

Automating System Configuration

Nestan Tsiskaridze

07/01/2021

Joint work with:

AHA Faculty: Clark Barrett, Mark Horowitz

Lake Team: Qiaoyi Liu, Kavya Sreedhar, Maxwell Strange

Pono Team: Ahmed Irfan, Florian Lonsing, Makai Mann

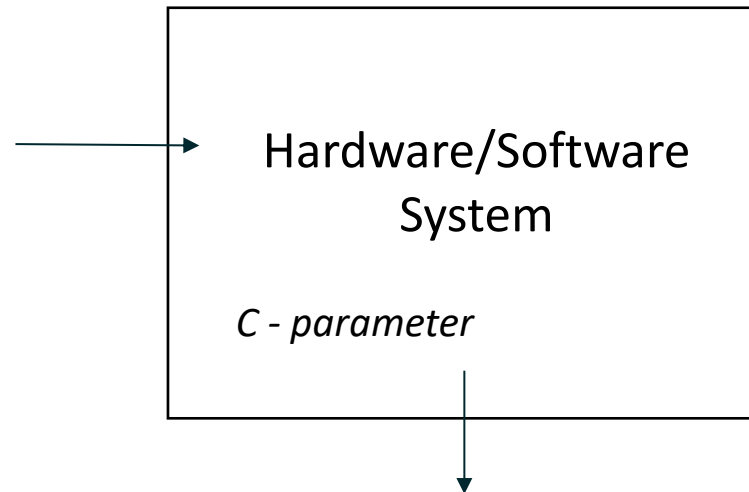
Boolector Team: Aina Niemetz, Mathias Preiner

Configurable Systems

Configurable systems are **parameterized** to increase their flexibility or functionality.

Such systems can be used in **various contexts** or **applications**.

Goal: Choose the **appropriate parameter values** for the context or application in which the system will be used.



Configurable Systems

Why configurable systems?

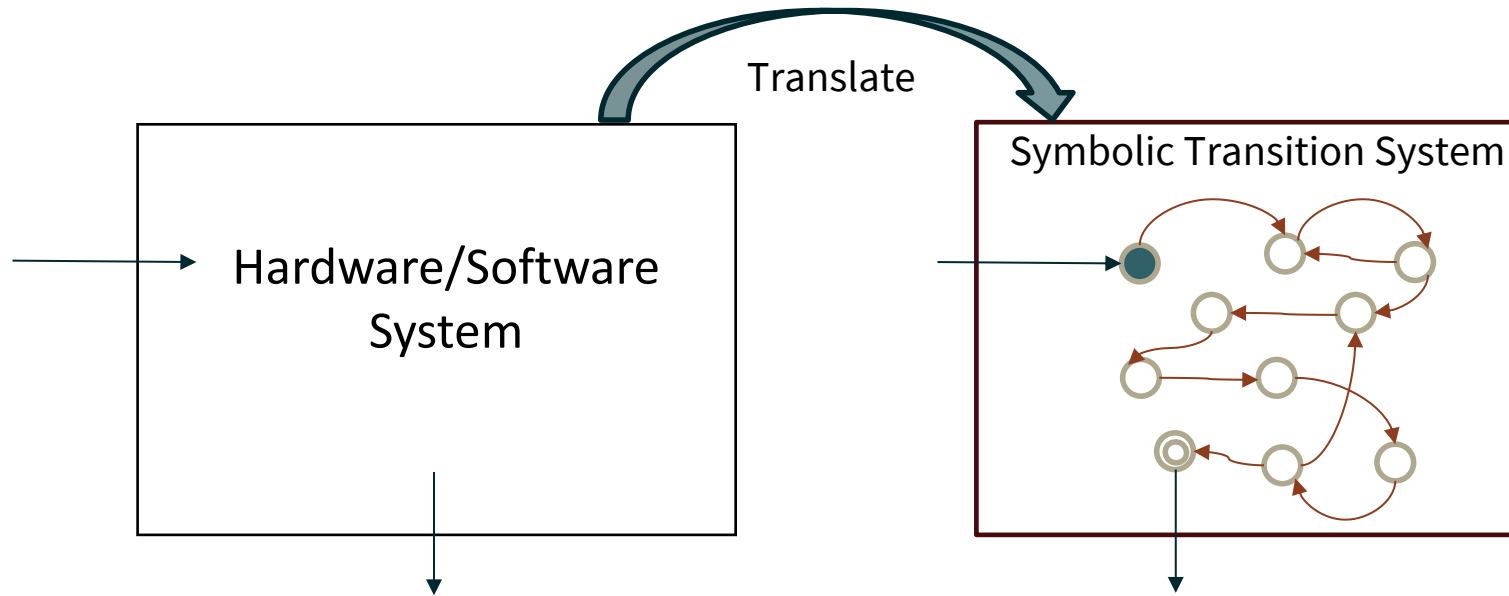
- Make designs more **widely applicable** and **re-usable**.
- Systems can be configured at **design time**, **setup time**, or **during normal operation**.
- Most hardware and software systems require **some degree of configuration**.
- Often used when **integrating decoupled parts** of a system, including integrating software and hardware.
- **Agile design approaches** require **rapid integration** of changing parts of a system while continuously maintaining correct end-to-end functionality.

Configurable Systems

Why automation?

- Today's systems increase in **scale** and **complexity**.
- **Manual** configuration is **error-prone** and may even be **impossible**, depending on how frequently the systems need to be reconfigured.
- **Automation is needed.**
- Especially useful for an agile design process.

Formal System Model

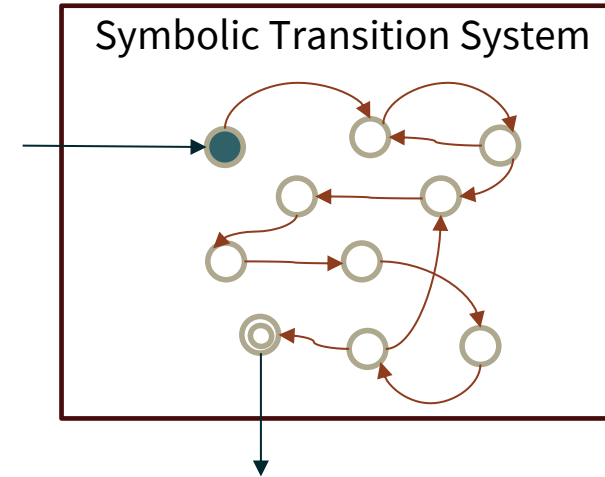


Symbolic Transition System is built by translating another representation:

E.g. a program, a mathematical model, a hardware description, etc.

Symbolic Transition System

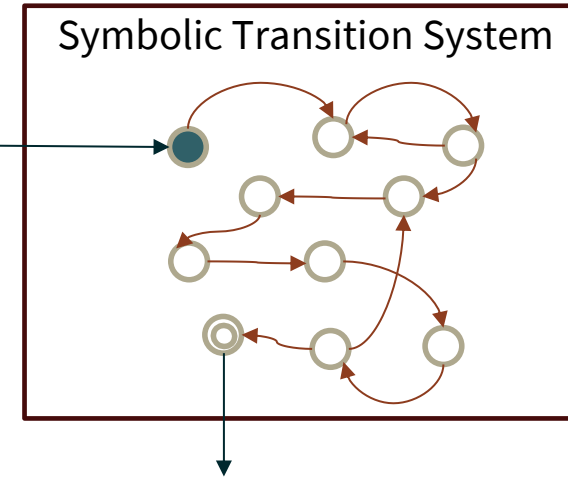
- **States** are made up of **state variables** $v \in V$;
- A **state** is an **assignment** to all variables;
- A **Symbolic Transition System** is a tuple $S := \langle V, I, T \rangle$
 - V – a finite set of **state variables**;
 - $I(V)$ – a formula denoting the **initial states** of S ;
 - $T(V, V')$ – a formula denoting a **transition relation**, V' denotes next state variables.



Symbolic Transition System

- An **execution of length k** is a sequence of states $\pi := s_0 s_1 \dots s_{k-1}$ such that:

- the first state s_0 respects I ;
- every adjacent pair (s_{i-1}, s_i) , $0 \leq i < k$, respects T .

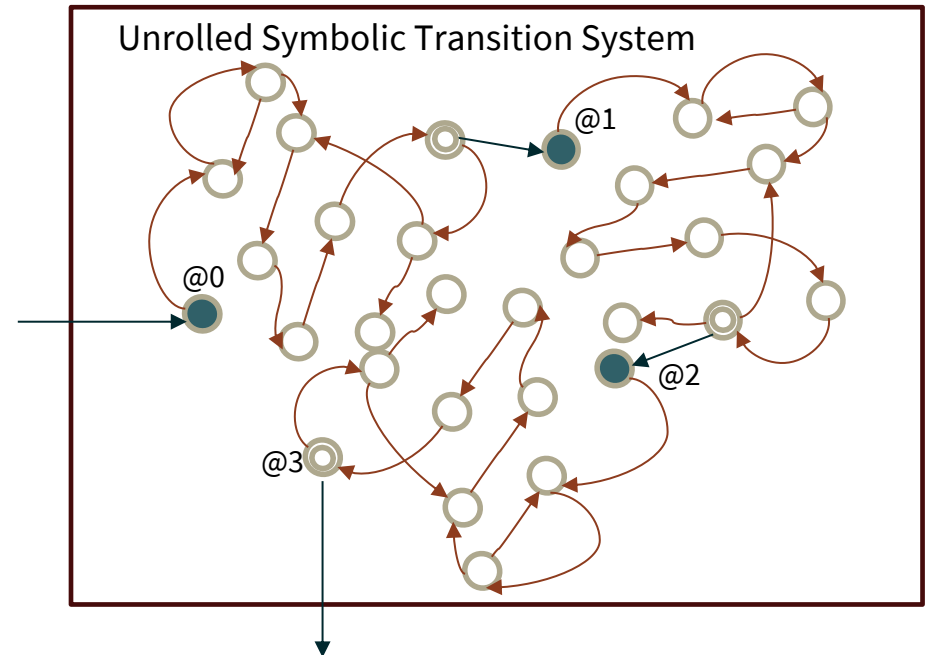


- An **unrolling of length k** of a symbolic transition system is a **formula** that **captures an execution of length k** by:

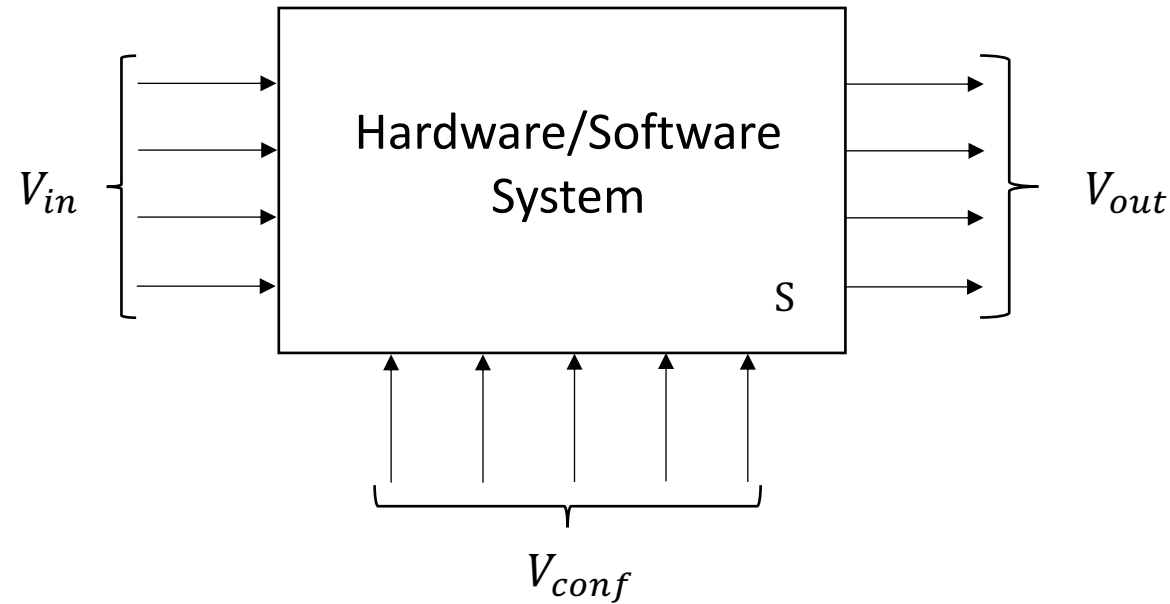
introducing fresh timed variables $V@i$;

creating copies of the transition relation $T(V@i, V@(i + 1))$.

$$\text{unroll}(S, k) := I(V@0) \wedge \bigwedge_{i=0}^{k-1} T(V@i, V@(i + 1))$$



Formal System Model



- $S := \langle V, I, T \rangle$

$V_{in} \cup V_{out} \cup V_{conf} \subseteq V$. Pairwise intersections of these sets may either be empty or non-empty.

$$V_{conf} \neq \emptyset$$

V_{in} do not appear in $I(V)$ and V'_{in} do not appear in T .

Formal System Model

- P — an **application-supplied input-output property**, or an **input-output specification**, a formula capturing an input-output relationship for k transitions:

$$P(V_{in}@0, \dots, V_{in}@k, V_{out}@0, \dots, V_{out}@k).$$

Example

For some constant values of inputs c_{in}^i and constant values of outputs c_{out}^i :

$$P := \bigwedge_{i=0}^{k-1} (V_{in}@i = c_{in}^i \wedge V_{out}@i = c_{out}^i),$$

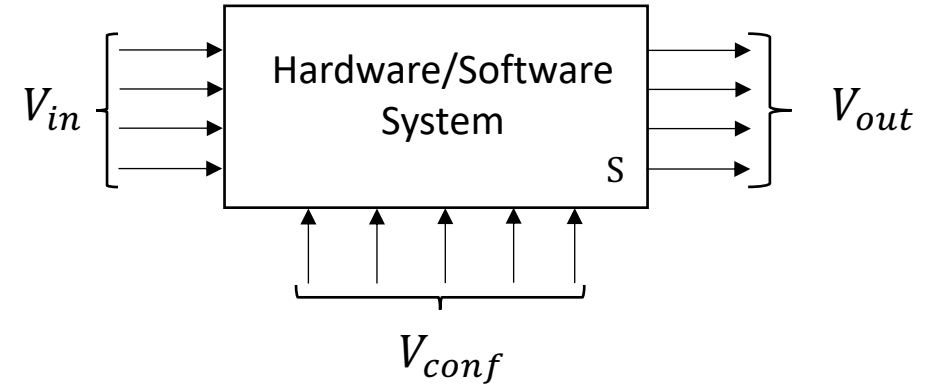
- Typically, we want a **configuration constancy constraint** to hold:

$$conf(V_{conf}, k) := \bigwedge_{i=0}^{k-1} (V_{conf}@(i+1) = V_{conf}@i)$$

Configuration Problem

A **Configuration Problem** is a tuple: $CP := \langle S, k, V_{in}, V_{out}, V_{conf}, P \rangle$

A **configuration** C is an assignment to the variables in V_{conf} .



Configuration finding problem:

Given a CP , find a configuration C for S such that S satisfies P with configuration C .

We reduce to **satisfiability checking** of the **configuration formula**:

$$\varphi(CP) := \text{unroll}(S, k) \wedge \text{conf}(V_{conf}, k) \wedge P(V_{in}@0, \dots, V_{in}@k, V_{out}@0, \dots, V_{out}@k).$$

Configuration Problem Example

(simple ALU)

Let $S := \langle x: int, a: int, c: Bool, x = 0, x' = ite(c, x + a, x - a) \rangle$ be an STS.

$V_{in} = \{a\}, V_{out} = \{x\}, V_{conf} = \{c\}.$

Two ways to configure S :

- **always adds** the current input to the current state variable,
- **always subtracts** the current input from the current state variable.

Configuration Problem Example

(simple ALU)

Let $S := \langle x: int, a: int, c: Bool, x = 0, x' = ite(c, x + a, x - a) \rangle$ be an STS.

$V_{in} = \{a\}, V_{out} = \{x\}, V_{conf} = \{c\}.$

Consider an **input-output relation** for $k = 2$:

$P_1(a@0, a@1, a@2, x@0, x@1, x@2) := a@0 = 1 \wedge a@1 = 1 \wedge a@2 = 1 \wedge x@0 = 0 \wedge x@1 = 1 \wedge x@2 = 2$

Check satisfiability of:

$unroll(S, 2) \wedge conf(c@0, c@1, c@2) \wedge P_1(a@0, a@1, a@2, x@0, x@1, x@2).$

$x@0 = 0 \wedge$

$x@1 = ite(c@0, x@0 + a@0, x@0 - a@0) \wedge$

$x@2 = ite(c@1, x@1 + a@1, x@1 - a@1) \wedge$

$c@1 = c@0 \wedge c@2 = c@1 \wedge$

$a@0 = 1 \wedge a@1 = 1 \wedge a@2 = 1 \wedge x@0 = 0 \wedge x@1 = 1 \wedge x@2 = 2$

Satisfiable when $c@0 = True$!

Configuration Problem Example

(simple ALU)

Let $S := \langle x: int, a: int, c: Bool, x = 0, x' = ite(c, x + a, x - a) \rangle$ be an STS.

$V_{in} = \{a\}, V_{out} = \{x\}, V_{conf} = \{c\}.$

Consider an **input-output relation** for $k = 2$:

$P_2(a@0, a@1, a@2, x@0, x@1, x@2) := a@0 = 1 \wedge a@1 = 1 \wedge a@2 = 1 \wedge x@0 = 0 \wedge x@1 = 1 \wedge x@2 = 0$

Check satisfiability of:

$unroll(S, 2) \wedge conf(c@0, c@1, c@2) \wedge P_2(a@0, a@1, a@2, x@0, x@1, x@2).$

$x@0 = 0 \wedge$

$x@1 = ite(c@0, x@0 + a@0, x@0 - a@0) \wedge$

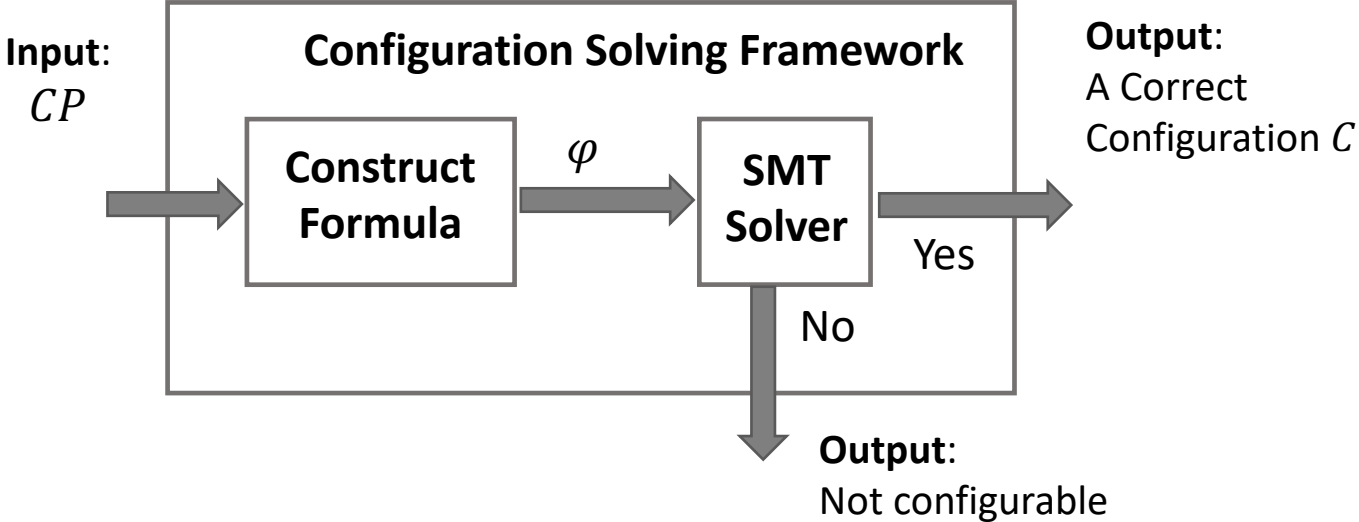
$x@2 = ite(c@1, x@1 + a@1, x@1 - a@1) \wedge$

$c@1 = c@0 \wedge c@2 = c@1 \wedge$

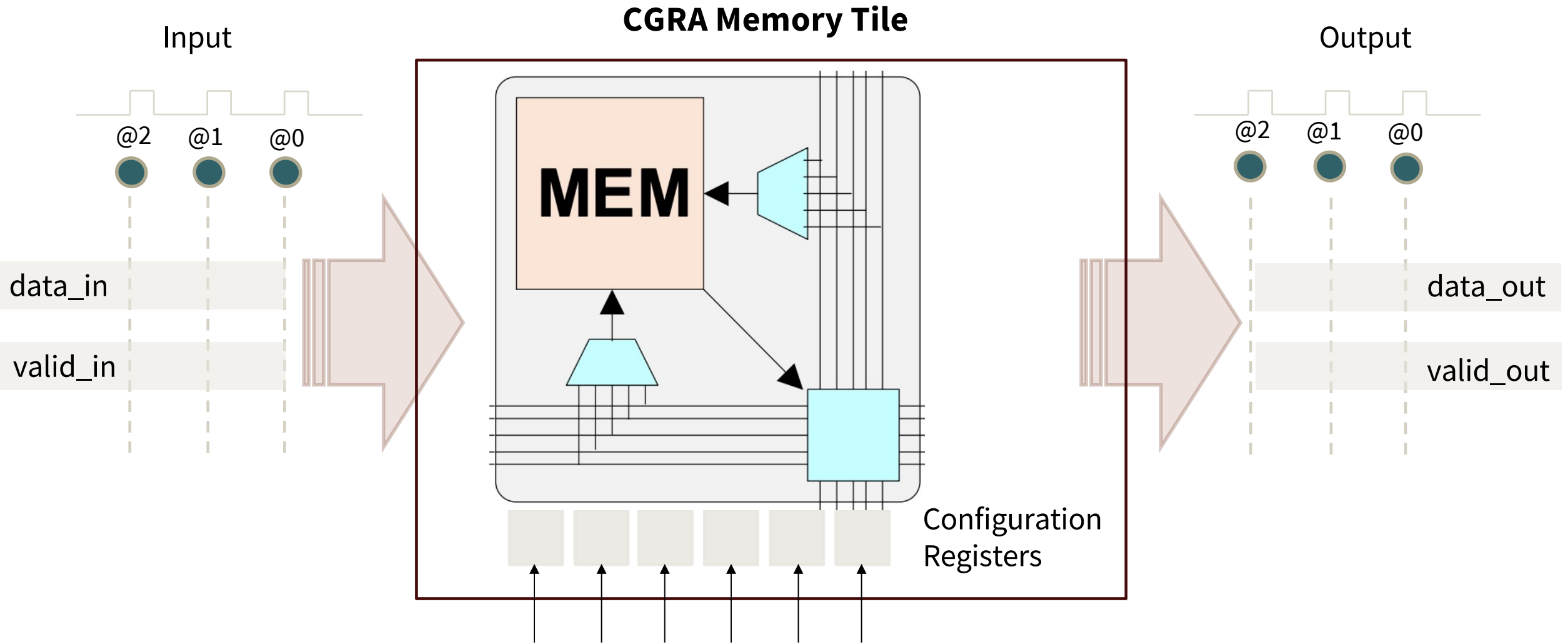
$a@0 = 1 \wedge a@1 = 1 \wedge a@2 = 1 \wedge x@0 = 0 \wedge x@1 = 1 \wedge x@2 = 0$

Unsatisfiable!

Configuration Solving Framework (Basic) Scheme



Coarse-Grained Reconfigurable Architecture (CGRA)



CGRA Memory Tile

Yosys

Pono

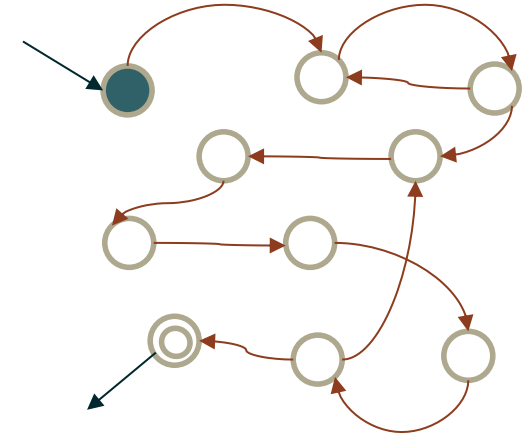
System Verilog

```
63
64 module LakeTop (
65     input logic [1:0] [15:0] addr_in,
66     input logic [1:0] [15:0] chain_data_in,
67     output logic [1:0] [15:0] chain_data_out,
68     input logic chain_idx_input,
69     input logic chain_idx_output,
70     input logic [1:0] chain_valid_in,
71     output logic [1:0] chain_valid_out,
72     input logic clk,
73     input logic clk_en,
74     input logic [7:0] config_addr_in,
75     input logic [31:0] config_data_in,
76     output logic [1:0] [31:0] config_data_out,
77     input logic [1:0] config_en,
```

Btor2

```
1 ; BTOR description generated by Yosys 0.9+1706
  for module LakeTop.
2 1 sort bitvec 16
3 2 input 1 addr_in[0]
4 3 input 1 addr_in[1]
5 4 input 1 chain_data_in[0]
6 5 input 1 chain_data_in[1]
7 6 sort bitvec 1
8 7 input 6 chain_idx_input
9 8 input 6 chain_idx_output
10 9 sort bitvec 2
11 10 input 9 chain_valid_in
12 11 input 6 clk
13 12 input 6 clk_en
14 13 sort bitvec 8
```

Symbolic Transition System



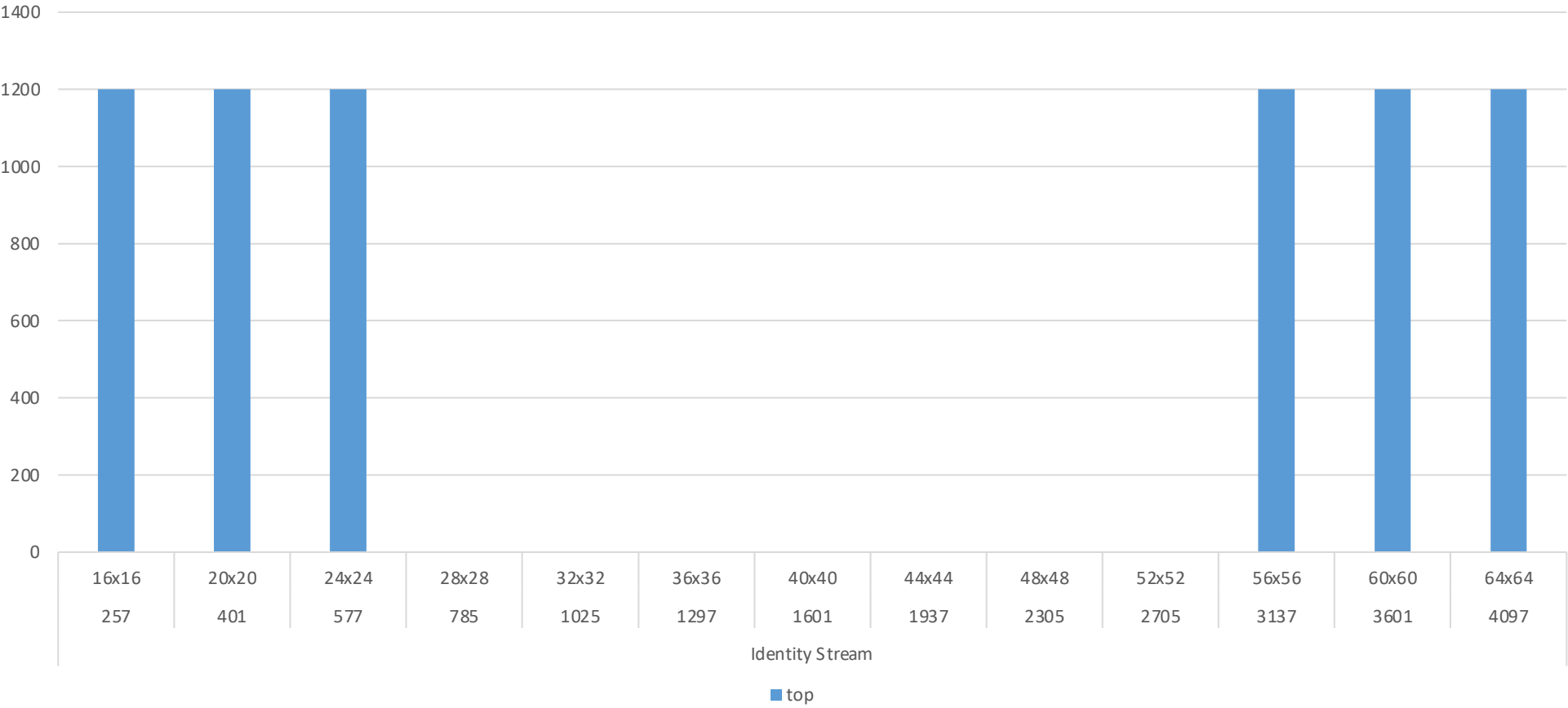
Btor2 format allows specifying word-level model checking problems.

Pono: our Performant, Adaptable, and Extensible SMT-based Model Checker.

We use **Boolector** as our underlying SMT solver.

Design	Size (Σ bw)	FF	Gates
top	34 980	34 998	164 696
agg	1 354	1 372	19 676
tb	1 108	1 126	18 538
sram	33 694	33 712	150 750

Identity Stream
2x Intel Xeon E5-2620 v4
2.10GHz 8-core 128GB.
Timeout: 20 min
Memout: 100 GB.



Scalability Bottlenecks

Two main **bottlenecks**:

- **Size of the transition system S ;**
- **Length of the input sequence k** — number of transitions in P , size of the formula $unroll(S, k)$.

Modular Decomposition

Let $CP := \langle S, k, V_{in}, V_{out}, V_{conf}, P \rangle$ with $S := \langle V, I, T \rangle$.

(CP_1, CP_2) is a **decomposition** of CP into two configuration problems:

$$CP_i := \langle S_i, k, V_{in}^i, V_{out}^i, V_{conf}^i, P_i \rangle \text{ with } S_i := \langle V_i, I_i, T_i \rangle \text{ for } i = 1, 2$$

if:

$$(i) \quad T_1(V_1, V'_1) \wedge T_2(V_2, V'_2) \Rightarrow T(V, V');$$

$$(ii) \quad I_1(V_1) \wedge I_2(V_2) \Rightarrow I(V);$$

$$(iii) \quad P_1 \wedge P_2 \Rightarrow P.$$

Modular Decomposition

Given a *candidate* decomposition (CP_1, CP_2) of a configuration problem CP , procedure **SOLVEMODULAR** attempts to solve CP by solving CP_1 and CP_2 .

Algorithm 1 Modular configuration finding.

Procedure SOLVEMODULAR

Input: $(\mathcal{CP}_1, \mathcal{CP}_2)$ a decomposition of \mathcal{CP} .

Output: a pair (r, \mathcal{C}) where if $r = sat$, then \mathcal{C} is a configuration of \mathcal{S}

```
1:  $\phi_1 := \text{MAKECP}(\mathcal{CP}_1)$ 
2:  $(r, \mathcal{I}_1) := \text{SOLVE}(\phi_1)$ ,
3: if  $r = sat$  then
4:    $\phi_2 := \text{MAKECP}(\mathcal{CP}_2) \wedge \text{GETABDUCT}(\phi_1, \mathcal{I}_1)$ 
5:    $(r, \mathcal{I}) := \text{SOLVE}(\phi_2)$ 
6: end if
7: return  $(r, \mathcal{I}^{V_{\text{conf}}})$ 
```

MAKECP constructs the configuration formula for a CP .

SOLVE invokes a solver to check the satisfiability of the configuration formula.

GETABDUCT returns an *abduct* formula ψ of ϕ_1 : all interpretations that satisfy ψ also satisfy ϕ_1 .

Goal: use information in \mathcal{I}_1 to find a simple formula ψ .

We search for a set of sub-terms in ϕ_1 such that, if we constrain them to be equal to their values in \mathcal{I}_1 , this ensures that ϕ_1 is satisfied.

Modular Decomposition

Theorem. (*Soundness*)

If (CP_1, CP_2) is a decomposition of a configuration problem CP , and **SOLVEMODULAR** (CP_1, CP_2) returns a pair (sat, C) , then C is a correct configuration of CP .

Algorithm 1 Modular configuration finding.

Procedure SOLVEMODULAR

Input: $(\mathcal{CP}_1, \mathcal{CP}_2)$ a decomposition of \mathcal{CP} .

Output: a pair (r, \mathcal{C}) where if $r = sat$, then \mathcal{C} is a configuration of \mathcal{S}

```
1:  $\phi_1 := \text{MAKECP}(\mathcal{CP}_1)$ 
2:  $(r, \mathcal{I}_1) := \text{SOLVE}(\phi_1)$ ,
3: if  $r = sat$  then
4:    $\phi_2 := \text{MAKECP}(\mathcal{CP}_2) \wedge \text{GETABDUCT}(\phi_1, \mathcal{I}_1)$ 
5:    $(r, \mathcal{I}) := \text{SOLVE}(\phi_2)$ 
6: end if
7: return  $(r, \mathcal{I}^{V_{\text{conf}}})$ 
```

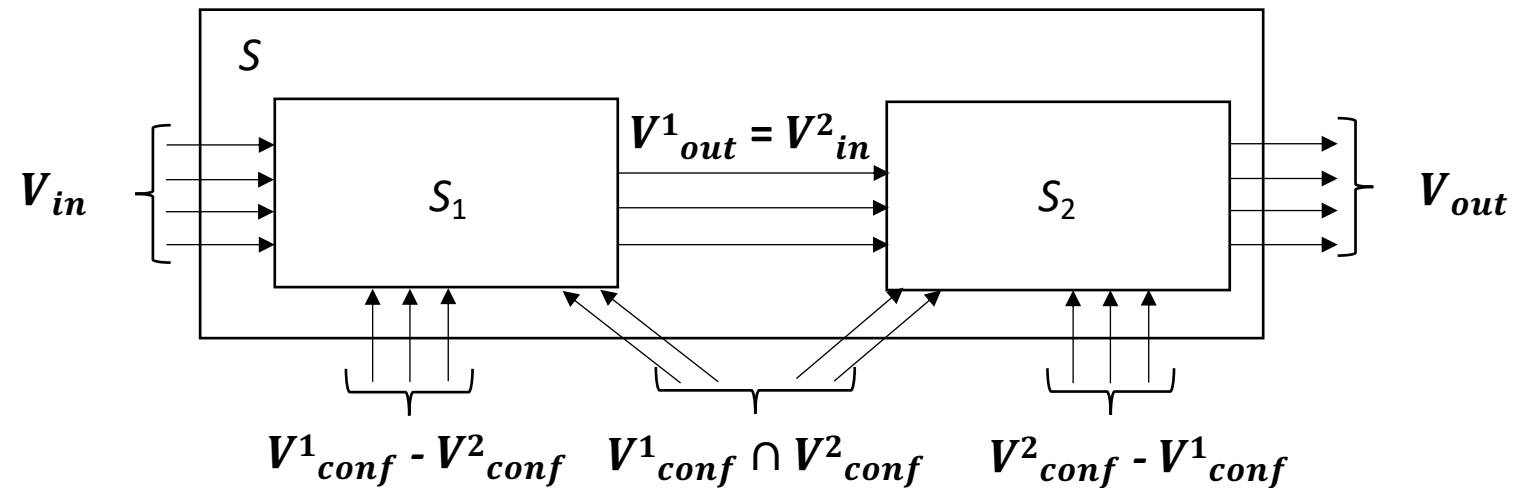
Note: If **SOLVEMODULAR** returns $r = unsat$, this **does not** (in general) **imply** that CP is **unconfigurable**.

It may be that the particular decomposition fails, or the particular solution found for CP_1 is at fault.

Modular Decomposition in Practice

In practice, the algorithm **works well** when the decomposition separates a module into two largely independent parts.

A modular decomposition of system S into systems S_1 and S_2 .

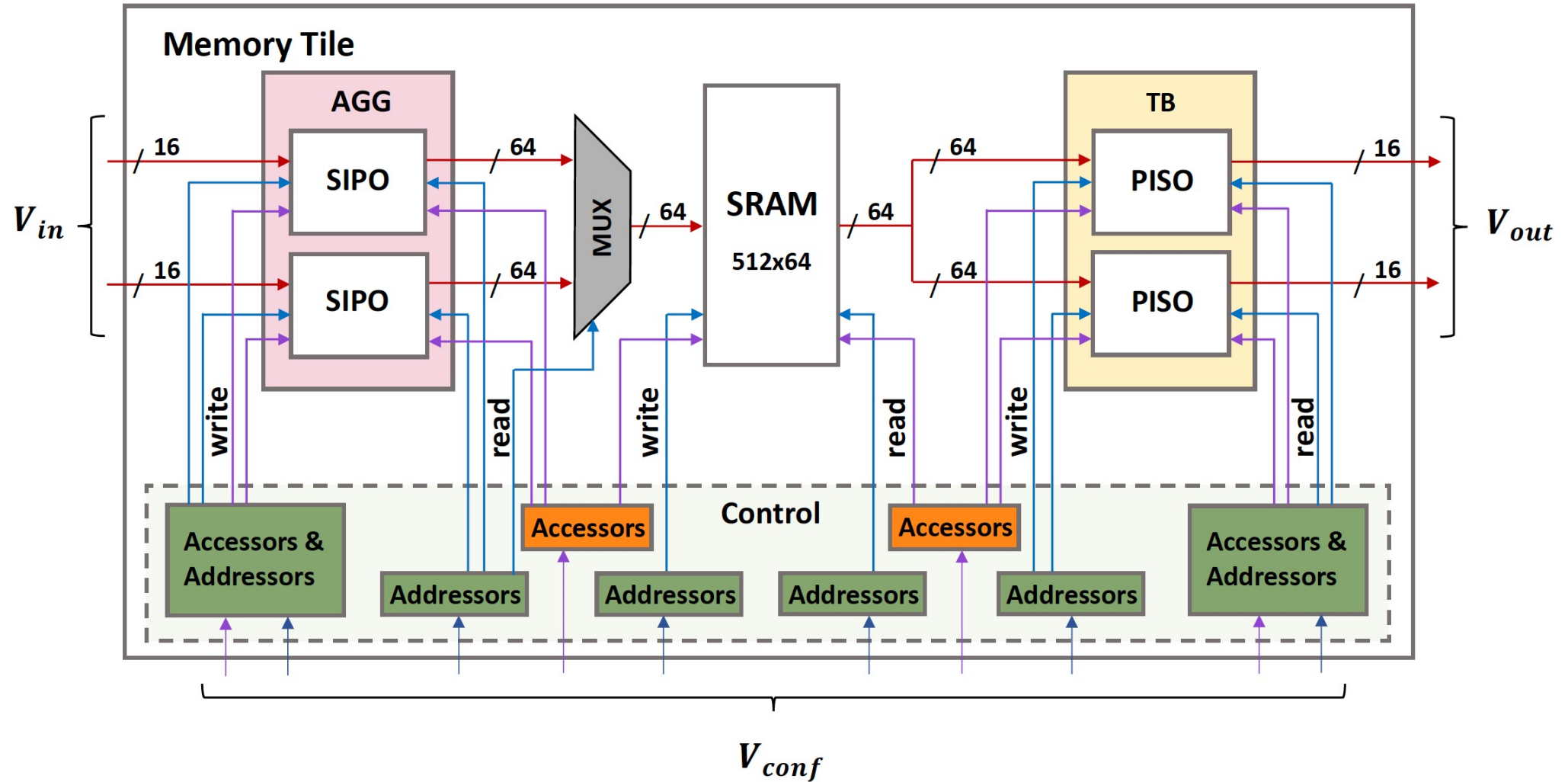


V^1_{out} and V^1_{conf} are the output and the configuration variables of S_1 .

V^2_{in} and V^2_{conf} are the input and the configuration variables of S_2 .

$$V_{conf} \subseteq V^1_{conf} \cup V^2_{conf}.$$

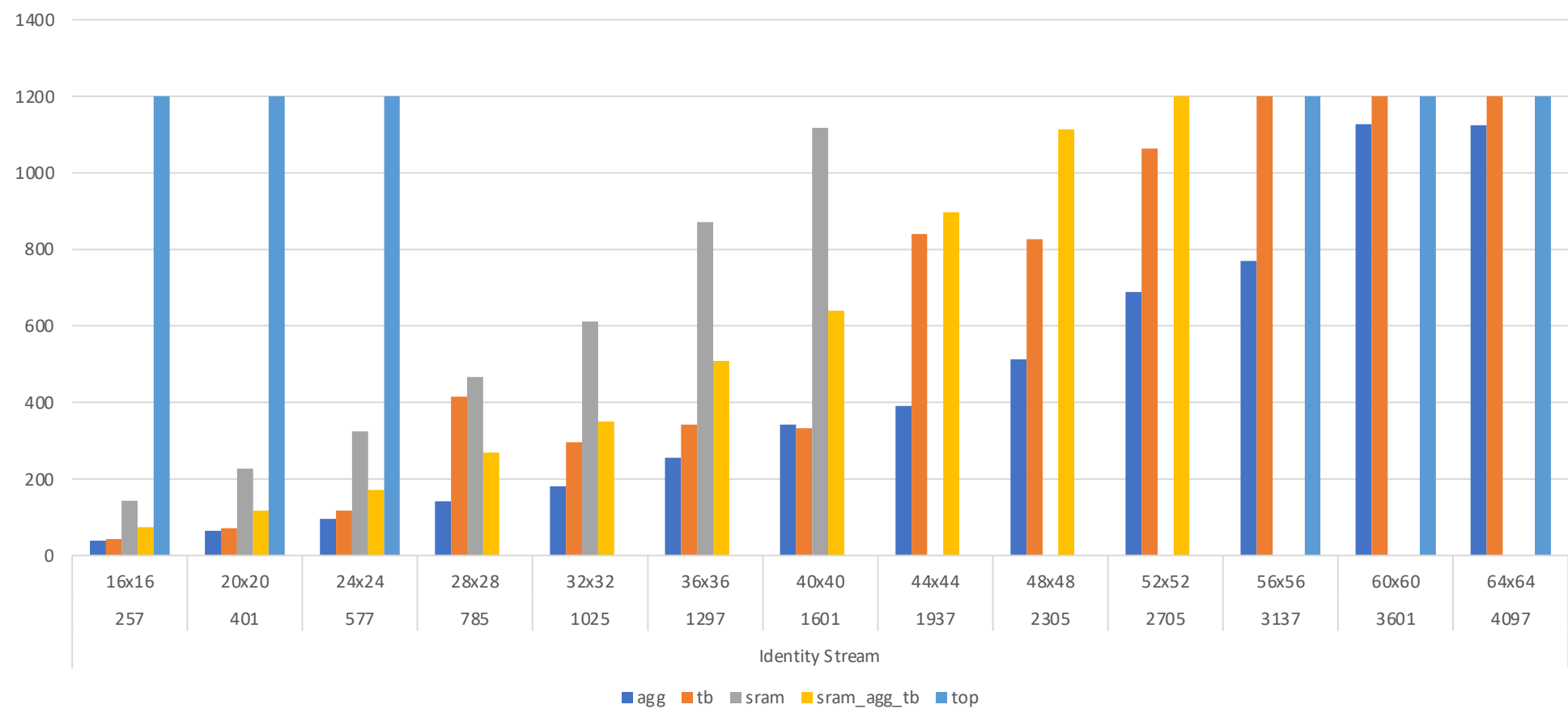
Memory Tile Architecture



Design	Size (Σ bw)	FF	Gates
top	34 980	34 998	164 696
agg	1 354	1 372	19 676
tb	1 108	1 126	18 538
sram	33 694	33 712	150 750

Identity Stream
 2x Intel Xeon E5-2620 v4
 2.10GHz 8-core 128GB.

Timeout: 20 min
Memout: 100 GB.



Configuration Optimization Problem

- **Human-readable configurations** are particularly important for users;
- Humans tend to generate the **simplest solutions** for **simplest application-specific scenarios**;
- **Producing simple solutions** is often analogous to **minimizing some metric** when finding solutions.
E. g.: readable solutions might minimize the value of some objective term representing the simplicity of a desired scenario;
- **Can we produce configurations that are best according to some metric?**

Configuration Optimization Problem

An **optimization problem** \mathcal{OP} is a tuple $\langle t, A, \preceq, \phi, \mathcal{O} \rangle$ where:

- t is an *objective term* to optimize of sort σ ;
- A is a set and \preceq is a total order over A .
- ϕ is a formula to satisfy; and
- $\mathcal{O} \in \{min, max\}$ is the optimization objective.

\mathcal{I} is a solution to \mathcal{OP} if $\sigma^{\mathcal{I}} = A$, $\mathcal{I} \models \phi$, and for any \mathcal{I}' , such that $\sigma^{\mathcal{I}'} = A$ and $\mathcal{I}' \models \phi$:

$$(\mathcal{O} = min \rightarrow t^{\mathcal{I}} \preceq t^{\mathcal{I}'}) \wedge (\mathcal{O} = max \rightarrow t^{\mathcal{I}'} \preceq t^{\mathcal{I}}).$$

Configuration Optimization Problem

A **multi-objective optimization problem** \mathcal{MOP} is a finite sequence of optimization problems $\{\mathcal{OP}_1, \dots, \mathcal{OP}_n\}$ over the same formula ϕ , where

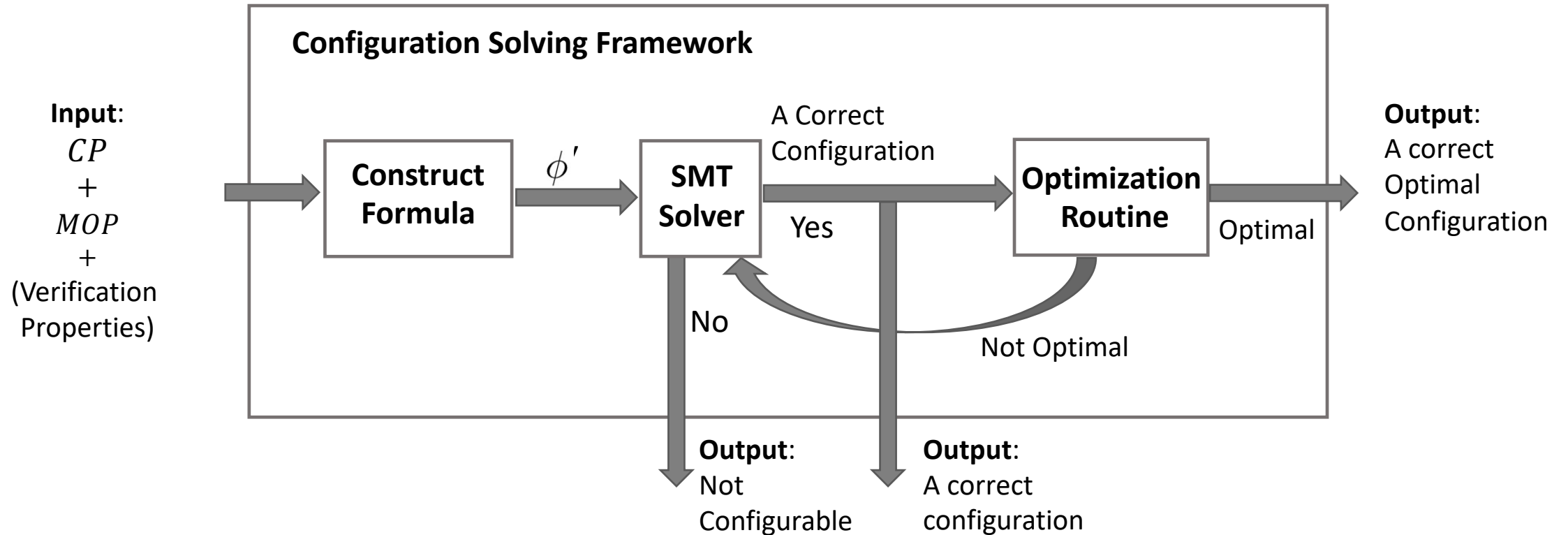
- $\mathcal{OP}_i := \langle t_i, A_i, \preceq_i, \phi, \mathcal{O}_i \rangle$ and
- t_i is of sort σ_i for $i \in [1, n]$.

\mathcal{I} is a solution to \mathcal{MOP} if $\sigma_i^{\mathcal{I}} = A_i$, $\mathcal{I} \models \phi$, and for any \mathcal{I}' , such that $\sigma_i^{\mathcal{I}'} = A_i$ and $\mathcal{I}' \models \phi$, either:

- (i) $t_i^{\mathcal{I}} = t_i^{\mathcal{I}'}$ for all $i \in [1, n]$; or
- (ii) for some $j \in [1, n]$, $t_i^{\mathcal{I}} = t_i^{\mathcal{I}'}$ for all $i \in [1, j)$, and
$$(\mathcal{O}_j = \min \rightarrow t_j^{\mathcal{I}} \prec_j t_j^{\mathcal{I}'}) \wedge (\mathcal{O}_j = \max \rightarrow t_j^{\mathcal{I}'} \prec_j t_j^{\mathcal{I}}),$$

where \prec is the strict total order associated with \preceq .

Optimization-Assisted Configuration Solver





ϕ' is a conjunction of the configuration formula and the optional verification properties.

Optimize Memory Tile Configurations

The addressors and accessors use affine sequence generators to generate sequences of values for reading and writing.

4 types of configuration variables in streaming memory controllers:

They **manage loop-nests** and **address calculations**.

- Dimensionalities
 - Ranges
- 
- control the structure**
-
- Strides
 - Starting Addresses
- 
- control particular address**

Goal: find an addressing pattern for the **simplest looping**.

Optimization Objective 1

Minimize the *dim* variables – fewer nested loops and fewer loop counters, simpler solutions.

Prioritize minimizing *dim* variables controlling writes over those controlling reads – lower write complexity leads to lower read complexity.

A multi-objective optimization problem:

$$\begin{aligned}\mathcal{MOP}_1 &:= \{\mathcal{OP}_1, \mathcal{OP}_w^1, \dots, \mathcal{OP}_w^{d_w}, \mathcal{OP}_r^1, \dots, \mathcal{OP}_r^{d_r}\} : \\ \mathcal{OP}_1 &:= \langle \sum_i \dim_i, A_{BV}, \preceq_{BV}, \phi, \min \rangle \text{ for } i \in [1, d], \\ \mathcal{OP}_w^i &:= \langle \dim_w^i, A_{BV}, \preceq_{BV}, \phi, \min \rangle \text{ for } i \in [1, d_w] \\ \mathcal{OP}_r^i &:= \langle \dim_r^i, A_{BV}, \preceq_{BV}, \phi, \min \rangle \text{ for } i \in [1, d_r]\end{aligned}$$

A_{BV} – is the domain of bit-vectors;

\preceq_{BV} – the usual total order on bit-vector values;

d – the number of affine sequence generators in the module;

\dim_i for $i \in [1; d]$ – all of the *dim* variables in the module;

\dim_w^i for $i \in [1; d_w]$ – write *dim* variables;

\dim_r^i for $i \in [1; d_r]$ – read *dim* variables, $d_w + d_r = d$;

ϕ – the configuration formula.

Optimization Objective 2

Minimize the products of the *range* configuration variables in each loop-nest structure – eliminate unnecessary reads and writes to the memory.

An optimization problem:

$$\mathcal{OP}_2 := \langle \sum_{i=0}^{d-1} \prod_{j=0}^{dim_i-1} ranges_i[j], A_{BV}, \preceq_{BV}, \phi, min \rangle$$

$\sum_{i=0}^{d-1} \prod_{j=0}^{dim_i-1} ranges_i[j]$ – the aggregate number of reads or writes that occur to a particular memory.

Optimization Objective 3

Minimize *stride* variables – avoid generating configurations using unnecessarily large addresses.

An optimization problem:

$$\mathcal{OP}_3 := \langle \sum_i \text{strides}_i, A_{BV} \preceq_{BV} \phi, \text{min} \rangle$$

Optimization Objective 4

Minimize *offset* configuration variables in addressor modules – prevents unnecessary offsets, improves the readability.

Note: values of *offset* variables in the accessors are fixed by the application.

An optimization problem:

$$\mathcal{OP}_4 := \langle \sum_i \text{offset}_i, A_{BV}, \preceq_{BV}, \phi, \min \rangle$$

Combined Optimization Objective

A multi-objective optimization problem for finding human-readable configurations:

$$\mathcal{MOP}_{\mathcal{H}} := \{\mathcal{MOP}_1, \mathcal{OP}_2, \mathcal{OP}_3, \mathcal{OP}_4\}$$

with:

- $\mathcal{MOP}_1 := \{\mathcal{OP}_1, \mathcal{OP}_w^1, \dots, \mathcal{OP}_w^{d_w}, \mathcal{OP}_r^1, \dots, \mathcal{OP}_r^{d_r}\} :$
 $\mathcal{OP}_1 := \langle \sum_i \dim_i, A_{BV}, \preceq_{BV}, \phi, \min \rangle$ for $i \in [1, d]$,
 $\mathcal{OP}_w^i := \langle \dim_w^i, A_{BV}, \preceq_{BV}, \phi, \min \rangle$ for $i \in [1, d_w]$
 $\mathcal{OP}_r^i := \langle \dim_r^i, A_{BV}, \preceq_{BV}, \phi, \min \rangle$ for $i \in [1, d_r]$
- $\mathcal{OP}_2 := \langle \sum_{i=0}^{d-1} \prod_{j=0}^{\dim_i-1} \text{ranges}_i[j], A_{BV}, \preceq_{BV}, \phi, \min \rangle$
- $\mathcal{OP}_3 := \langle \sum_i \text{strides}_i, A_{BV}, \preceq_{BV}, \phi, \min \rangle$
- $\mathcal{OP}_4 := \langle \sum_i \text{offset}_i, A_{BV}, \preceq_{BV}, \phi, \min \rangle$

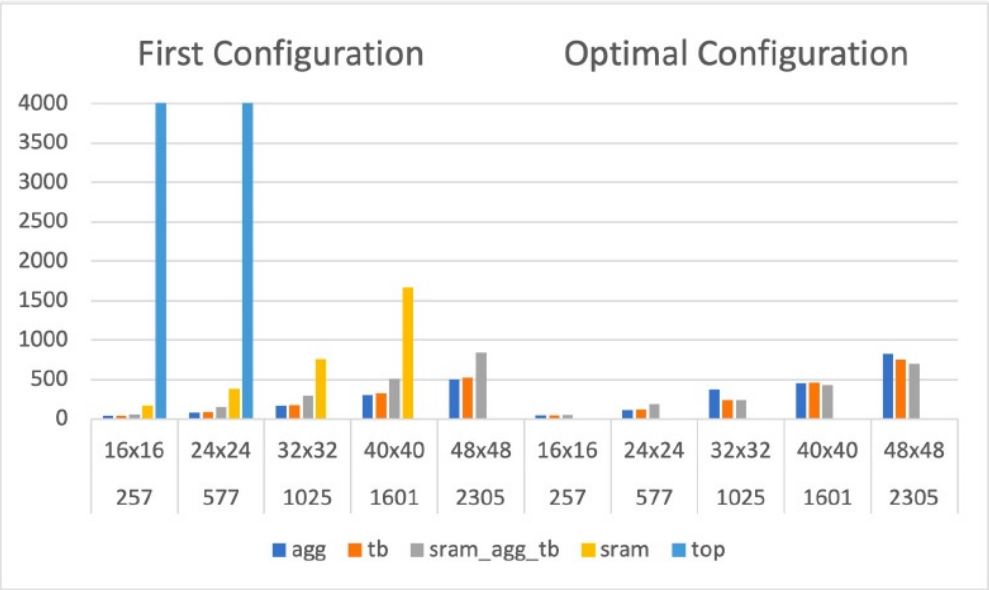
Stencil Applications

- **Identity** – simply streams the input back out in the same order. Useful as a baseline test, can also be used to implement a fixed delay on a stream.
- **3x3 Convolution** – multiplies a 3x3 sliding image window by a 3x3 kernel of constant values. Used in a variety of image processing applications.
- **Cascade** — implements a pipeline with two convolution kernels executed in sequence. Requires configuration of two memory tiles (conv and hw).
- **Harris** – a corner detection algorithm that can be used to infer image features. Requires configuration of five different memory tiles (cim, lxx, lxy, lyy, and pad).

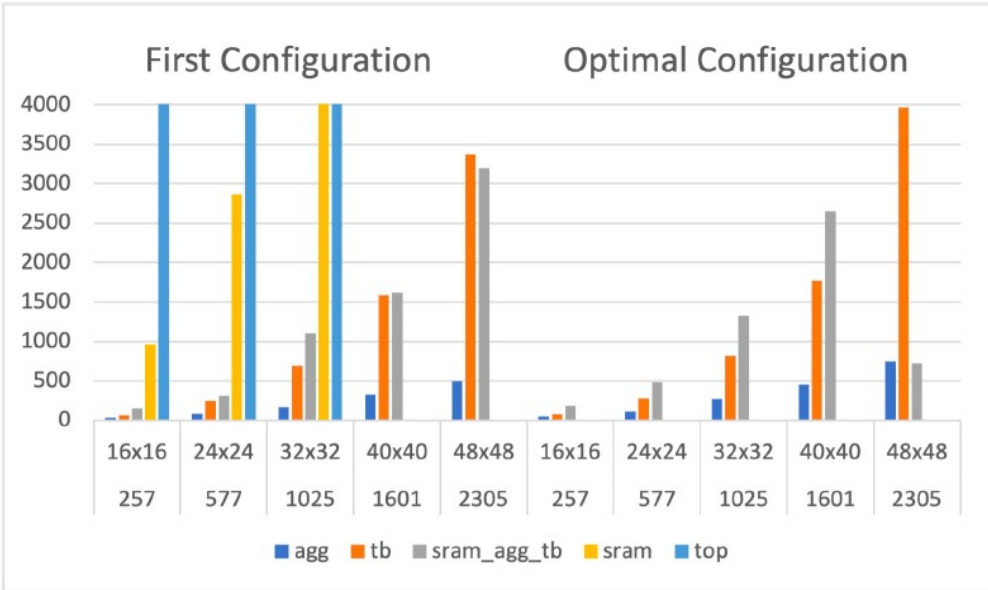
Design	Size (Σ bw)	FF	Gates
top	34 980	34 998	164 696
agg	1 354	1 372	19 676
tb	1 108.	1 126	18 538
sram	33 694	33 712	150 750

2x Intel Xeon E5-2620 v4
2.10GHz 8-core 128GB.

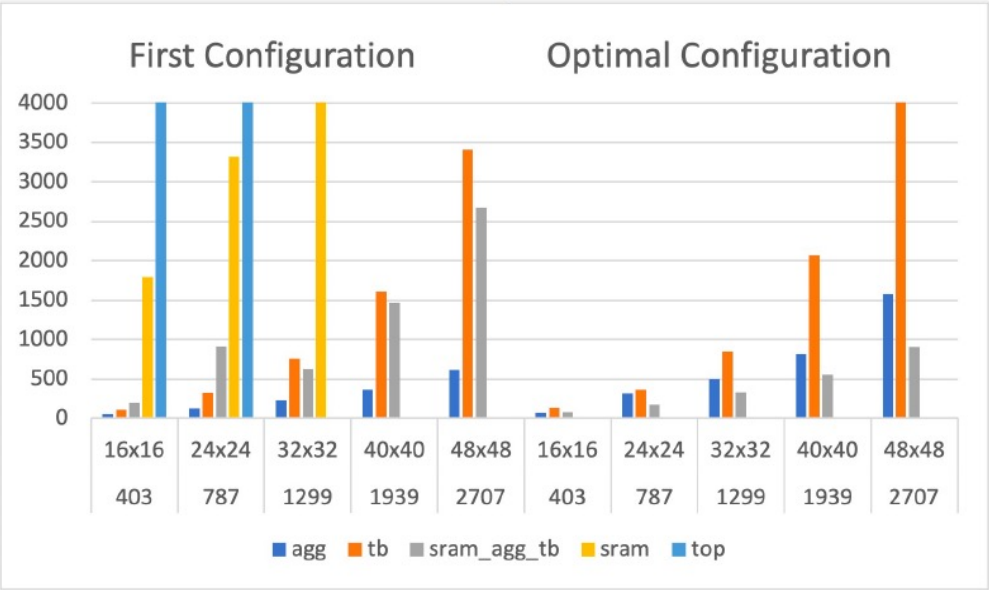
Timeout: 4000 sec
Memout: 100 GB.



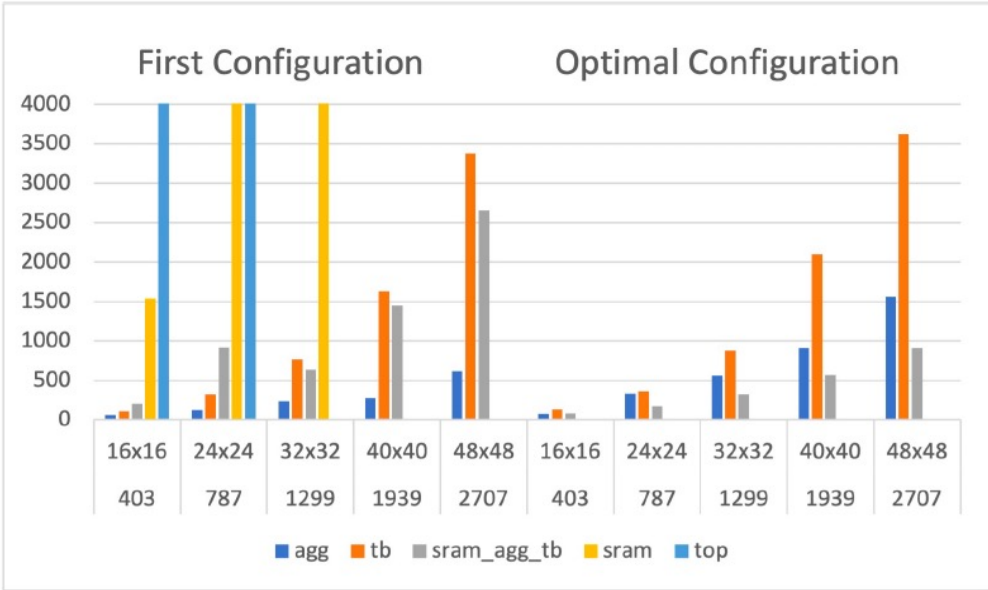
(a) Identity Stream



(b) 3x3 Convolution



(c) Cascade (conv)



(d) Harris (lxx)

Summary

- Our new approach provides a general framework for automatically configuring systems representable as transition systems.
- Key contributions include the ability to leverage modularity and the use of optimization.
- Optimal configurations are more human-understandable.
- It works well for our CGRA memory tile.
- Both modularity and optimization can improve scalability:
- These results suggest that modular configuration with optimization may be the best strategy in practice.