Propelling SAT-based Debugging using Reverse Domination

Bao Le, Hratch Mangassarian, Brian Keng, Andreas Veneris

University of Toronto

Outline

Introduction	• SAT-based Design Debugging • Motivation and Previous Work
Non-Solution Implications	• Dominators and Reverse Dominators • Non-Solution Implications from Reverse Domination Relationships
SAT Branching Scheme for Early Non-Solution Learning	• SAT Branching Scheme • Non-Solution Detection
Results and Final Remarks	• Experimental Results

Outline

Introduction	• SAT-based Design Debugging • Domination Relationships
Non-Solution Implications	• Dominators and Reverse Dominators • Non-Solution Implications from Reverse Domination Relationships
SAT Branching Scheme for Early Non-Solution Learning	• SAT Branching Scheme • Non-Solution Detection
Results and Final Remarks	• Experimental Results

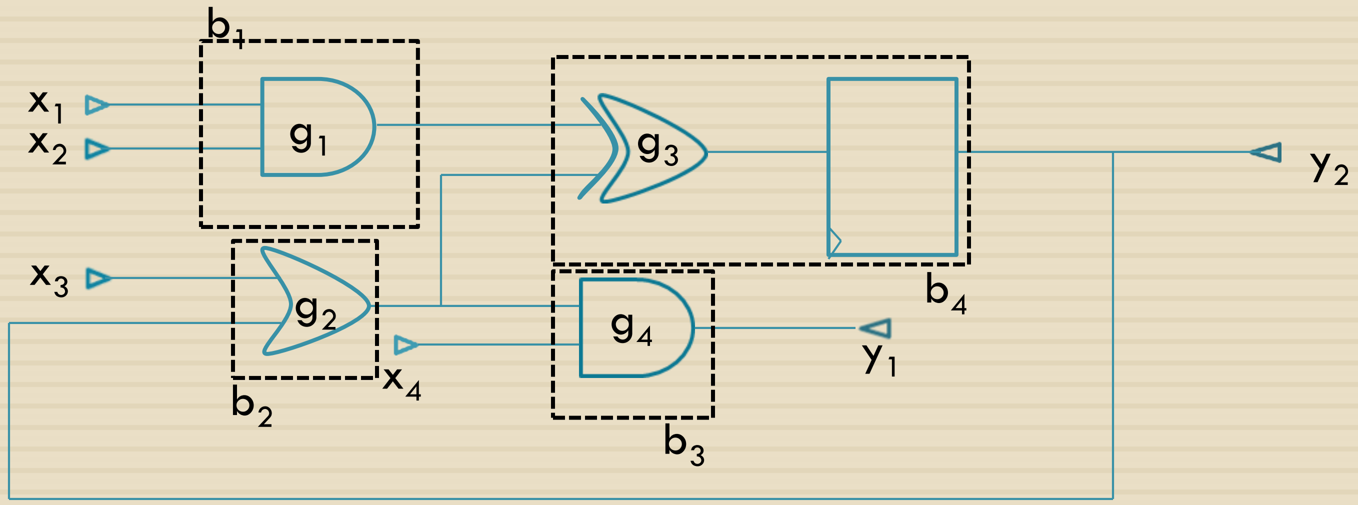
SAT-based Design Debugging

Given an erroneous circuit, a counter example of length k, and error cardinality N:

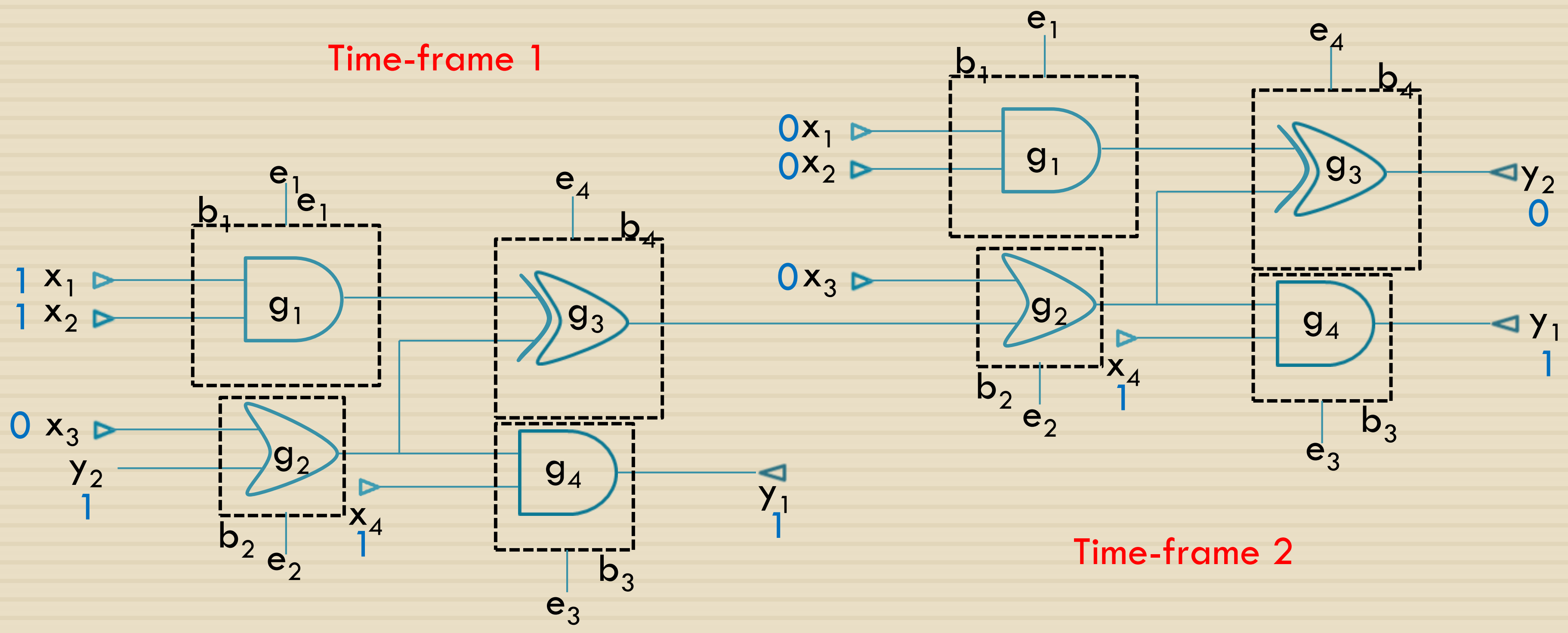
- **Goal**: Return shortlist of potentially buggy RTL blocks (**solutions**)
  - Blocks that can be modified to fix counter-example
- **Procedure:**
  - An *error-select* variable e_i is inserted at the outputs of each RTL block.
    - $e_i = 1$ disconnects block from fan-ins, making its outputs free variables
    - $e_i = 0$ does not modify the circuit
  - Enhanced circuit is replicated k times using time-frame expansion.
  - Initial state, primary inputs and outputs are constrained to expected behavior of counter-example.
  - Each satisfying assignment to $e = \{e_1, \dots, e_n\}$ is a debugging **solution**
  - The SAT solver must find all such assignments to e using blocking clauses.

SAT-based Design Debugging

- Example:



SAT-based Design Debugging



SAT Solver returns $e_4 = 1$ for $N = 1$; therefore, block b_4 (i.e. gate g_3) is the bug.

SAT-based Design Debugging

- SAT-based Design Debugging
  - Fault diagnosis and logic debugging using Boolean Satisfiability [Smith, Veneris, Ali, Viglas-TCAD2005]
- Large designs, long counter-examples pose a scalability challenge even to modern SAT solvers.
- Our contributions:
  - *On-the-fly discovery of implied non-solution blocks using reverse domination*
  - Goal is to ***prune the search space*** of design debugging
    - ***1.7x*** speed up in SAT solving time.

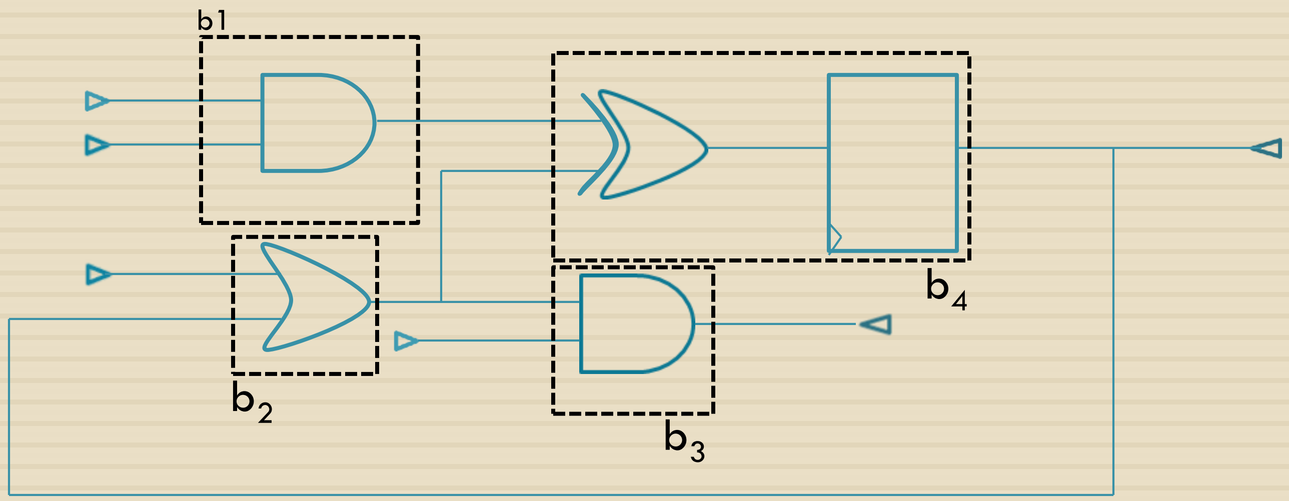
Outline

Introduction	• SAT-based Design Debugging • Motivation and Previous Work
Non-Solution Implications	• Dominators and Reverse Dominators • Non-Solution Implications from Reverse Domination Relationships
SAT Branching Scheme for Early Non-Solution Learning	• SAT Branching Scheme • Non-Solution Detection
Results and Final Remarks	• Experimental Results

Outline

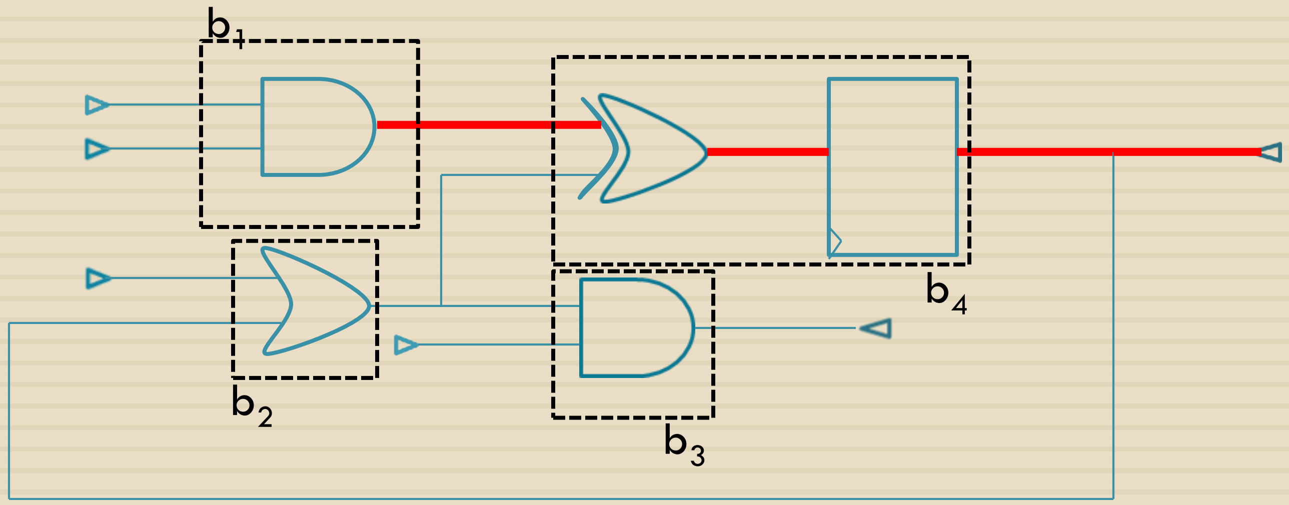
Dominators

- Block b_j is said to dominate block b_i if any path from a node in b_i to a primary output passes through a node in b_j.



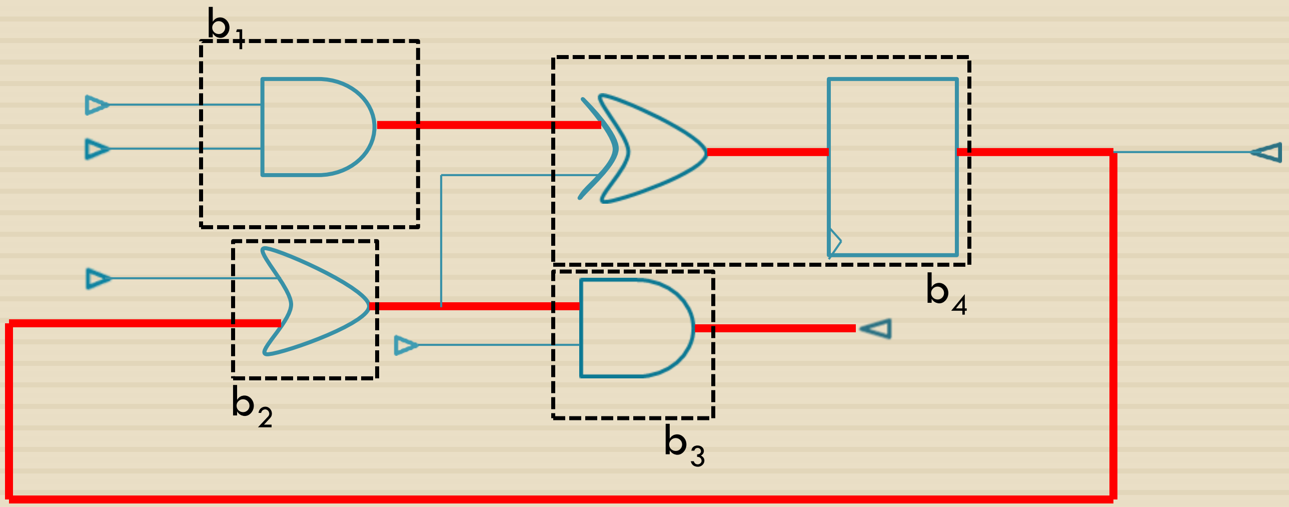
Dominators

- Block b_j is said to dominate block b_i if any path from a node in b_i to a primary output passes through a node in b_j.



Dominators

- Block b_j is said to dominate block b_i if any path from a node in b_i to a primary output passes through a node in b_j.

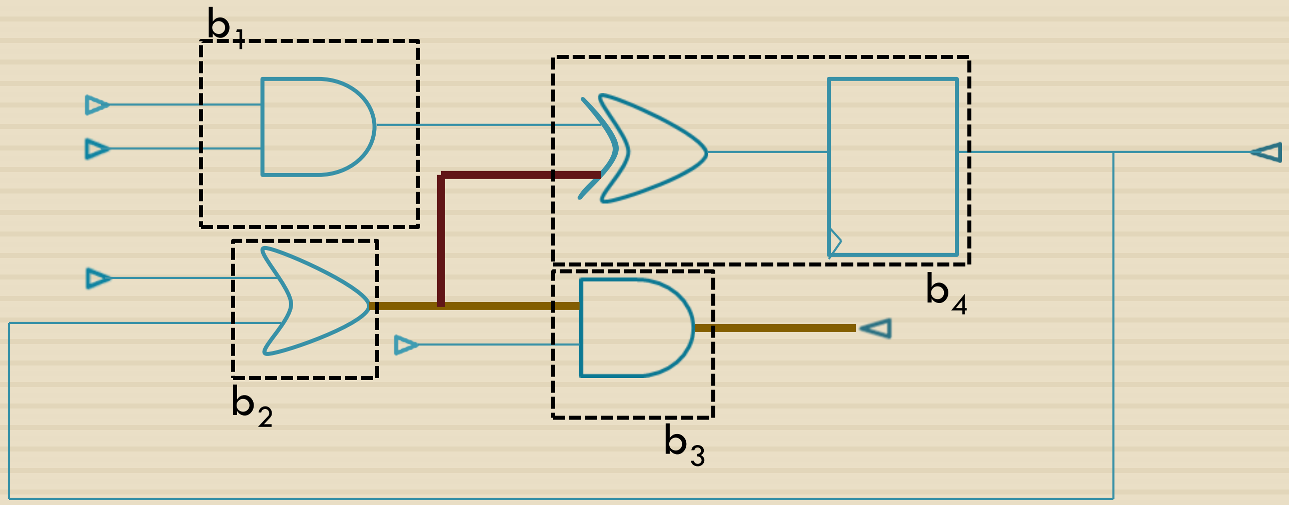


- Theorem [Mangassarian, Veneris, Smith, Safarpour-ICCAD'11]:
  - If b_j is a solution block, and b_i dominates b_j, then b_i is also a solution block

b_4 dominates b_1

Dominators

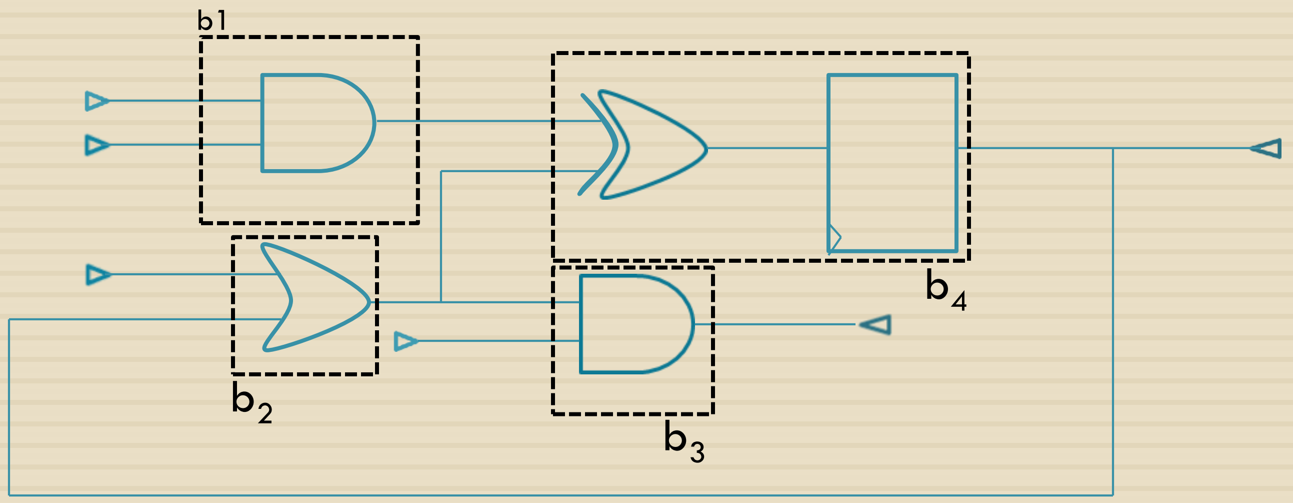
- Block b_j is said to dominate block b_i if any path from a node in b_i to a primary output passes through a node in b_j.



No block dominates b_2

Reverse Dominators

- A block b_i is a reverse dominator of block b_j if and only if b_j dominates b_i, denotes $b_i D^{-1} b_j$.

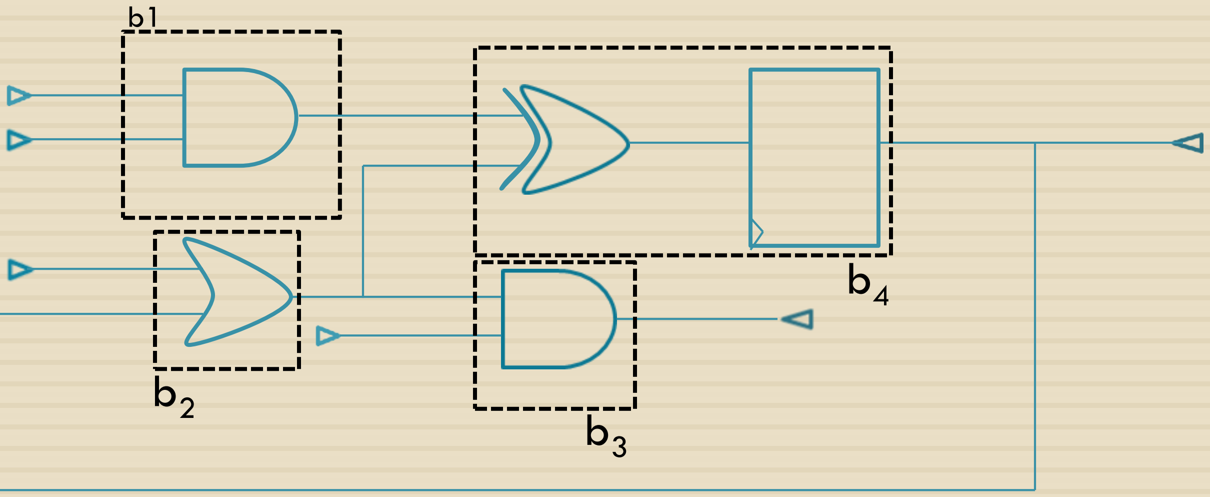


Block b_1 is a reverse dominator of b_4

Non-solution Implications

Definition: *Block b_i is a non-solution block iff $e_i = 0$ for all satisfying assignments.*

- Theorem:
  - If b_j is a non-solution block, and $b_i D^{-1} b_j$, then b_i is also a non-solution block



If b_4 is a non-solution block, b_1 is also a non-solution block. $\longrightarrow$ But how would we know that b_4 is a non-solution in the first place?

Outline

Outline

Introduction
- SAT-based Design Debugging
- Motivation and Previous Work

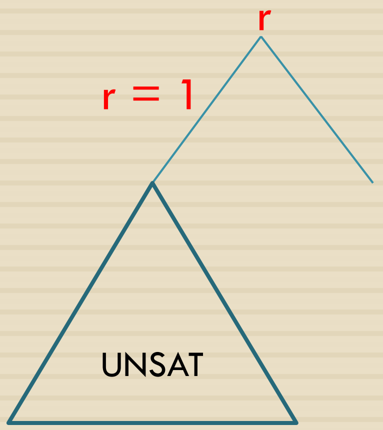
Non-Solution Implications
- Dominators and Reverse Dominators
- Non-Solution Implications from Reverse Domination Relationships

SAT Branching Scheme for Early Non-Solution Learning
- SAT Branching Scheme
- Non-Solution Detection

Results and Final Remarks
- Experimental Results

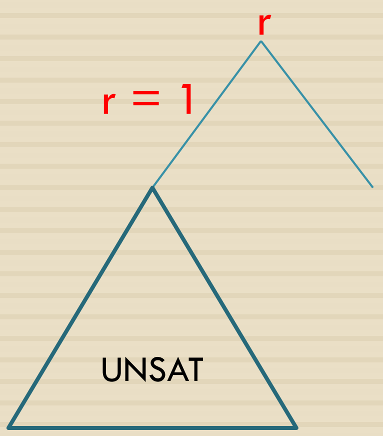
SAT Branching Scheme

- A decision tree in a SAT solver gives the order in which variables are decided upon. Consider the decision tree:



SAT Branching Scheme

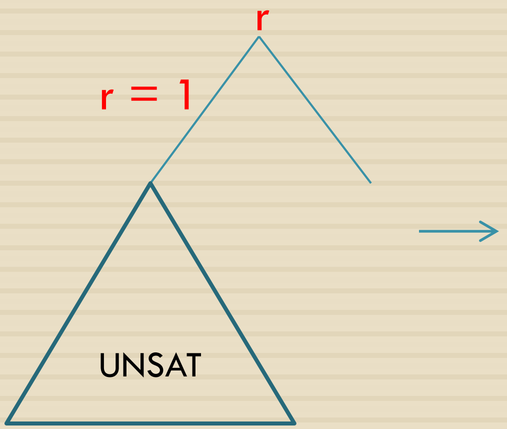
- A decision tree in a SAT solver gives the order in which variables are decided upon. Consider the decision tree:



$\longrightarrow$ r = 0 for all satisfying assignment

SAT Branching Scheme

- A decision tree in a SAT solver gives the order in which variables are decided upon. Consider the decision tree:

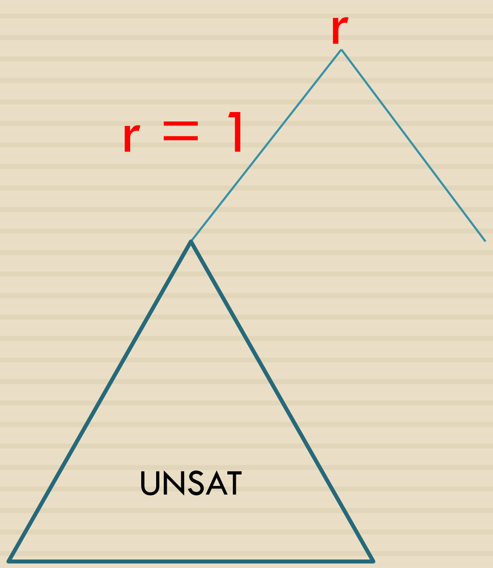


r = 0 for all satisfying assignment

If after analyzing r = 1, SAT Solver returns no satisfying assignment and starts analyzing r = 0, clearly r = 0 for any satisfying assignment (if one exists).

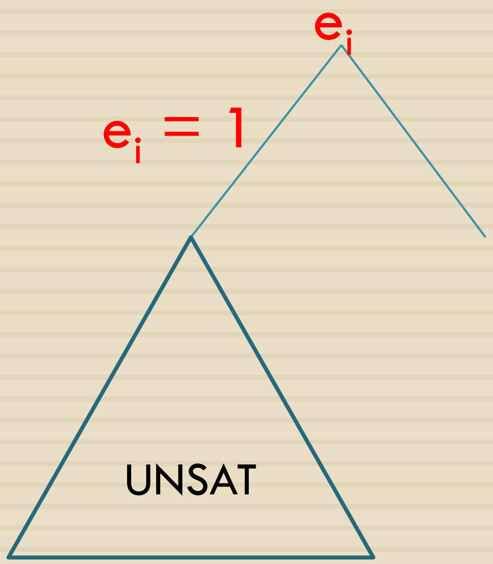
Non-Solution Detection

- What we have so far:



Non-Solution Detection

- What about:



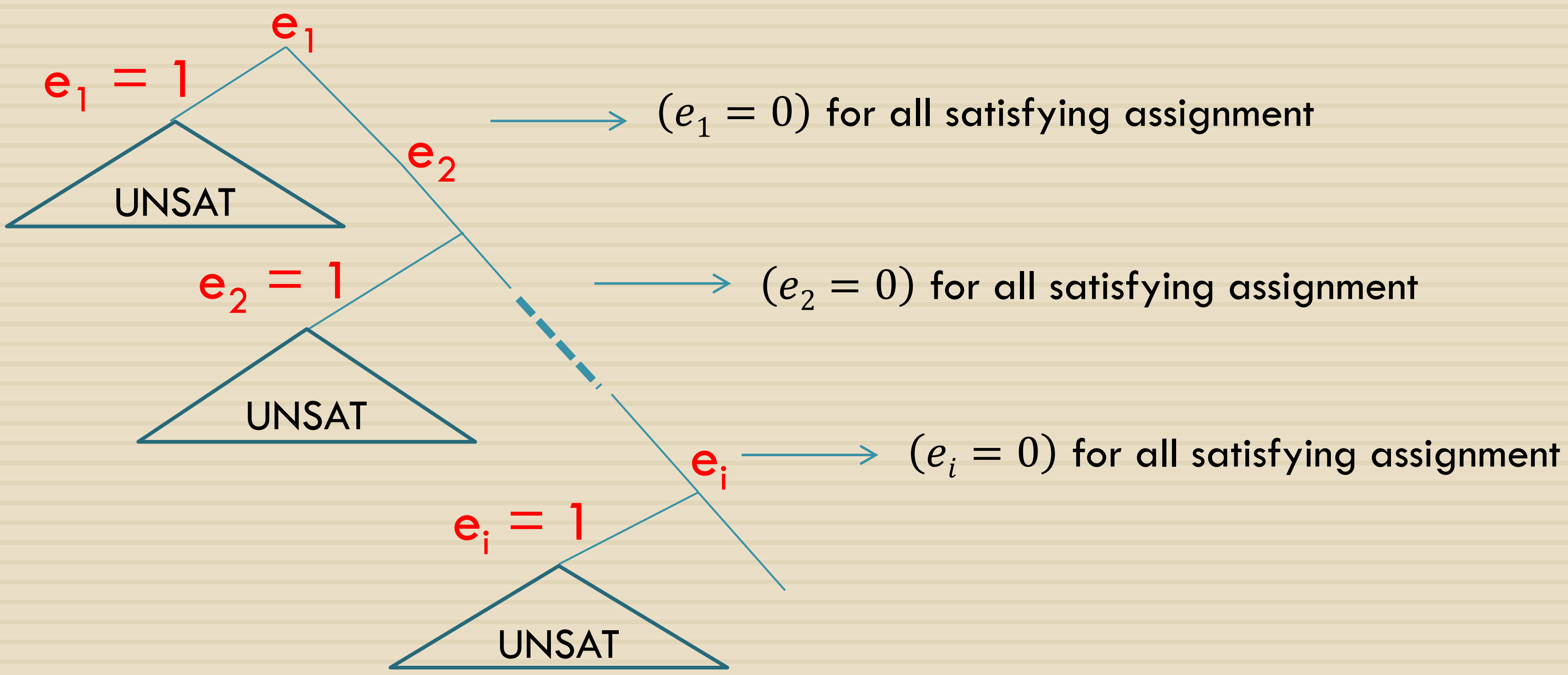
$\longrightarrow$ $e_i = 0$ for all satisfying assignments

$\downarrow$

b_i is a non-solution block.

Non-Solution Detection

- In general, we can incrementally detect non-solution blocks. For example:



$(e_1 = 0)$ for all satisfying assignment

$(e_2 = 0)$ for all satisfying assignment

$(e_i = 0)$ for all satisfying assignment

- $e_2, \ldots\ e_i$ are also detected as non-solution blocks even though they are not the root of the decision tree.

Non-Solution Detection

- Deciding on the error-select variables first forces the SAT solver to learn about them faster
- Pruning using non-solution implications can have a stronger effect

Algorithm Overview

- Rearrange the order such that error select variables e appear first in the decision tree.
- Extract learned non-solution blocks by inspecting the decision tree.
- Use reverse domination relationships to learn more non-solution blocks. Add a blocking clause for each implied non-solution block.

Outline

Outline

Introduction

- SAT-based Design Debugging
- Motivation and Previous Work

Non-Solution Implications

- Dominators and Reverse Dominators
- Non-Solution Implications from Reverse Domination Relationships

SAT Branching Scheme for Early Non-Solution Learning

- SAT Branching Scheme
- Non-Solution Detection

Results and Final Remarks

- Experimental Results

Experimental Results

- Platform: i5 3.1Ghz, 8GB memory, 2 hour time-limit.

- Benchmarks: Eight Opencores circuits and three industrial designs. For each, several bugs are injected to generate debugging instances.

- We modified MiniSAT 2.2.0 to implement our techniques.
  - **MiniSAT** vs. **dbgSAT**

- We compare to a state-of-the-art SAT-based debugger **with solution implications** [Mangassarian, etal-ICCAD'11]:

Experimental Results

On average, 28% of non-Solution blocks are implied

		MiniSAT(s)	Non-Sol(%)	dbgSAT(s)	Imp(x)
		T/O	74%	6955.90	∞
rsdecoder2	13564	33.35	58%	20.46	1.6x
usb_funct1	35158	53.17	21%	45.46	1.2x
		134.46	32%	117.83	1.1x
		123.89	28%	97.26	1.3x
		49.14	41%	36.90	1.3x
wb_dma3	299862	304.18	61%	182.09	1.7x
vga1	89412	434.81	13%	172.51	2.5x
vga2	89402	106.98	8.1%	147.95	0.7x
		7.97	0%	3.94	2.0x
		12.53	17%	24.67	0.5x
		11.76	0%	4.78	2.5x
		22.08	6%	13.51	1.6x

For rsdecoder, while MiniSAT times out, we are able to solve it in under two hours.

For certain cases, only rearranging the order of variables improves the performance

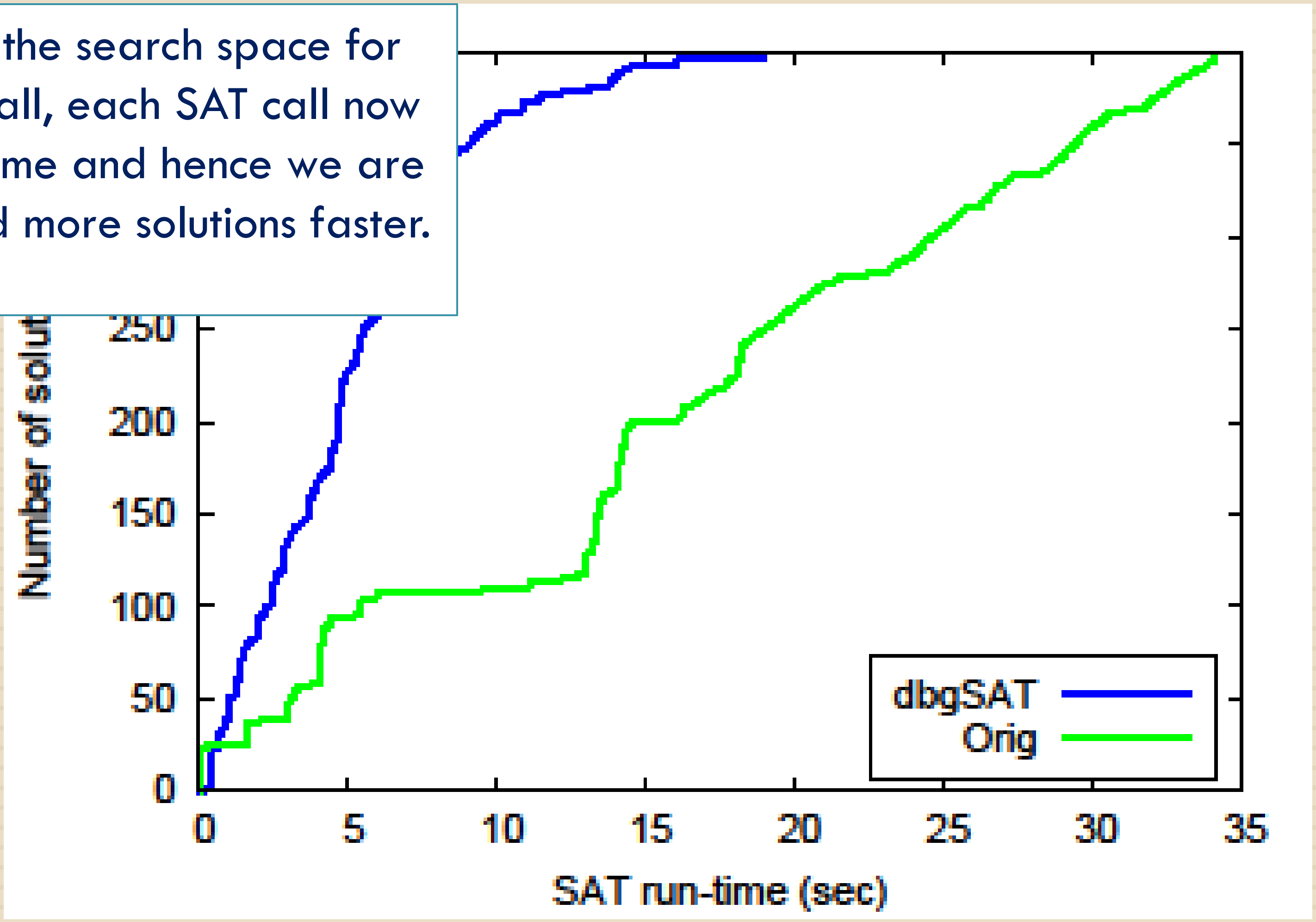
Experimental Results

23/25 cases show improvement

Instance	# of Nodes	MiniSAT	Non-Sol(%)	dbgSAT	Imp(x)
[illegible]	[illegible]	48.45	44%	33.42	1.4x
open_sparc2	64915	44.11	50%	39.39	1.1x
Design1-1	499325	53.40	0.1%	25.08	2.1x
Design1-2	499705	72.54	25%	38.27	1.9x
Design1-3	499696	39.63	1%	31.69	1.3x
Design1-4	499705	100.89	29%	45.69	2.2x
Design1-5	499705	73.72	29%	27.04	2.7x
Design2-1	45632	18.47	10%	14.59	1.3x
Design2-2	203706	7.38	0.7%	4.23	1.7x
Design2-3	2082	0.13	53%	0.08	1.6x
Design3-1	5454	3.03	51%	2.07	1.6x
Design3-2	2333	0.083	44%	0.07	1.2x
Average					1.68x

Experimental Results

By pruning the search space for each SAT call, each SAT call now takes less time and hence we are able to find more solutions faster.



Conclusions

- Summary
  - Non-solution implications using reverse domination to prune the search space of design debugging SAT calls.
  - A SAT branching scheme to detect non-solution early and enhance non-solution implications.
- Future Work
  - Study the error-select variables' order to maximize the implications (solution + non-solution).
  - Extend the work to higher cardinality.

Questions/Discussions