c + h_1$

$\Rightarrow 2a_0b_0 - 2a_0^2b_0c - 2a_0b_0^2c$
$+4a_0^2b_0^2c + c + a_0 + b_0 - 2a_0b_0$

Symbolic Computer Algebra – Example of Backward Rewriting

- Full-Adder with specification $2c_0 + s_0 = a_0 + b_0 + c$:



- Start with output word $2c_0 + s_0$

$\Rightarrow 2h_2 + 2h_3 - 2h_2h_3 + s_0$

$\Rightarrow 2h_2 + 2h_3 - 2h_2h_3 + s_0$

$\Rightarrow 2h_2 + 2ch_1 - 2ch_1h_2 + s_0$

$\Rightarrow 2h_2 - 2ch_1h_2 + c + h_1$

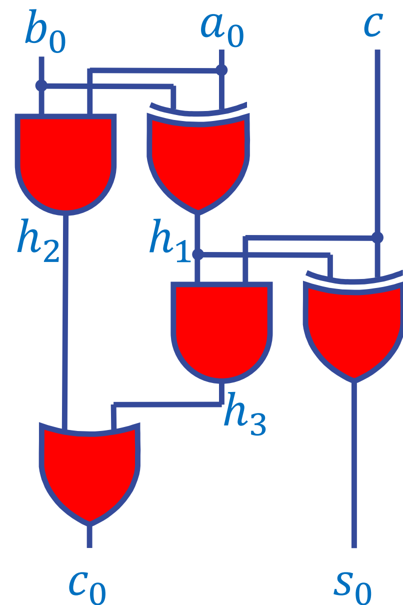
$\Rightarrow 2a_0b_0 - 2a_0b_0ch_1 + c + h_1$

$\Rightarrow a_0 + b_0 + c$

Verification of Arithmetic Circuits

Symbolic Computer Algebra – Example of Backward Rewriting

- **Full-Adder with specification $2c_0 + s_0 = a_0 + b_0 + c$:**



- **Alternatively: Start with**

$$2c_0 + s_0 - a_0 - b_0 - c$$

$\Rightarrow$ …

$\Rightarrow$ …

$\Rightarrow$ …

$\Rightarrow$ …

$\Rightarrow a_0 + b_0 + c - a_0 - b_0 - c = 0$

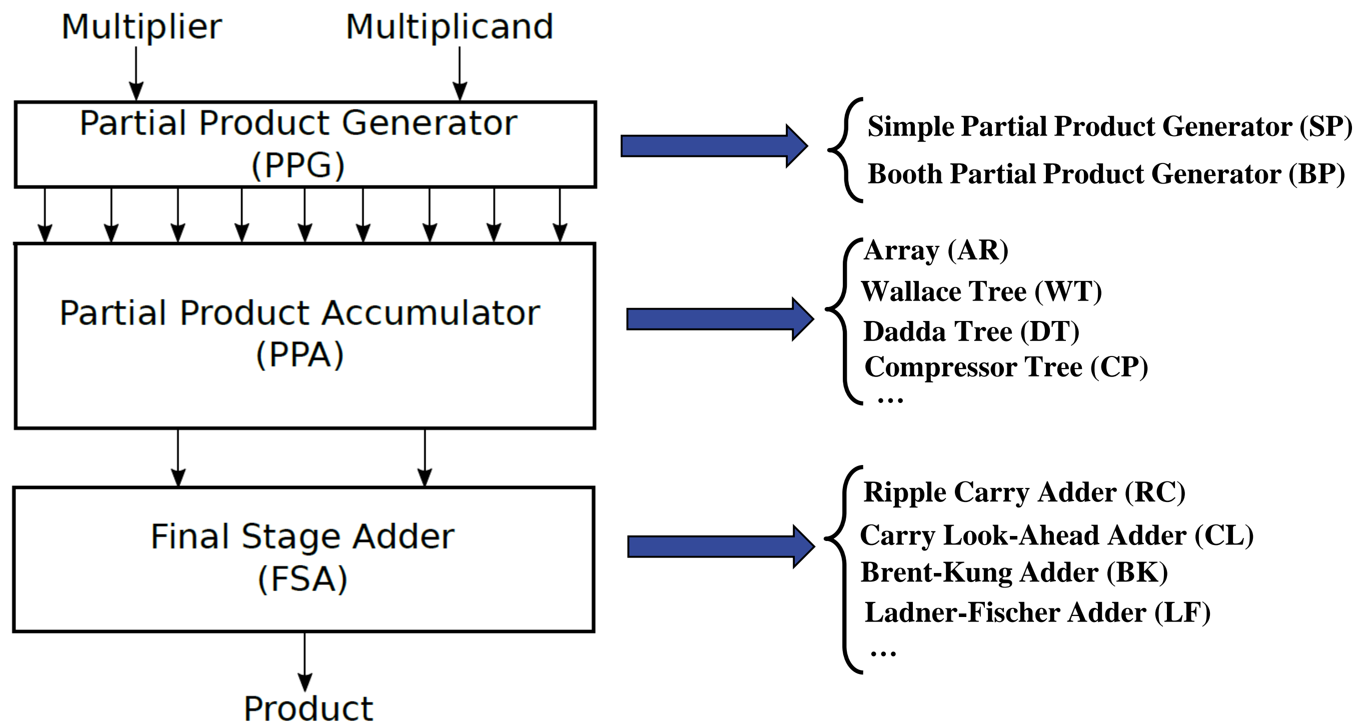
Verification of Arithmetic Circuits

Unsigned Integer Multiplication

- n-bit unsigned multiplier is specified as:
  - Inputs $(a_{n-1}a_{n-2}\dots a_0)$ and $(b_{n-1}b_{n-2}\dots b_0)$
  - Output $(p_{2n-1}p_{2n-2}\dots p_0)$
  - Specification polynomial $SP = \left(\sum_{i=0}^{2n-1} p_i\, 2^i - \left(\sum_{j=0}^{n-1} a_j\, 2^j\right) \times \left(\sum_{k=0}^{n-1} b_k\, 2^k\right)\right) \bmod 2^{2n}$
- **Multiplier is correct iff backward rewriting reduces SP to 0.**

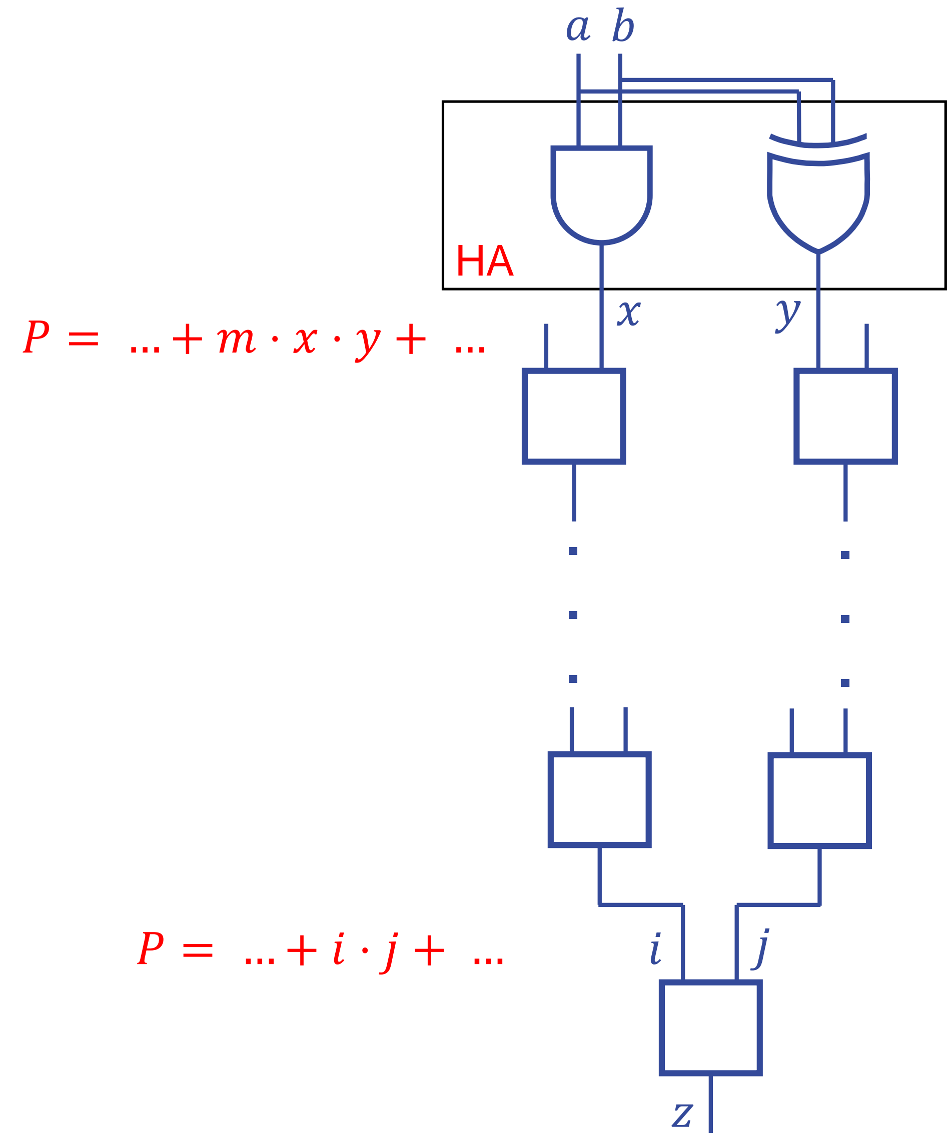
Verification of Arithmetic Circuits
Unsigned Integer Multiplication

- Three-stage multiplier structure:



Previous SCA-based Approaches
Removing Vanishing Monomials

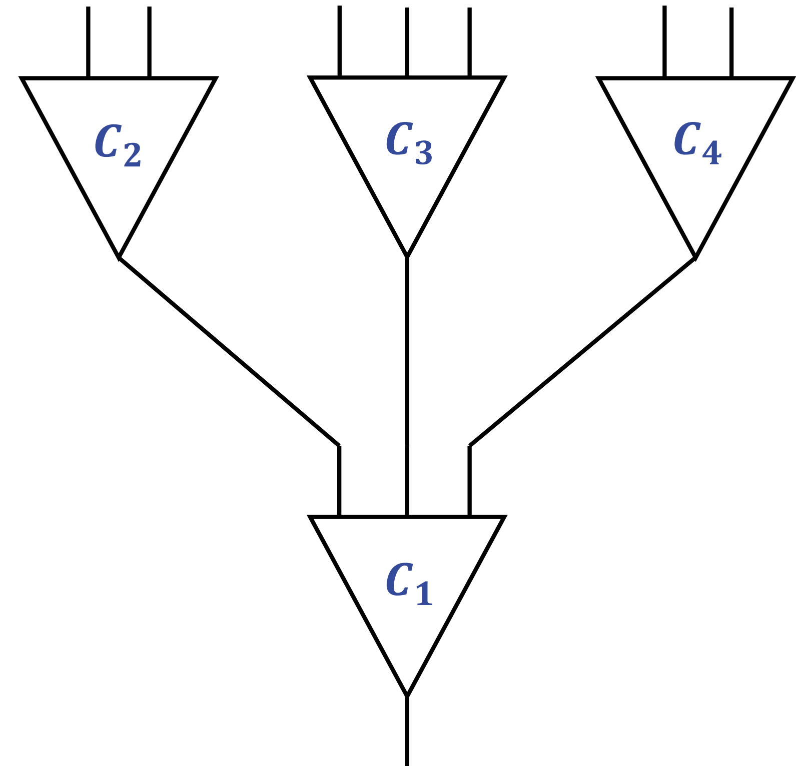
- [Sayed-Ahmed et al., DATE‘16] discovered ***vanishing monomials*** as a major problem in backward rewriting
- further developed into several tools:
  - RevSCA [Mahzoon et al., DAC‘19]
  - DyPoSub [Mahzool et al., DATE‘20]
  - RevSCA-2.0 [Mahzoon et al., TCAD‘22]



Previous SCA-based Approaches

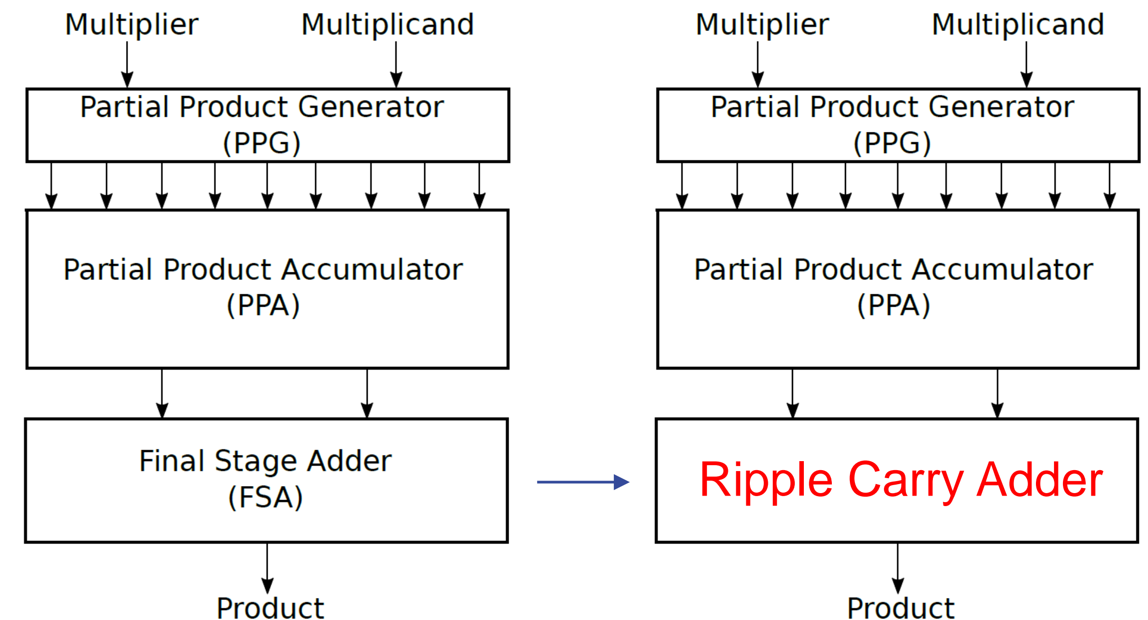
Removing Vanishing Monomials + Dynamic Ordering

- [Sayed-Ahmed et al., DATE‘16] discovered ***vanishing monomials*** as a major problem in backward rewriting
- further developed into several tools:
  - RevSCA [Mahzoon et al., DAC‘19]
  - DyPoSub [Mahzool et al., DATE‘20]
  - RevSCA-2.0 [Mahzoon et al., TCAD‘22]
- DyPoSub **first** to use **dynamic ordering** approach



Previous SCA-based Approaches
Simplifying FSA stage

- [Kaufmann et al., FMCAD‘19] proposed to **simplify multiplier** structure …
- … by **substituting FSA stage**
  - AMulet [Kaufmann et al., FMCAD‘19]
  - AMulet 2 [Kaufmann, Biere, TACAS‘19]
  - AMulet 2.2 [Kaufmann, Biere, TAP‘22]
- … by using **Dual Variables Encoding**
  - TeluMA [Kaufmann et al., DATE‘22]



Our new approach

Dynamic Phase and Order Optimization

- Various interesting approaches proposed
- But **rely on detecting certain structures → robustness in question**
- We claim:

  better (and more robust) results with **two, simple dynamic approaches**:
  - **Dynamic Phase Optimization**
  - **Dynamic Order Optimization**

Our new approach
Dynamic Phase Optimization

- Simplification of Dual Variables Encoding in TeluMA [Kaufmann et al., DATE'22]
- Occuring **literals either all positive or all negative**, no mix-up of phases
- **Dynamic**:
  - Start with positive phases for all variables
  - Phase Optimization in current polynomial while backward rewriting
  - Decision based on simple greedy heuristic: polynomial size
- Idea: make backward rewriting more robust against different traversal orders

Our new approach
Dynamic Phase Optimization Example

- Consider the function $f(x_1, x_2, x_3) = x_1 \lor x_2 \lor x_3$
- with polynomial representation $P = x_1 + x_2 + x_3 - x_1 x_2 - x_1 x_3 - x_2 x_3 + x_1 x_2 x_3$

- Change phase of x_1 by replacing it with $1 - \overline{x_1}$

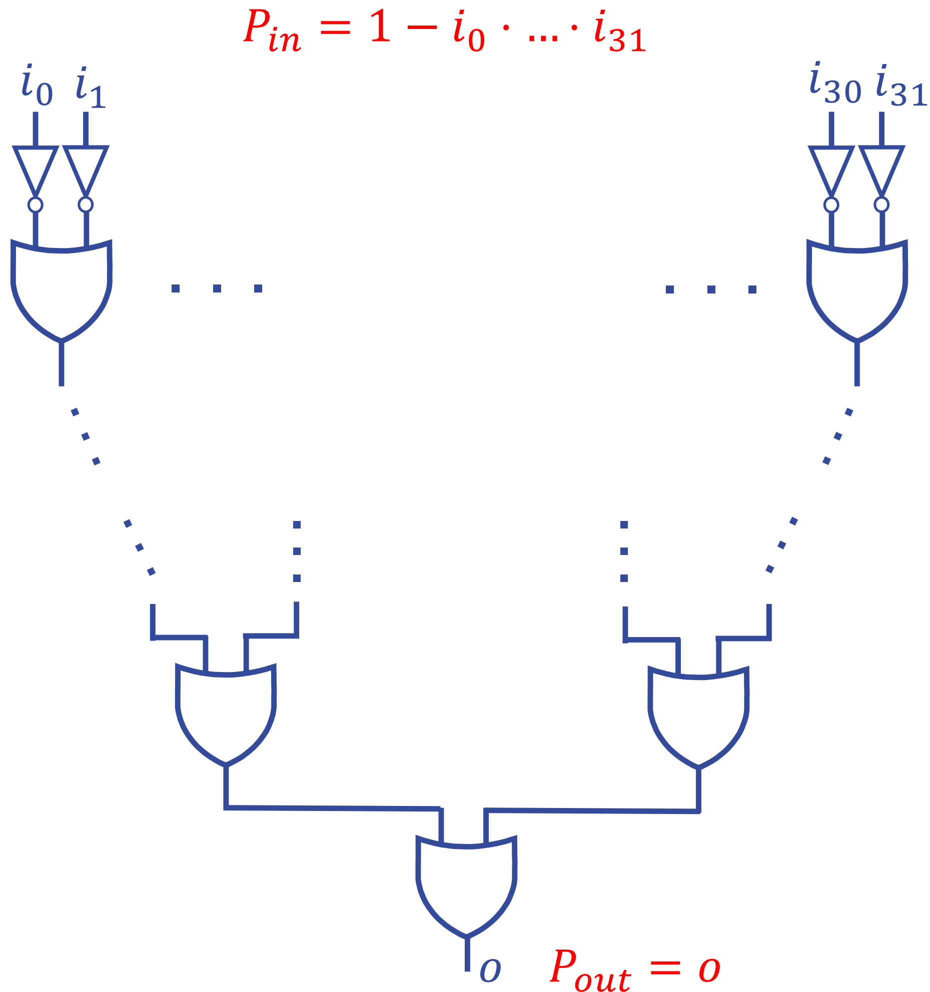
→ $P' = 1 - \overline{x_1} + \overline{x_1} x_2 + \overline{x_1} x_3 - \overline{x_1} x_2 x_3$

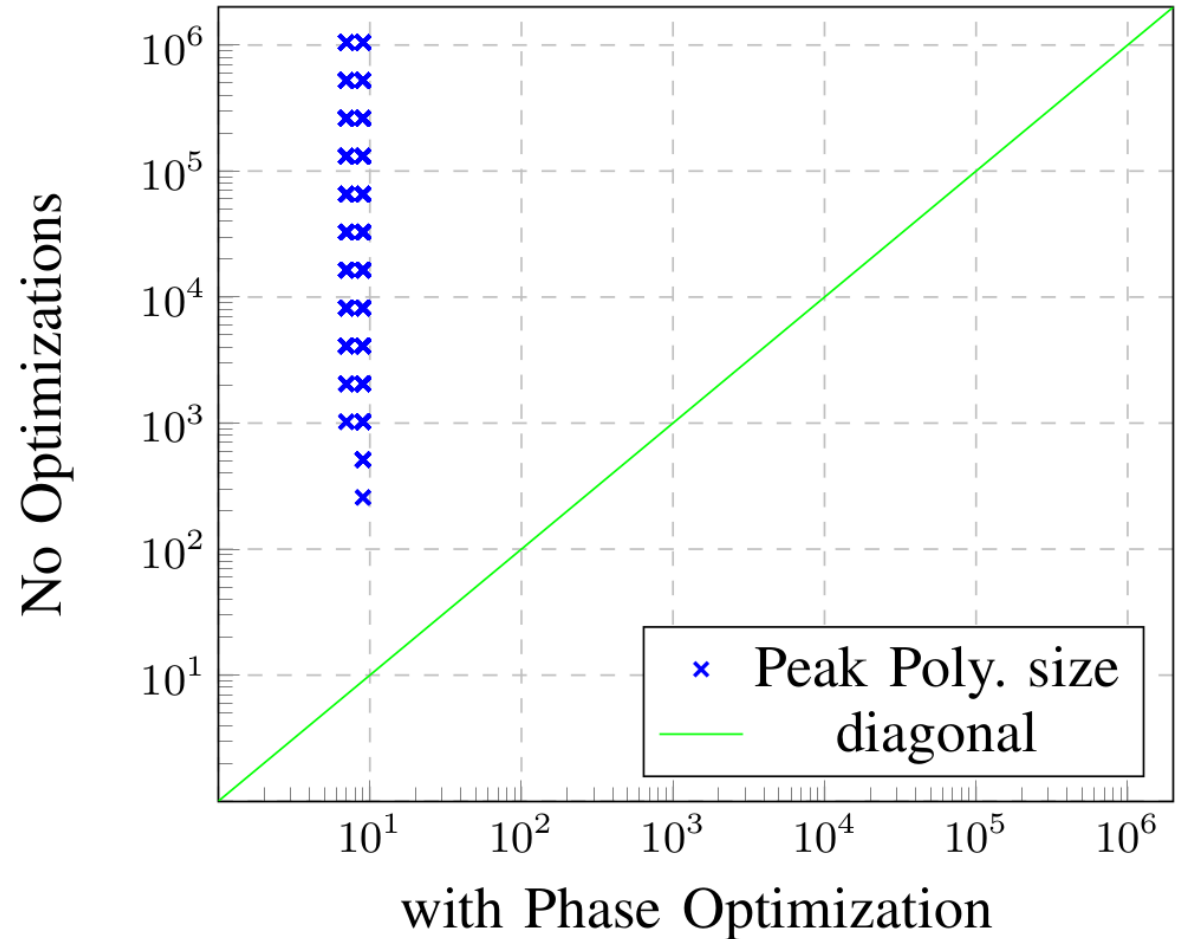
- Change phase of x_2 by replacing it with $1 - \overline{x_2}$

→ $P'' = 1 - \overline{x_1}\,\overline{x_2} + \overline{x_1}\,\overline{x_2} x_3$

- Change phase of x_3 by replacing it with $1 - \overline{x_3}$ finally leads to $P''' = 1 - \overline{x_1}\,\overline{x_2}\,\overline{x_3}$

Our new approach
Dynamic Phase Optimization Experiment





Our new approach
Dynamic Order Optimization

- Improvement of Dynamic Ordering from DyPoSub [Mahzool et al., DATE'20]

- Main difference: **hierarchical dynamic approach on two different levels**

1. **Higher "component level"**
   → choosing a good candidate component in every step
2. **Lower "individual node level"** inside the components
   → avoid peaks inside rewriting of a component
   → **Phase Optimization comes into play here**
   → for speed-up: first try out static orders (BFS-, DFS-based)

Experimental Results
Experimental Setup

- Examine 5 different tools:
  1. Our tool DynPhaseOrderOpt
  2. AMulet 2.2 [Kaufmann, Biere, TAP'22]
  3. TeluMA [Kaufmann et al., DATE'22]
  4. DyPoSub [Mahzool et al., DATE'20]
  5. RevSCA-2.0 [Mahzoon et al., TCAD'22]

- Timeout: 12h
- Available memory: 32GB

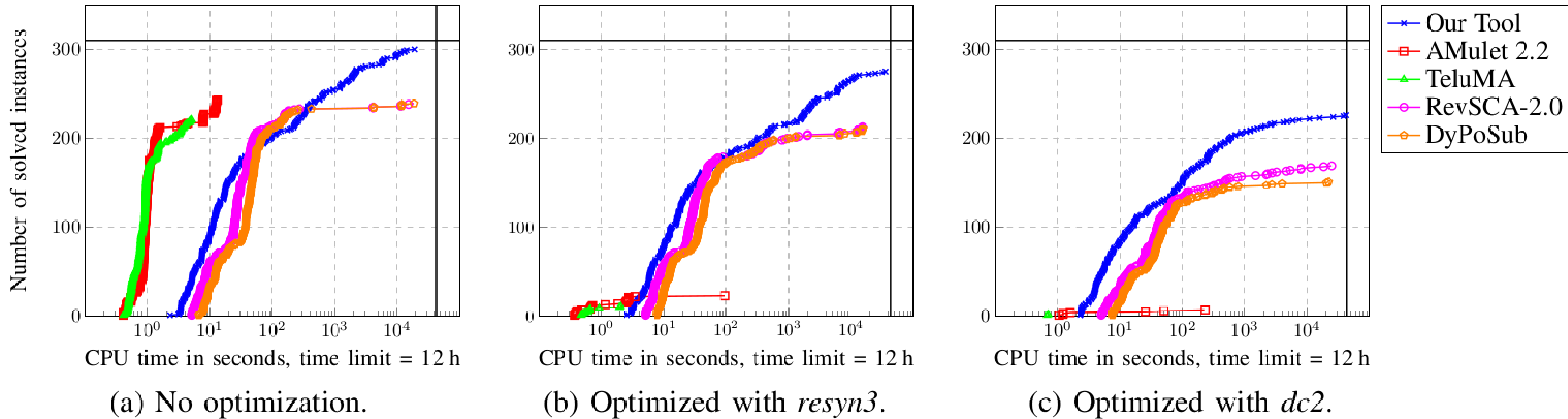
Experimental Results
Examined benchmark set

- Consists of **64-bit unsigned multipliers** from various sources:
    - All 192 benchmarks from AOKI set [Aoki et al., IEICE Trans. Fundam.'06]
    - All 28 benchmarks from GenMul [Mahzoon et al., 2023]
    - All 90 benchmarks from Multgen [Temel, 2019]

    → **In total 310 different multipliers**

    - For checking robustness, consider different optimizations:
        1. None
        2. Optimized by ABC with option “resyn3“
        3. Optimized by ABC with option “dc2“

Experimental Results
Run times of solved instances



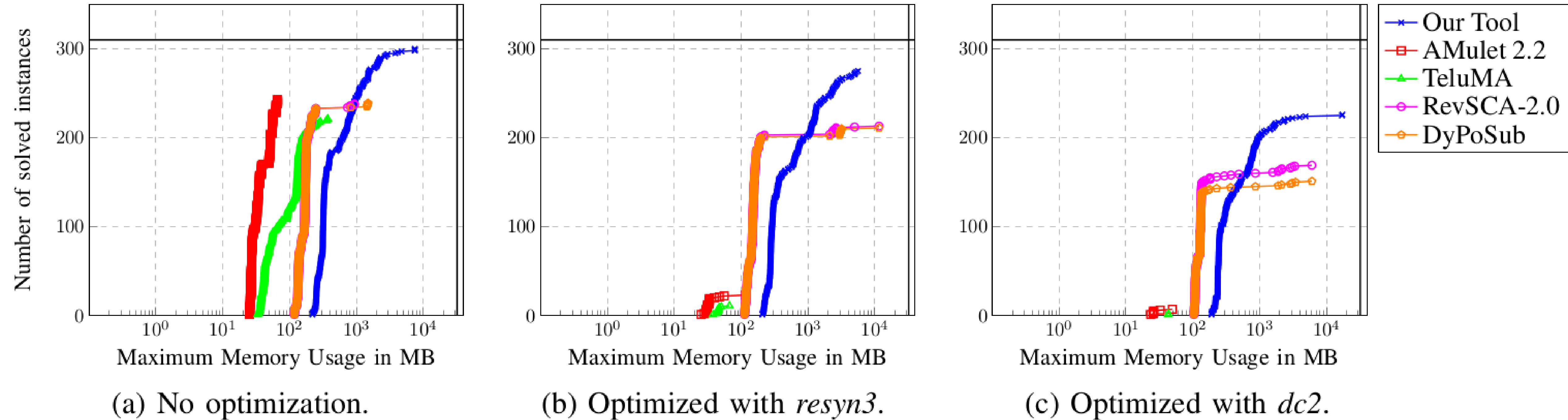
(a) No optimization.

(b) Optimized with *resyn3*.

(c) Optimized with *dc2*.

Experimental Results

Memory consumption of solved instances



Experimental Results

Causes for failed instances

	Our Tool			AMulet 2.2			TeluMA			RevSCA-2.0			DyPoSub		
Result Type	no	re3	dc2	no	re3	dc2	no	re3	dc2	no	re3	dc2	no	re3	dc2
Solved	300	275	226	243	23	7	221	11	2	238	213	169	239	211	151
Timeout	10	34	76	57	282	303	89	219	165	21	20	53	18	23	98
Mem.out	0	1	8	2	5	0	0	1	0	21	70	73	24	69	54
SegFault	0	0	0	(2)	(52)	(15)	0	77	143	12	0	0	12	0	0
False Buggy	0	0	0	8	0	0	0	2	0	18	7	15	17	7	7

Experimental Results

Causes for failed instances

	Our Tool			AMulet 2.2			TeluMA			RevSCA-2.0			DyPoSub		
Result Type	no	re3	dc2	no	re3	dc2	no	re3	dc2	no	re3	dc2	no	re3	dc2
Solved	300	275	226	243	23	7	221	11	2	238	213	169	239	211	151
Timeout	10	34	76	57	282	303	89	219	165	21	20	53	18	23	98
Mem.out	0	1	8	2	5	0	0	1	0	21	70	73	24	69	54
SegFault	0	0	0	(2)	(52)	(15)	0	77	143	12	0	0	12	0	0
False Buggy	0	0	0	8	0	0	0	2	0	18	7	15	17	7	7

Conclusion and Future Work

- Provided new simple method: **Dynamic Phase and Order Optimization**
- Solves more clean as well as optimized multiplier benchmarks → more robust
- **In the future:** Verification of other multipliers as well as other arithmetic circuits