

Scaling VLSI Design Debugging with Interpolation

Brian Keng and Andreas Veneris



FMCAD 2009

University of Toronto

Outline

- **Introduction**

 - Motivation

 - Contributions

- **Background**

- **Debugging with Interpolation**

- **Experiments**

- **Conclusion**

Motivation

- Debugging is a major bottleneck
 - Finding root cause of error
 - Consume up to 60% of total verification time
 - **Complexity = (design size) * (# cycles)**
- Debugging is a resource intensive process
 - Manual process with GUI-based tools
 - Automated debuggers
 - e.g. Simulation, BDDs, SAT
 - Need to scale to industrial sized problems

Contributions

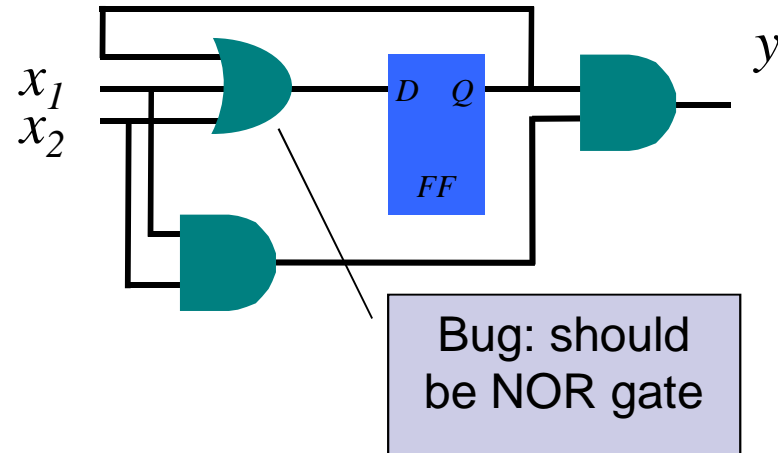
- Scalable SAT-based debugging algorithm
 - Partition trace into multiple windows and analyze each window of time-frames separately
 - Over-approximate time-frames not in current window using interpolants
 - Reduce memory usage
 - Multiple interpolants for better accuracy

Outline

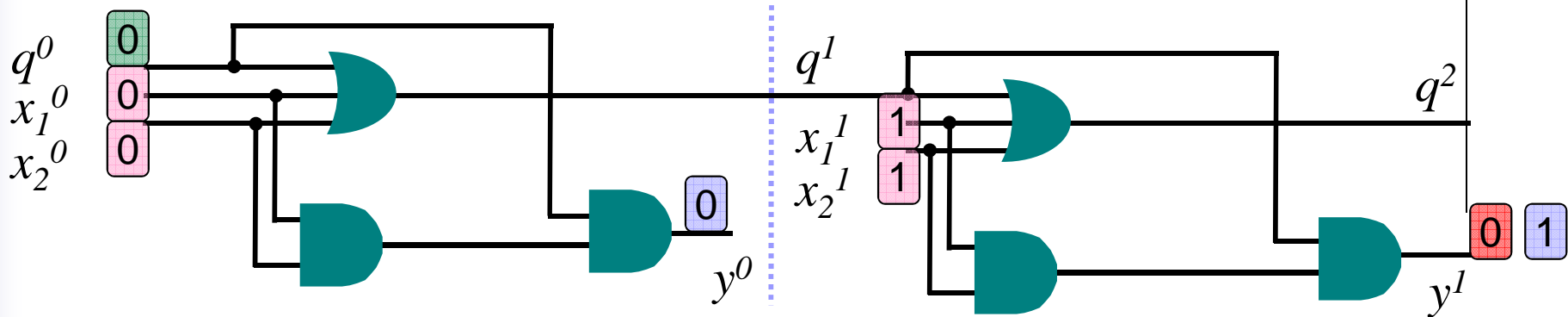
- Introduction
- **Background**
 - **Debugging**
 - **UNSAT cores and Interpolants**
- Debugging with Interpolation
- Experiments
- Conclusion

Debugging

- Erroneous Circuit
- Error Trace
 - Initial State
 - Inputs
 - Expected Output



Error!
Output Mismatch

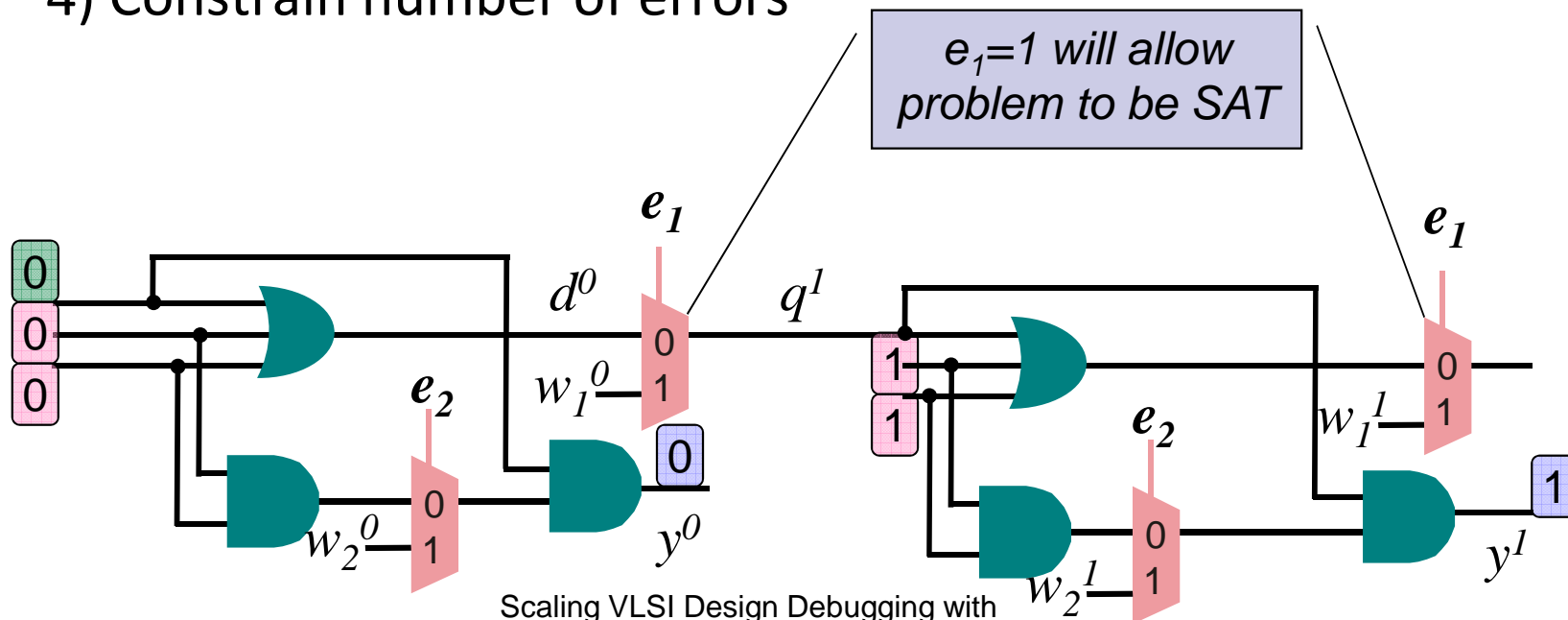
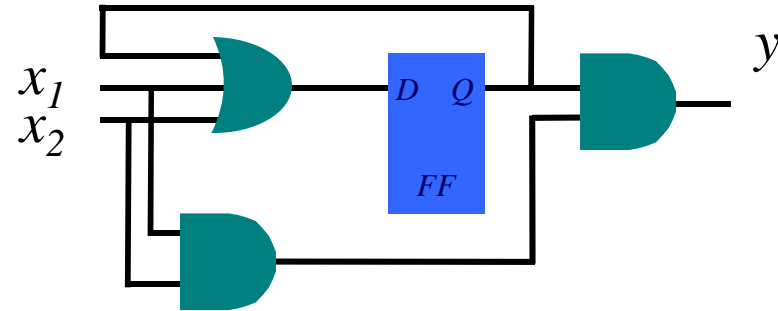


Scaling VLSI Design Debugging with Interpolation

Automated SAT-based Debugging

[Smith, et. al TCAD '05]

- Steps:
- 1) Unroll
- 2) Error modeling muxes
- 3) Constrain initial state, inputs, expected outputs
- 4) Constrain number of errors



Scaling VLSI Design Debugging with Interpolation

UNSAT Cores and Interpolants

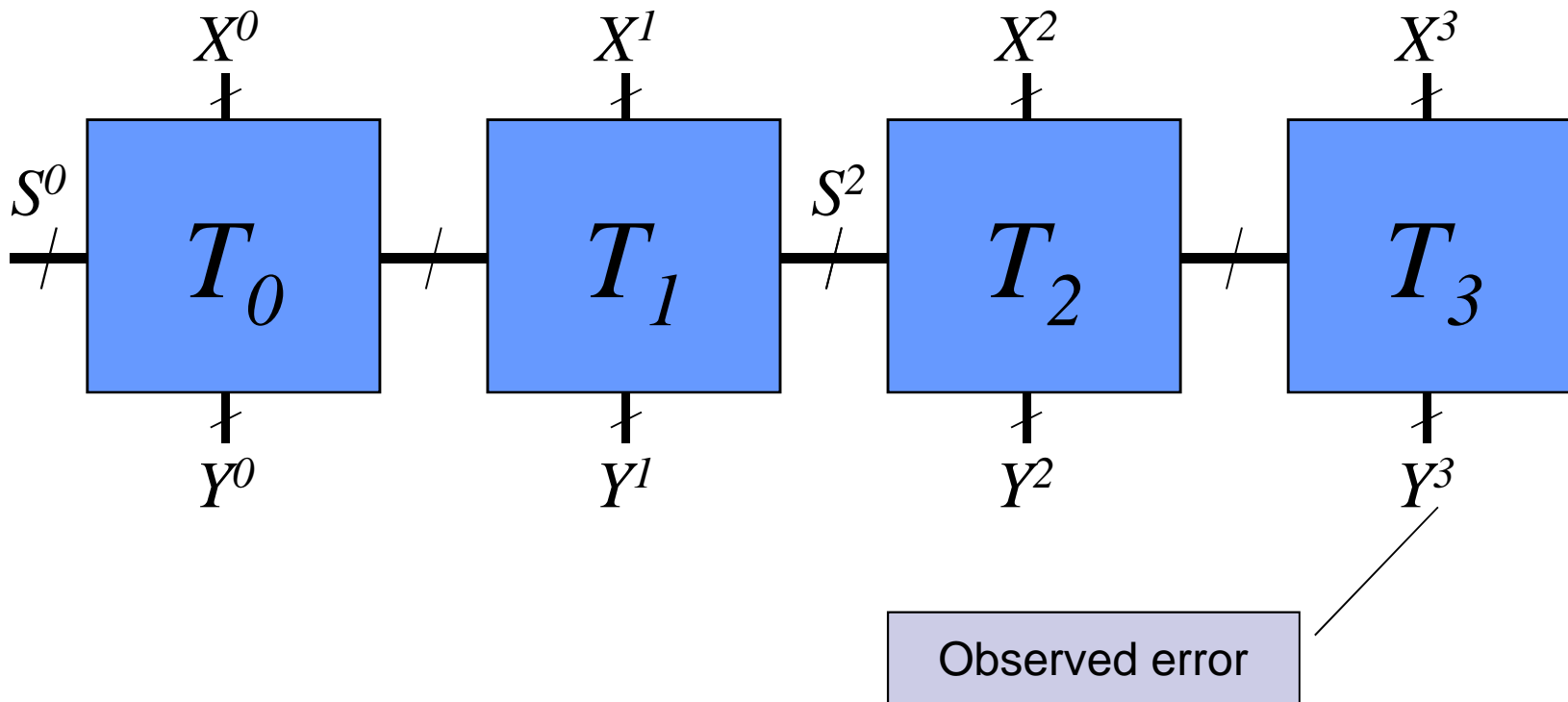
- UNSAT core
 - Subset of clauses that are unsatisfiable
 - Proof of unsatisfiability
- Interpolant P , for subsets A and B , has three properties:
 - $A \rightarrow P$
 - $B \wedge P$ is unsatisfiable
 - P only contains common variables of A and B
- Algorithm to generate an interpolant from proof of unsatisfiability in the form of a Boolean circuit [McMillan, CAV'03]

$$(a \vee b) \wedge (\bar{a} \vee \bar{b}) \wedge (a \vee c) \wedge (\bar{a} \vee c)$$
$$d \wedge (b \vee \bar{d}) \wedge (c \vee \bar{d}) \wedge (\bar{b} \vee \bar{c} \vee d)$$

Outline

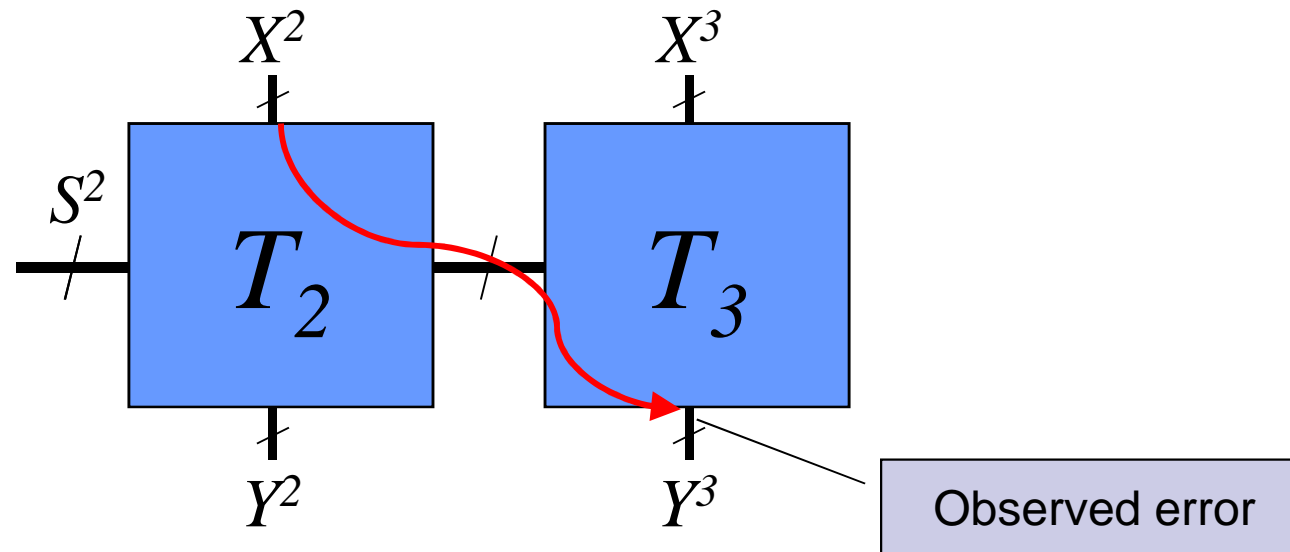
- Introduction
- Background
- **Debugging with Interpolation**
 - Suffix Window Debugging
 - UNSAT Suffix Instance
 - Prefix Window Debugging
 - Scalable Debugging Algorithm
 - Multiple Interpolants
 - Example
- Experiments
- Conclusion

Suffix Window Debugging



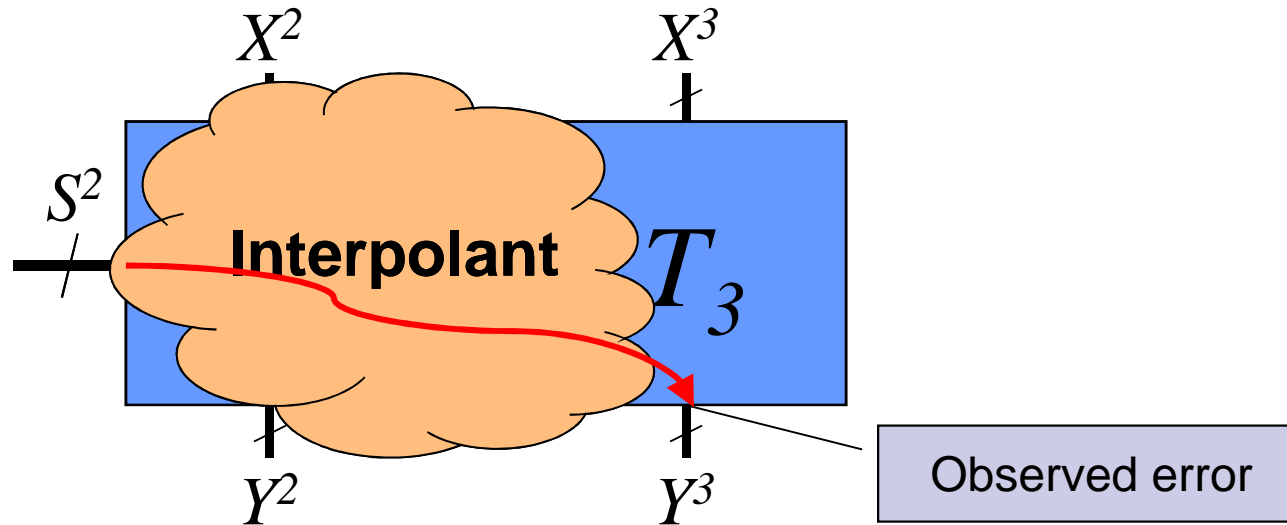
- Use only a suffix of the error trace
- Only find errors after 2nd time-frame

UNSAT Suffix Instance



- Use UNSAT suffix instance to learn information
- Case 1: UNSAT core contains no initial state variables
 - All solutions found
 - No need to analyze rest of error trace

UNSAT Suffix Instance

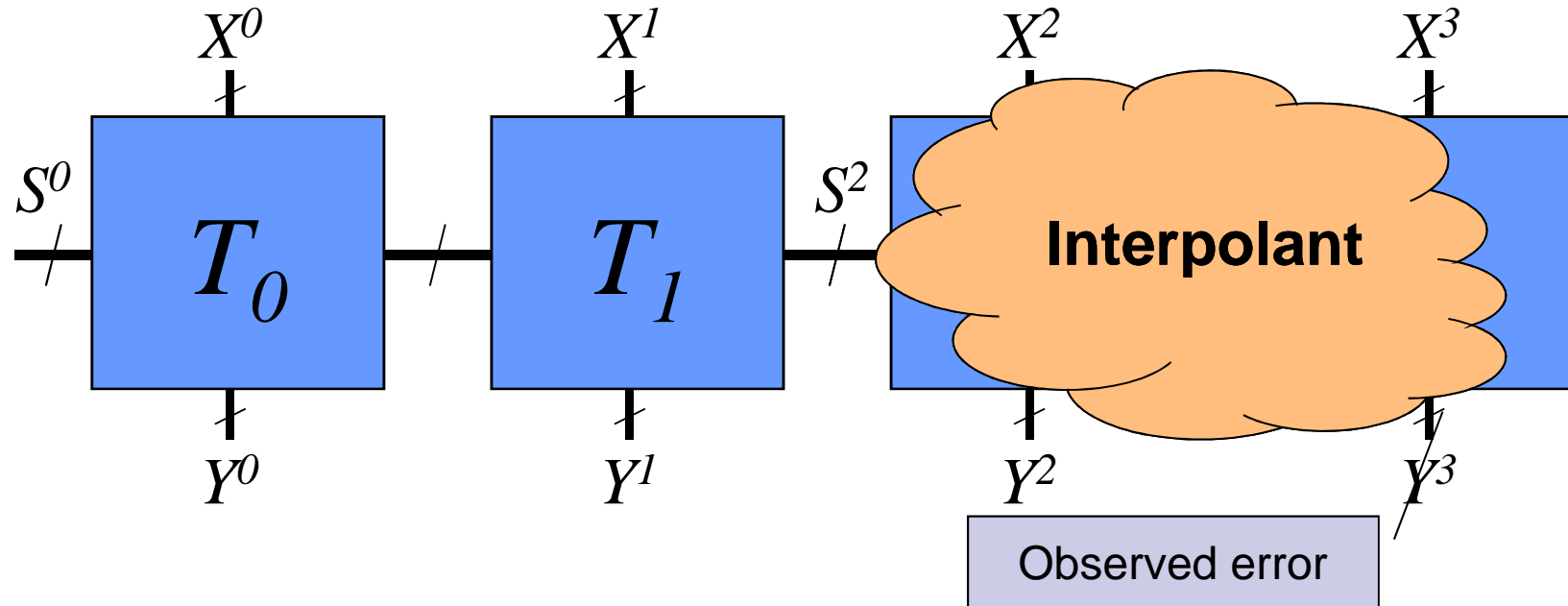


- Case 2: UNSAT core has initial state variables
 - Generate an interpolant from UNSAT instance
 - Erroneous behavior captured by interpolant
 - Interpolant is over-approximation of suffix instance

$$A = T^2 \wedge X^2 \wedge Y^2 \wedge T^3 \wedge X^3 \wedge Y^3$$

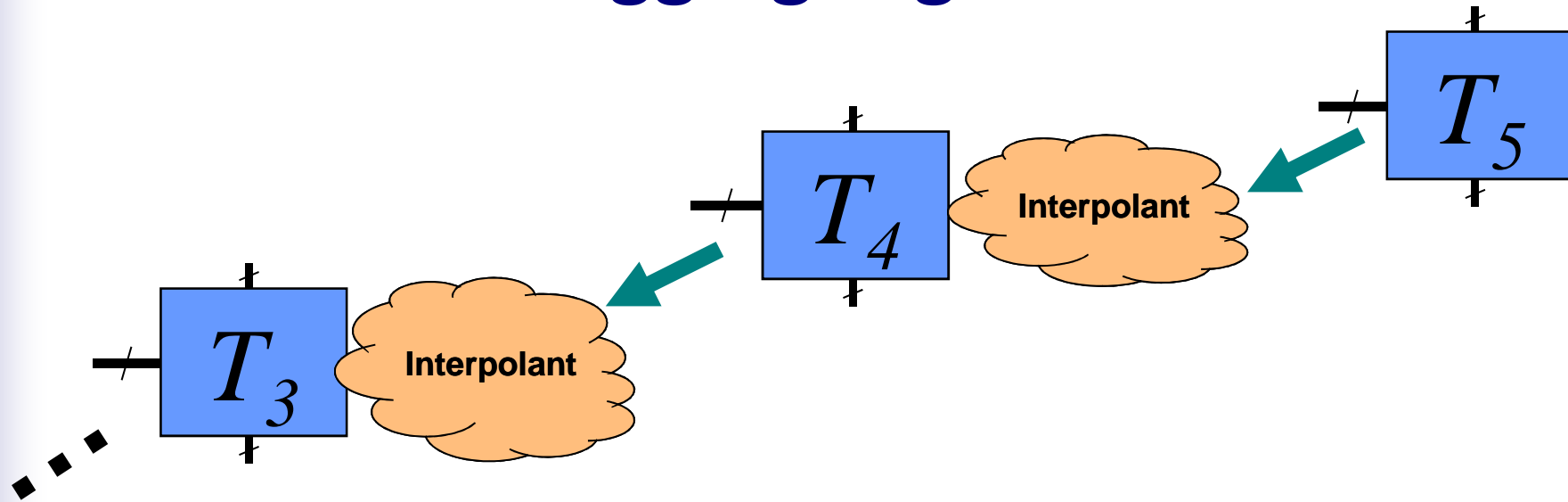
$$B = S^2 \wedge \Phi_N \wedge \textit{blocking}$$

Prefix Window Debugging



- Prefix cannot be used directly since erroneous behavior is not constrained
- Use interpolant to properly constrain erroneous behavior
- May get spurious solutions due to over-approximation

Scalable Debugging Algorithm

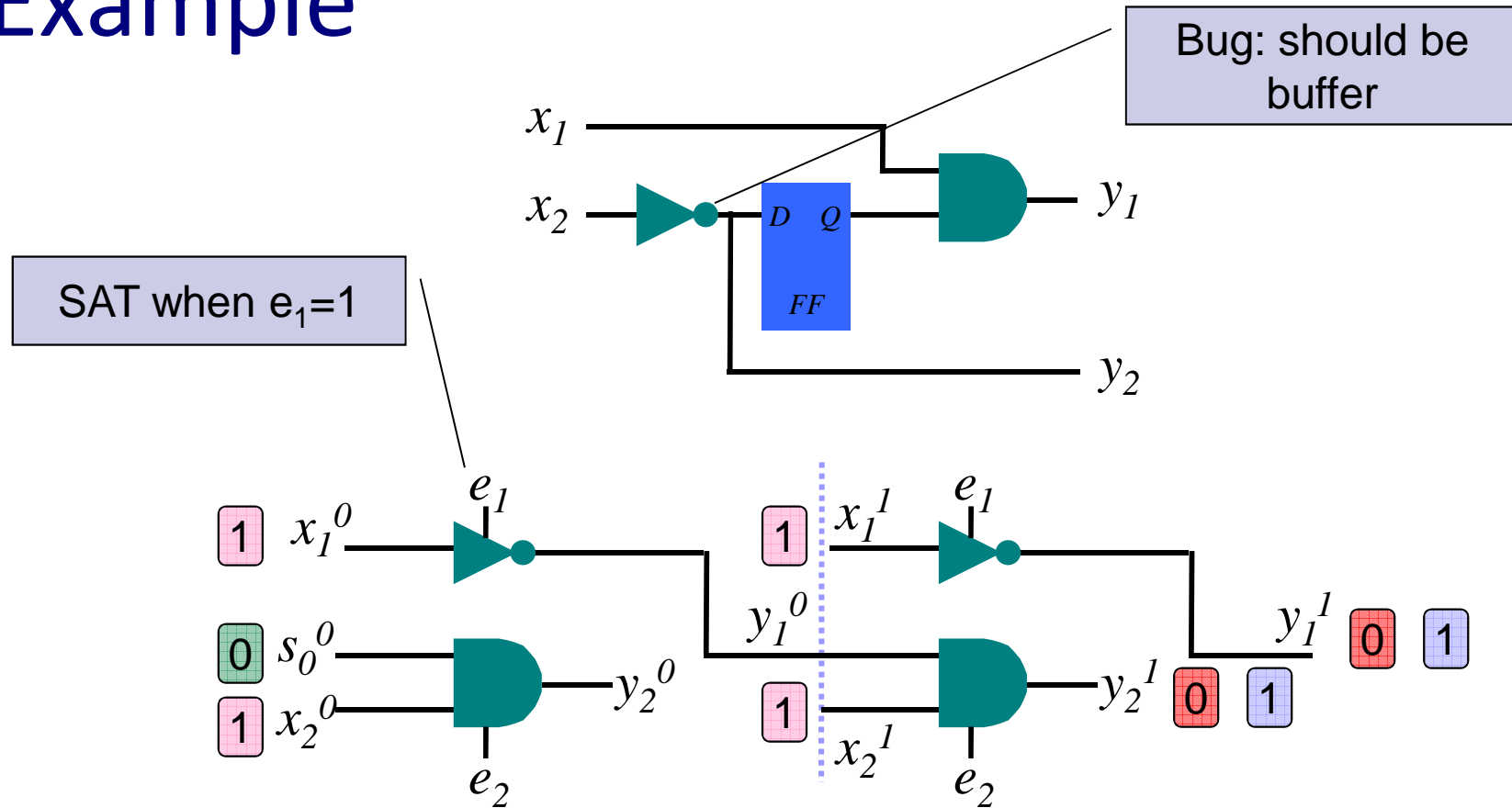


- Partition error trace into smaller windows
- Iteratively analyze each window separately
 - Use current instance to generate interpolant for next iteration
 - Limit # of simultaneous time-frames analyzed
- Each interpolant is potentially a weaker approximation than the previous one

Generating Multiple Interpolants

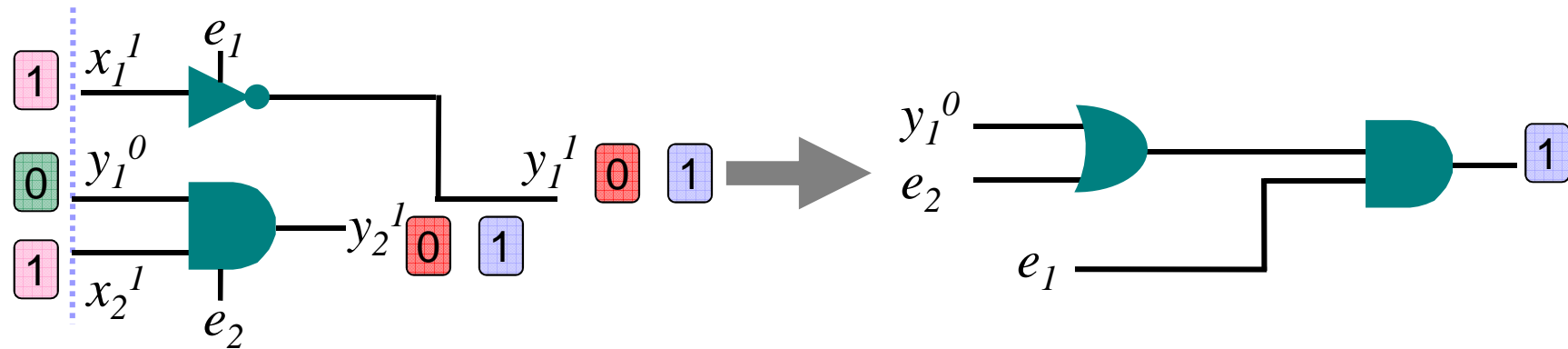
- Iteratively removing initial state variables from current instance until problem is SAT
- Using multiple interpolants will be a closer approximation to suffix
- Trade-off runtime/memory for better quality of results

Example



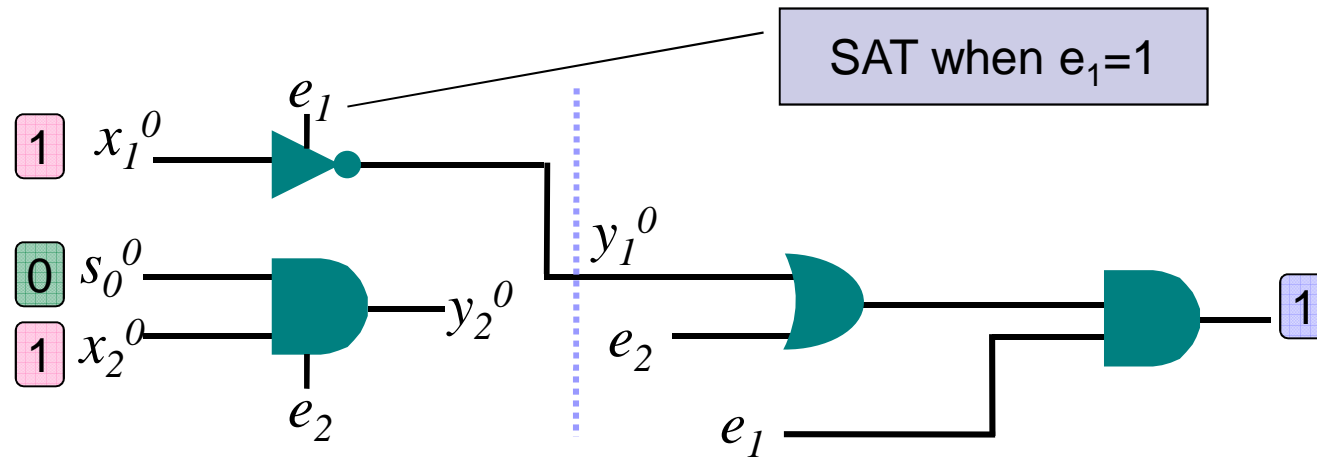
- 2 time frame error trace
- Error cardinality: $N=1$

Example: Suffix Debugging



- UNSAT with $N=1$
- Generate an interpolant from UNSAT instance
 - Over-approximation of suffix
 - Retains information about unsatisfiability

Example: Prefix Debugging



- Use interpolant to constrain prefix with erroneous behavior
- Finds all solutions as when modeling the entire error trace

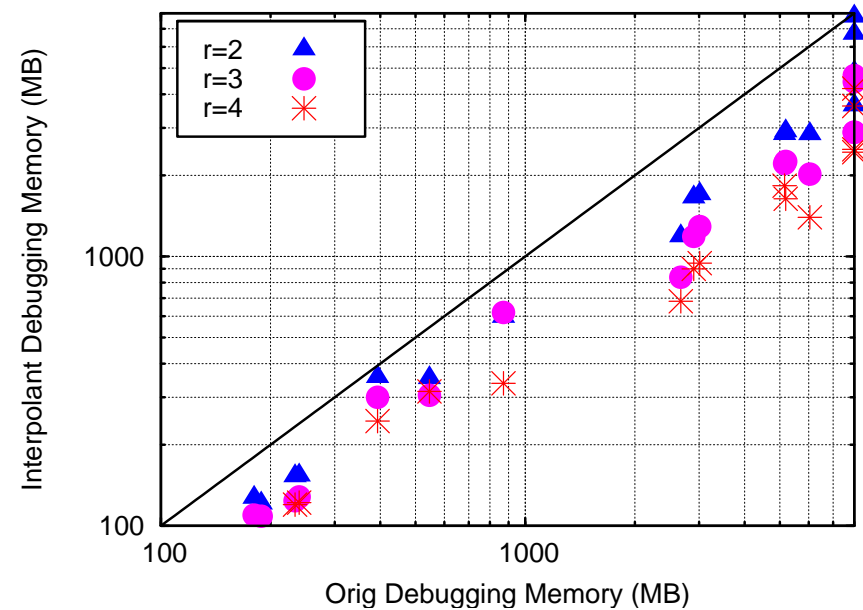
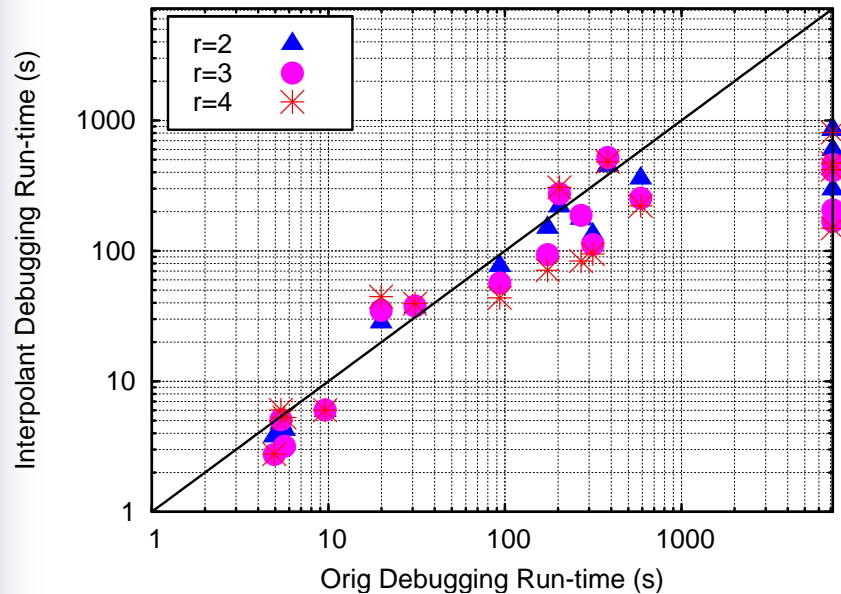
Outline

- Introduction
- Background
- Debugging with Interpolation
- **Experiments**
 - **Experimental Setup**
 - **Experimental Results**
- Conclusion

Experimental Setup

- Pentium Core 2, 2.4 Ghz workstation, 8 GB ram
- 10 circuits from OpenCores.org
- Inserted in a typical RTL error (wrong assignment, missing case statement, incorrect operator etc.)
- MiniSat 1.14 with proof logging
- r = number of windows

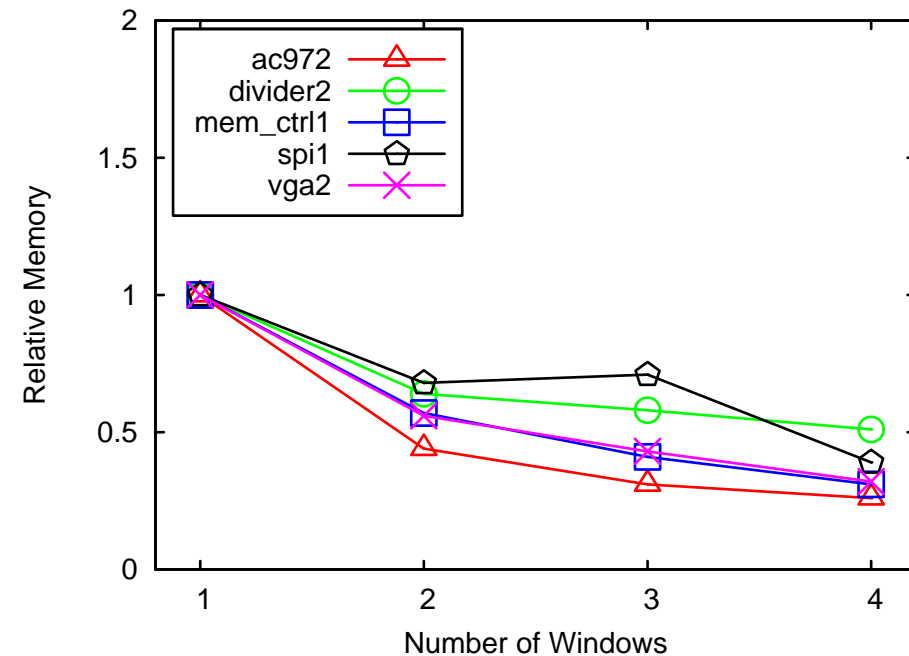
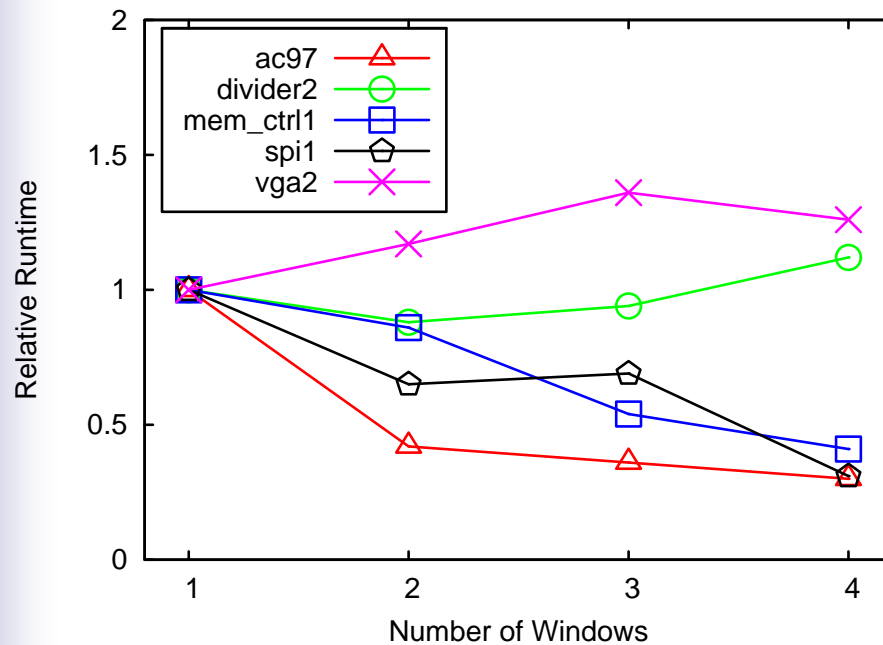
Experimental Results



■ r=4:

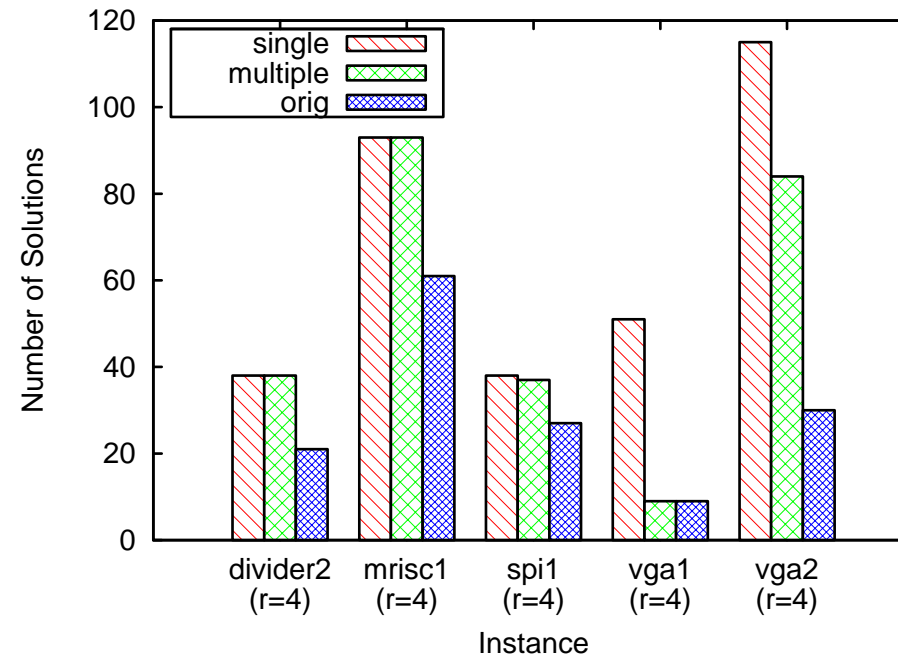
- 57% average reduction in memory
- 23% average reduction in run-time
- 2% increase number of solutions returned

Number of Windows



- Runtime does not necessarily decrease with r increases
- Peak memory decreases as r increases

Multiple Interpolants



- Instances from largest increase in number of suspects
- Improved quality in certain cases

Outline

- Introduction
- Background
- Debugging with Interpolation
- Experiments
- **Conclusion**

Conclusion

- Scalable Debugging Algorithm with Interpolation
 - Reduces number of simultaneously analyzed clock cycles by partitioning problem into multiple windows
 - Use interpolants as an over-approximation
 - Use multiple interpolants to get a better approximation
- Experimental Results
 - 57% average reduction in memory
 - 23% average reduction in run-time
 - 2% increase in suspects