CS740
Dec. 3, 1998
Special Presentation of

A Performance Study of BDD-Based Model Checking

Bwolen Yang

***Randal E. Bryant, David R. O'Hallaron,
Armin Biere, Olivier Coudert, Geert Janssen
Rajeev K. Ranjan, Fabio Somenzi***

Outline

BDD Background

- Data structure
- Algorithms

Organization of this Study

- participants, benchmarks, evaluation process

BDD Evaluation Methodology

- evaluation platform
- metrics

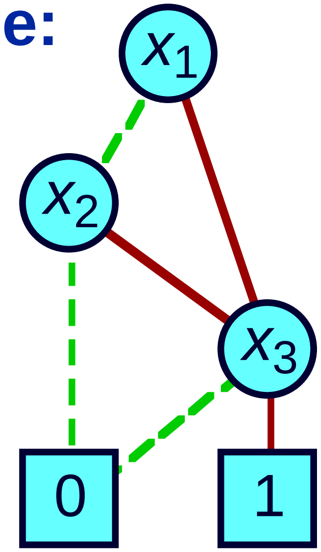
Experimental Results

- performance improvements
- characterizations of MC computations

Boolean Manipulation with OBDDs

- Ordered Binary Decision Diagrams
- Data structure for representing Boolean functions
- Efficient for many functions found in digital designs
- Canonical representation

Example:



$(x_1 \quad x_2) \& x_3$

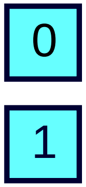
- **Nodes represent variable tests**
- **Branches represent variable values**

  Dashed for value 0

  Solid for value 1

Example OBDDs

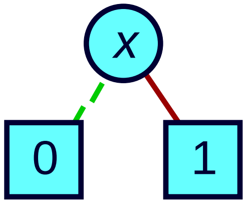
Constants



0 Unique unsatisfiable function

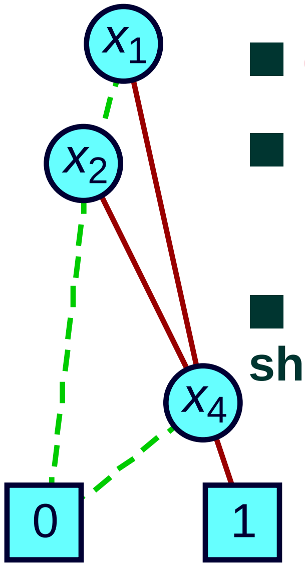
1 Unique tautology

Variable



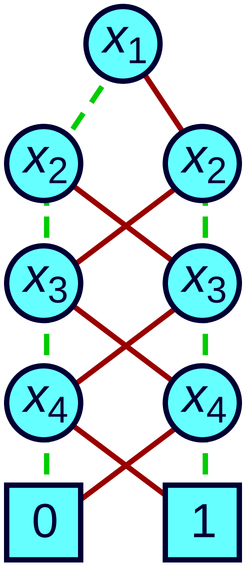
Treat variable as function

Typical Function



- $(x_1 \quad x_2) \& x_4$
- No vertex labeled x_3
  - independent of x_3
- Many subgraphs shared

Odd Parity



Linear representation

Symbolic Manipulation with OBDDs

Strategy

- **Represent data as set of OBDDs**
  - Identical variable orderings
- **Express solution method as sequence of symbolic operations**
  - Implement each operation by OBDD manipulation
  - Information always maintained in reduced, canonical form

Algorithmic Properties

- **Arguments are OBDDs with identical variable orderings.**
- **Result is OBDD with same ordering.**
- **"Closure Property"**

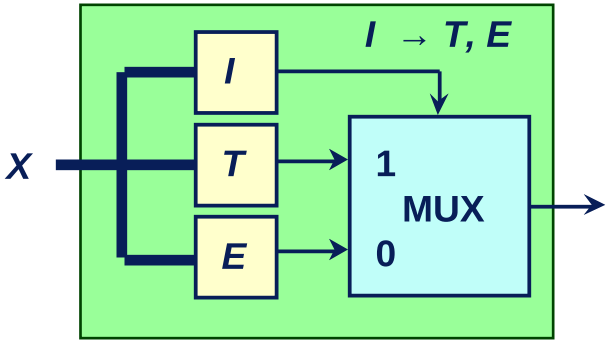
Treat as Abstract Data Type

- **User not concerned with underlying representation**

If-Then-Else Operation

Concept

- **Apply Boolean choice operation to 3 argument functions**



Arguments *I*, *T*, *E*

- Functions over variables ***X***
- Represented as OBDDs

Result

- OBDD representing composite function
- $I \ T + \neg I \ E$

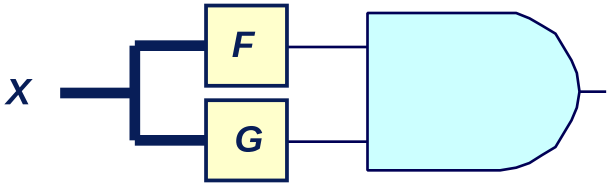
Implementation

- **Combination of depth-first traversal and dynamic programming.**
  - Maintain computed cache of previously encountered argument / result combinations
- **Worst case complexity product of argument graph sizes.**

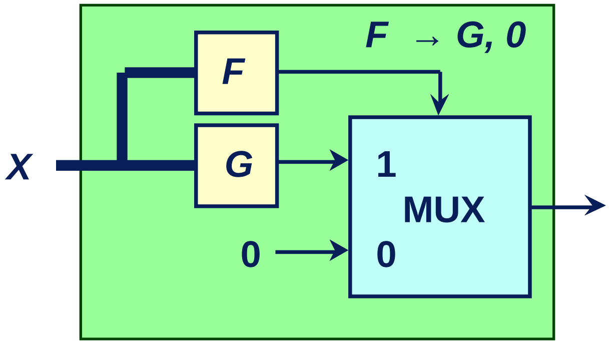
Derived Algebraic Operations

- Other common operations can be expressed in terms of If-Then-Else

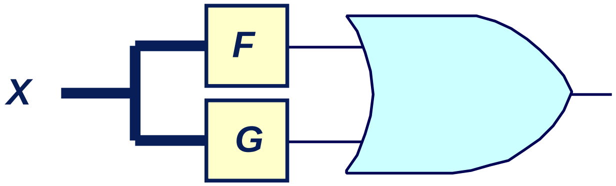
And(*F*, *G*)



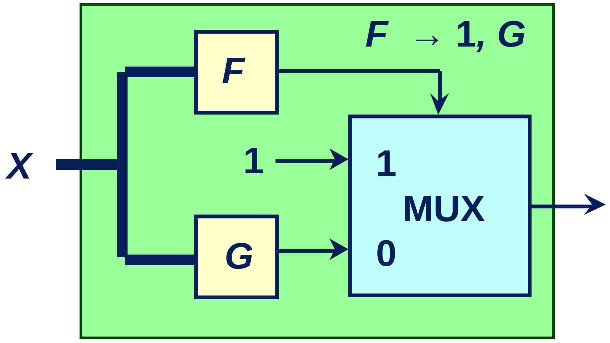
If-Then-Else(*F*, *G*, 0)



Or(*F*, *G*)



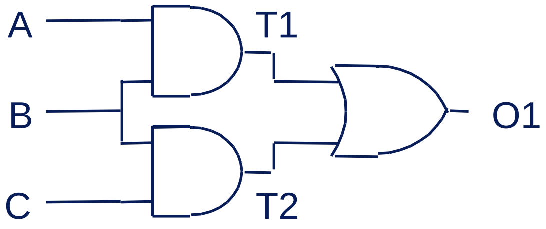
If-Then-Else(*F*, 1, *G*)



Generating OBDD from Network

Task: **Represent output functions of gate network as OBDDs.**

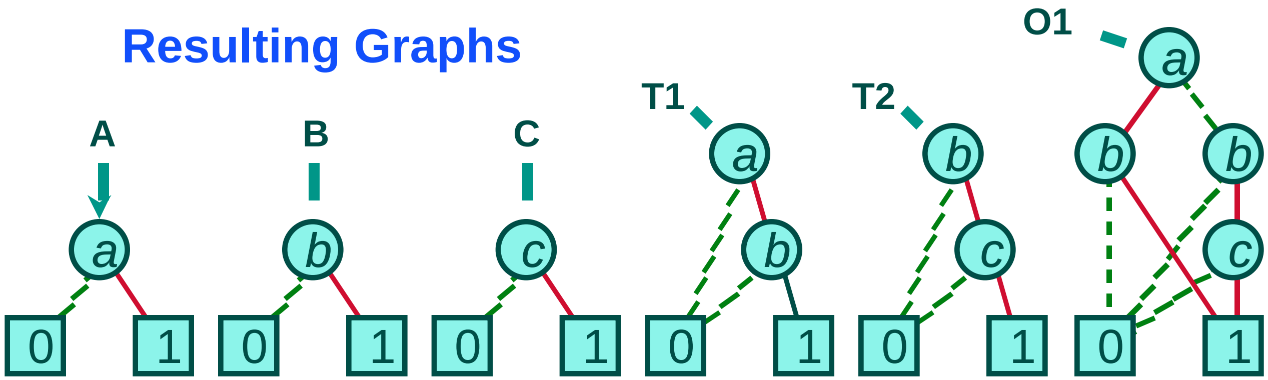
Network



Evaluation

```
A    ←  new_var ("a");
B    ←  new_var ("b");
C    ←  new_var ("c");
T1   ←  And (A, B);
T2   ←  And (B, C);
O1   ←  Or (T1, T2);
```

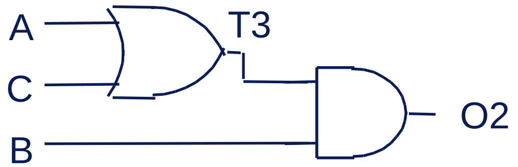
Resulting Graphs



Checking Network Equivalence

- **Determine:** Do 2 networks compute same Boolean function?
- **Method: Compute OBDDs for both networks and compare**

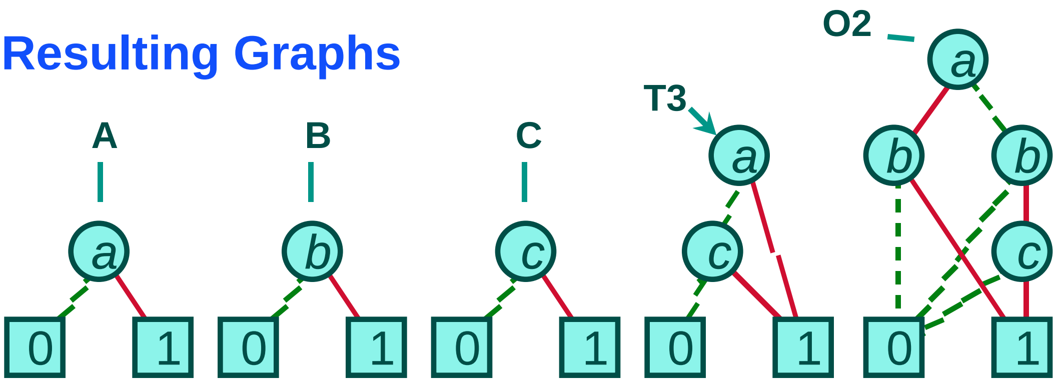
Alternate Network



Evaluation

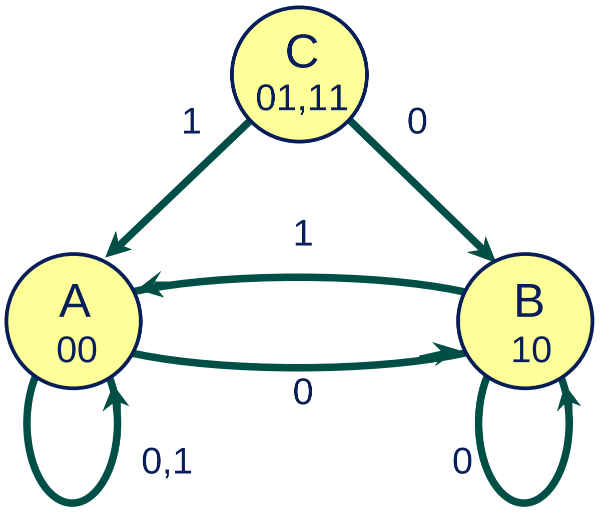
```
T3  ←  Or (A, C);
O2  ←  And (T3, B);
if (O2 == O1)
        then Equivalent
        else Different
```

Resulting Graphs

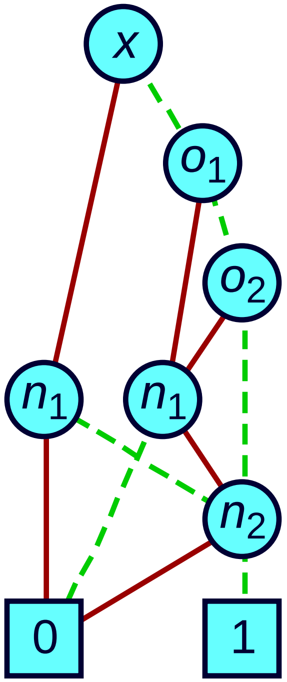


Symbolic FSM Representation

Nondeterministic FSM



Symbolic Representation



x input

o_1,o_2 encoded old state

n_1, n_2 encoded new state

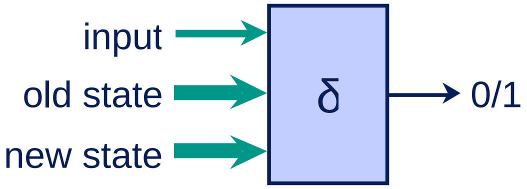
- **Represent set of transitions as function δ(*x*, *o*, *n*)**
  - Yields 1 if input *x* can cause transition from state *o* to state *n*.
- **Represent as Boolean function**
  - Over variables encoding states and inputs

Reachability Analysis

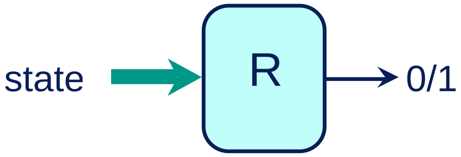
Task

- Compute set of states reachable from initial state Q0
- Represent as Boolean function **R(*s*)**.
- Never enumerate states explicitly

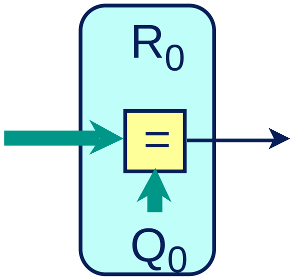
Given



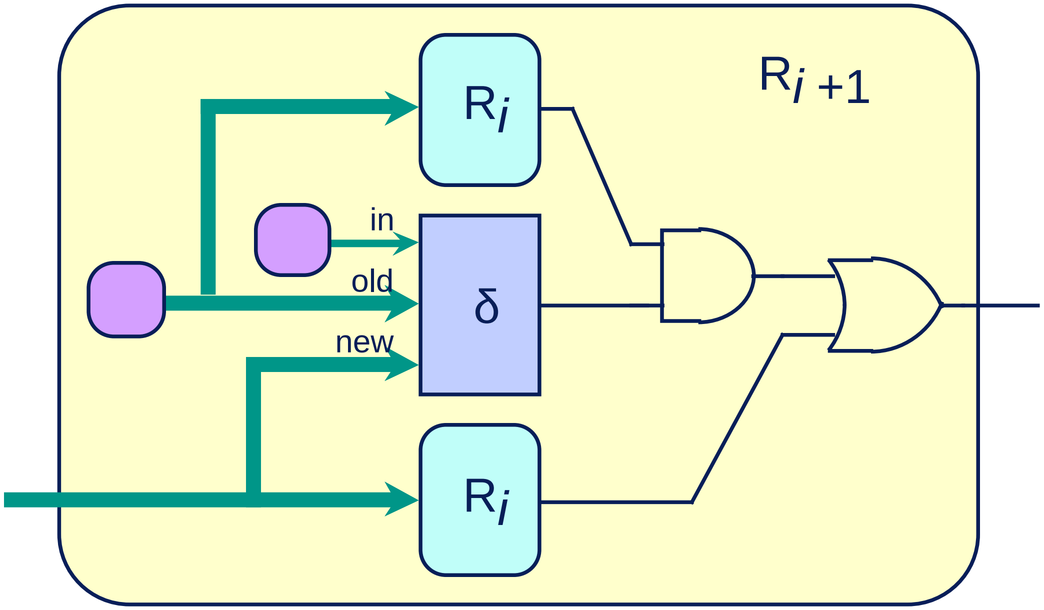
Compute



Initial



Iterative Computation

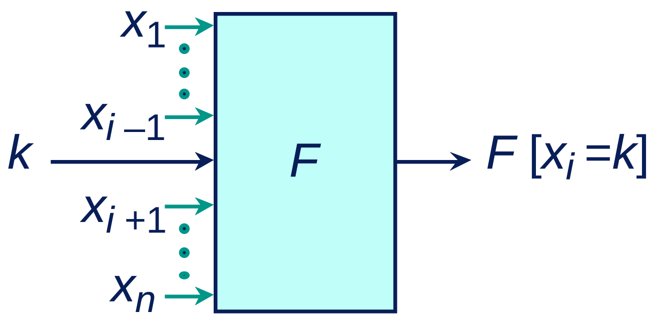


- ***R_{i+1}*** **– set of states that can be reached** ***i +1*** **transitions**
  - Either in *R_i*
  - or single transition away from some element of *R_i*
  - for some input
- **Continue iterating until** ***R_i*** **=** ***R_{i+1}***

Restriction Operation

Concept

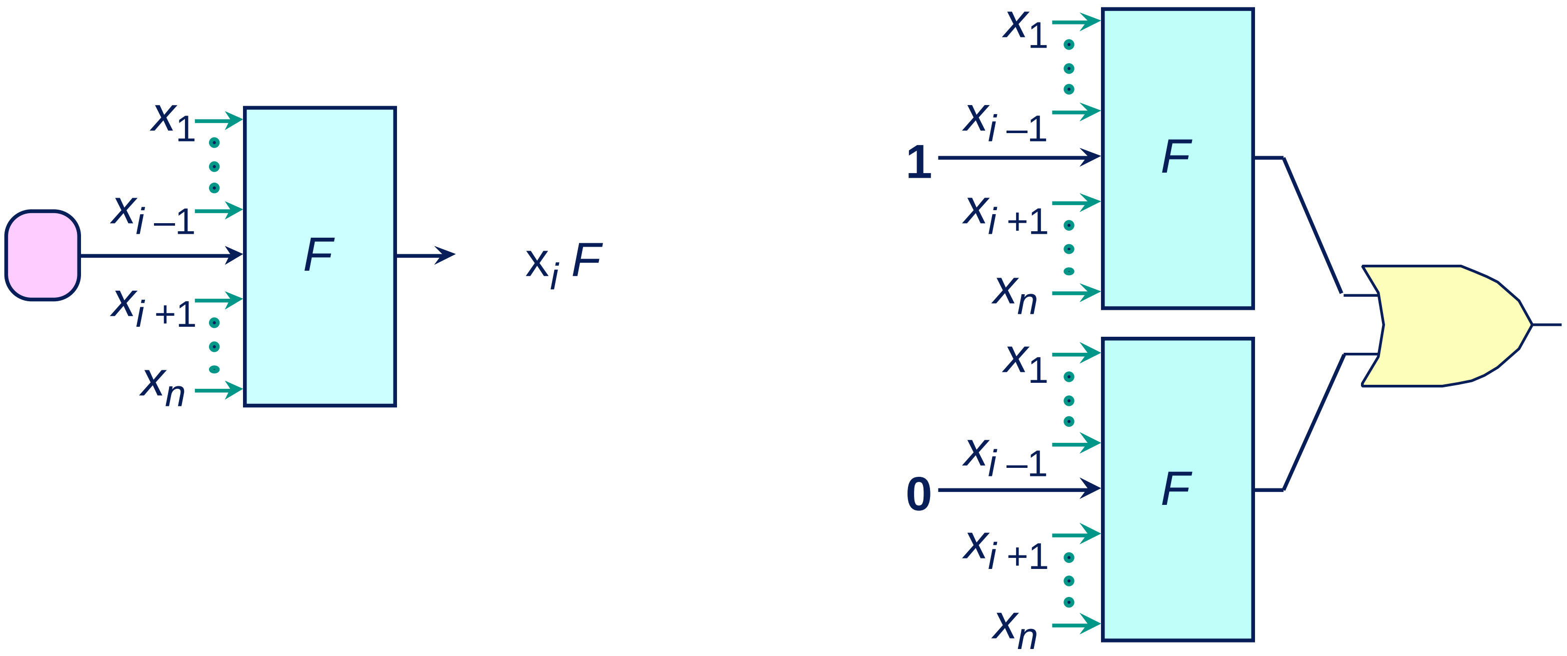
- Effect of setting function argument x_i to constant k (0 or 1).



Implementation

- Depth-first traversal.
- Complexity linear in argument graph size

Variable Quantification



- Eliminate dependency on some argument through quantification
- Same as step used in resolution-based prover
- Combine with AND for universal quantification.

Multi-Variable Quantification

Operation

- **Compute:** $\boldsymbol{X}\ \boldsymbol{F}\ (\boldsymbol{X}, \boldsymbol{Y})$
  - $\boldsymbol{X}$ Vector of bound variables $x_1, \ldots, x_n$
  - $\boldsymbol{Y}$ Vector of *free* variables $y_1, \ldots, y_m$
- **Result:**
  - Function of free variables $\boldsymbol{Y}$ only
  - yields 1 if $F(\boldsymbol{X}, \boldsymbol{Y})$ would yield 1 for some assignment to variables $\boldsymbol{X}$

Methods

- **Sequentially**
  - $x_1[\ \ x_2\,[\ldots\ \ x_n\,[F\,(\boldsymbol{X}, \boldsymbol{Y})]\ldots]]$
- **Simultaneously, by recursive algorithm over BDD for** F

Complexity

- **Each quantification can at most square graph size**
- **Typically not so bad**

Motivation for Studying Symbolic Model Checking (MC)

MC is an important part of formal verification

- digital circuits and other finite state systems
- BDDs are an enabling technology for MC

Not well studied

- Packages are tuned using combinational circuits (CC)

Qualitative differences between CC and MC computations

- CC: build outputs, constant time equivalence checking
- MC: build model, many fixed-points to verify the specs
- CC: BDD algorithms are polynomial
  - If-Then-Else algorithm
- MC: key BDD algorithms are exponential
  - Multi-variable quantification

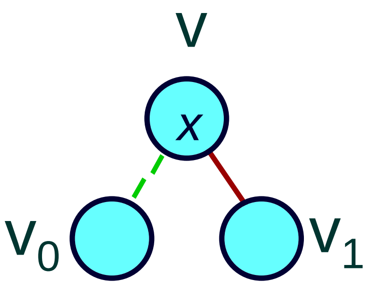
BDD Data Structures

BDD

- **Multi-rooted DAG**
  - Each root denotes different Boolean function
- **Provide automatic memory management**
  - Garbage collection based on reference counting

Unique Table

- **Provides mapping** $[x, v_0, v_1] \rightarrow v$
- **Required to make sure graph is canonical**



Computed Cache

- **Provides memoization of argument / results**
- **Reduce manipulation algorithms from exponential to polynomial**
- **Periodically flush to avoid excessive growth**

Interactions Between Data Structures

Dead Nodes

- **Reference Count → 0**
  - No references by other nodes or by top-level pointers
  - Decrement reference counts of children
    - Could cause death of entire subgraph
- **Still have invisible reference from unique table**

Garbage Collection

- **Eliminate all dead nodes**
- **Remove entries from unique table**

Rebirth

- **Possible to resurrect node considered dead**
- **From hit in unique table**
- **Must increment child reference counts**
  - Could cause rebirth of subgraph

Organization of this Study: Participants

Armin Biere: ABCD

Carnegie Mellon / Universität Karlsruhe

Olivier Coudert: TiGeR

Synopsys / Monterey Design Systems

Geert Janssen: EHV

Eindhoven University of Technology

Rajeev K. Ranjan: CAL

Synopsys

Fabio Somenzi: CUDD

University of Colorado

Bwolen Yang: PBF

Carnegie Mellon

Organization of this Study: Setup

Metrics: 17 statistics

Benchmark: 16 SMV execution traces

- **traces of BDD-calls from verification of**
  - cache coherence, Tomasulo, phone, reactor, TCAS…
- **size**
  - 6 million - 10 billion sub-operations
  - 1 - 600 MB of memory
- **Gives 6 * 16 = 96 different cases**

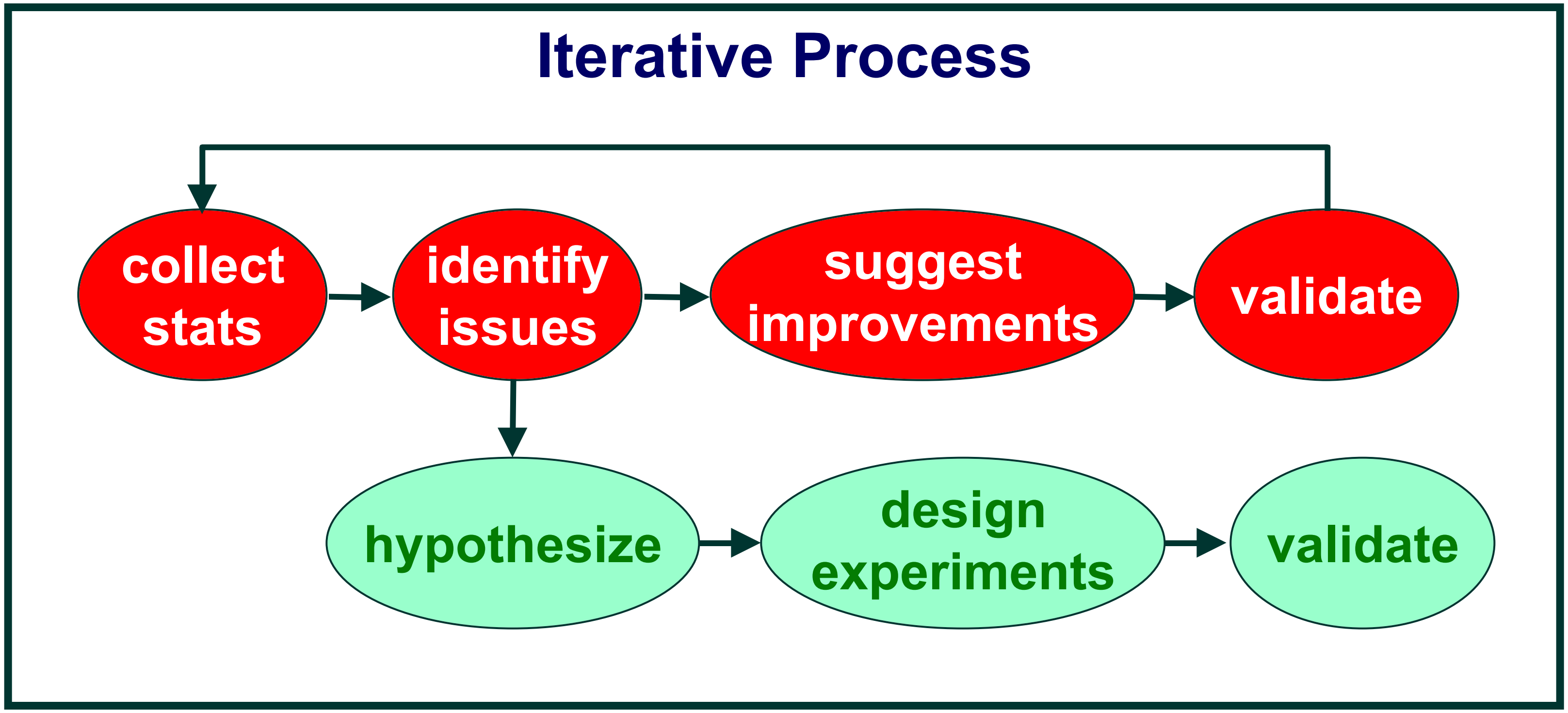
Evaluation platform: trace driver

- **“drives” BDD packages based on execution trace**

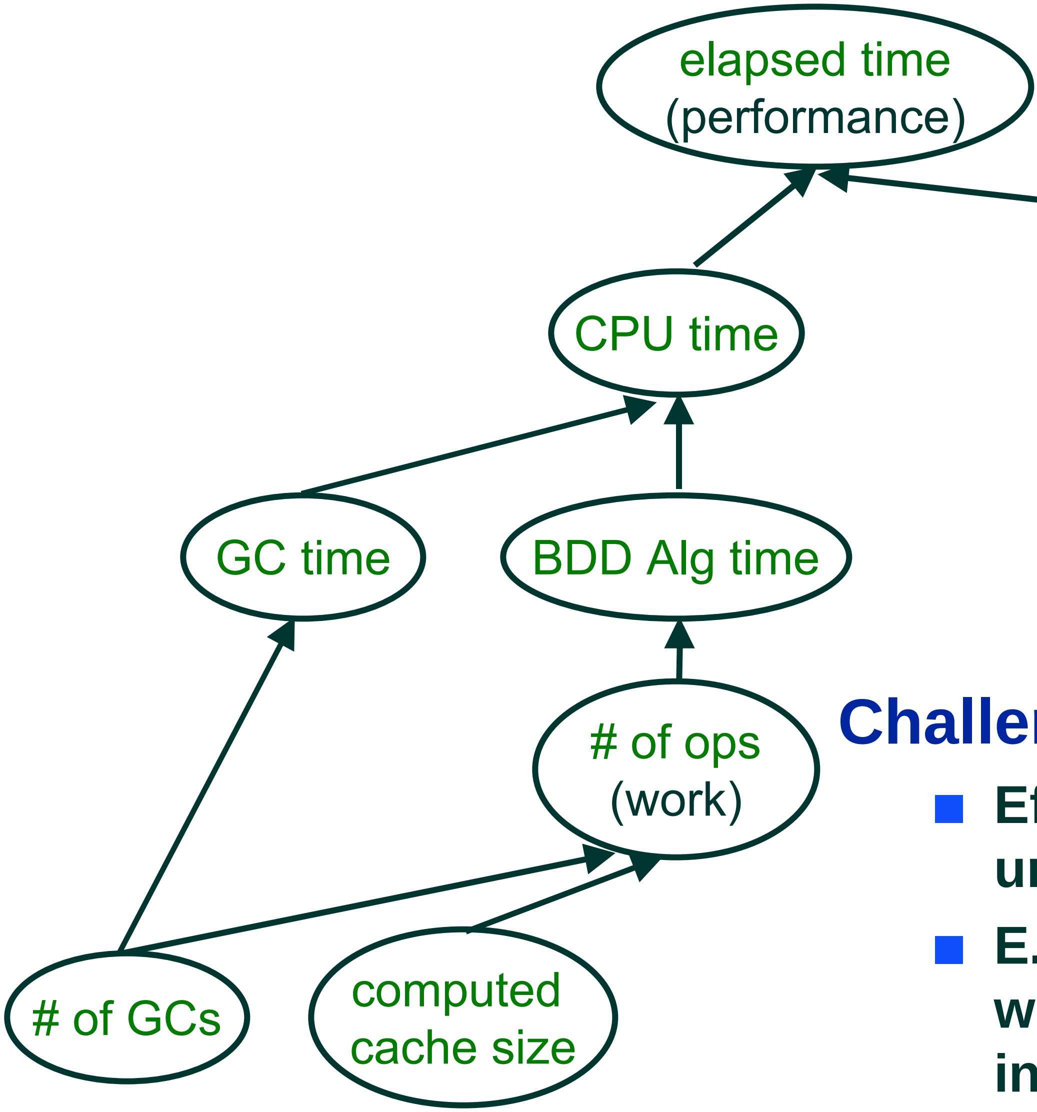
Organization of this Study: Evaluation Process

Phase 1: **no** dynamic variable reordering

Phase 2: **with** dynamic variable reordering



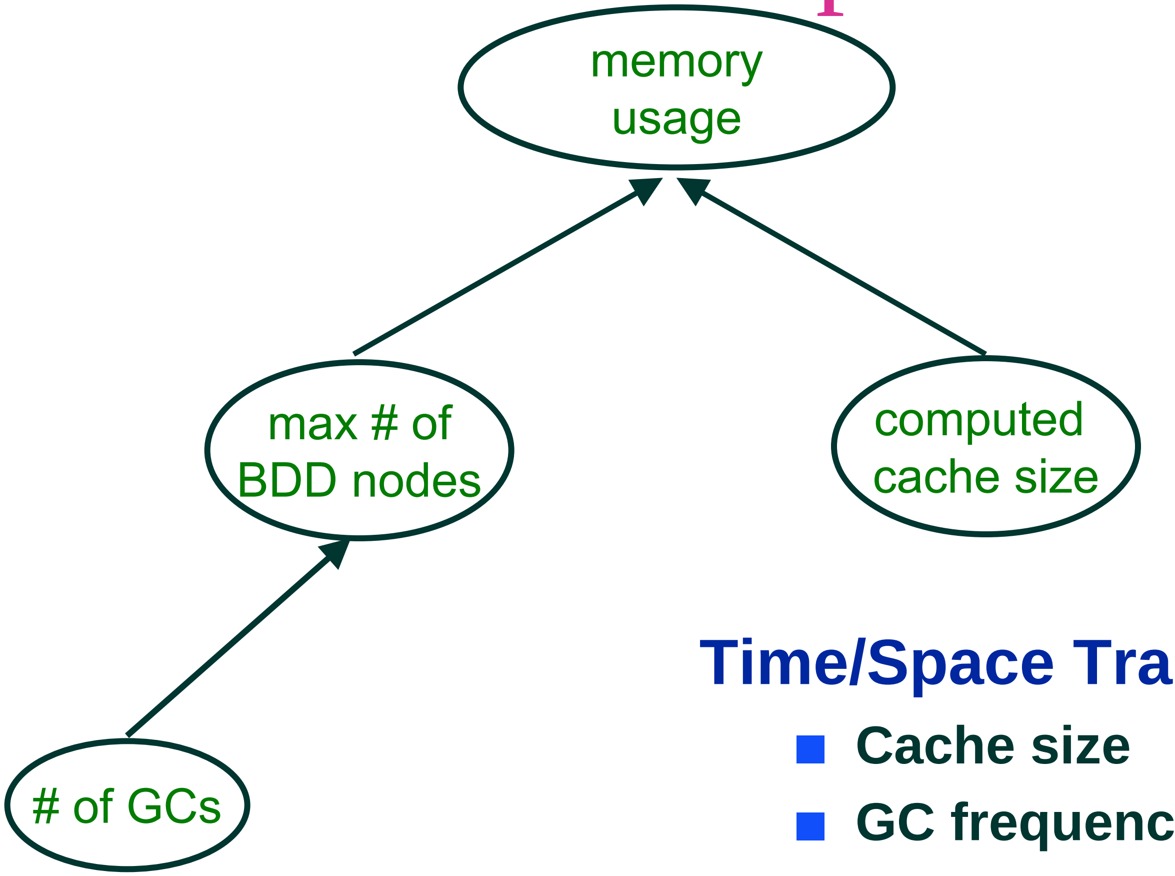
BDD Evaluation Methodology
Metrics: Time



Challenge

- Effect of given optimization uncertain
- E.g., decreasing #GCs would save CPU time, but increase page faults

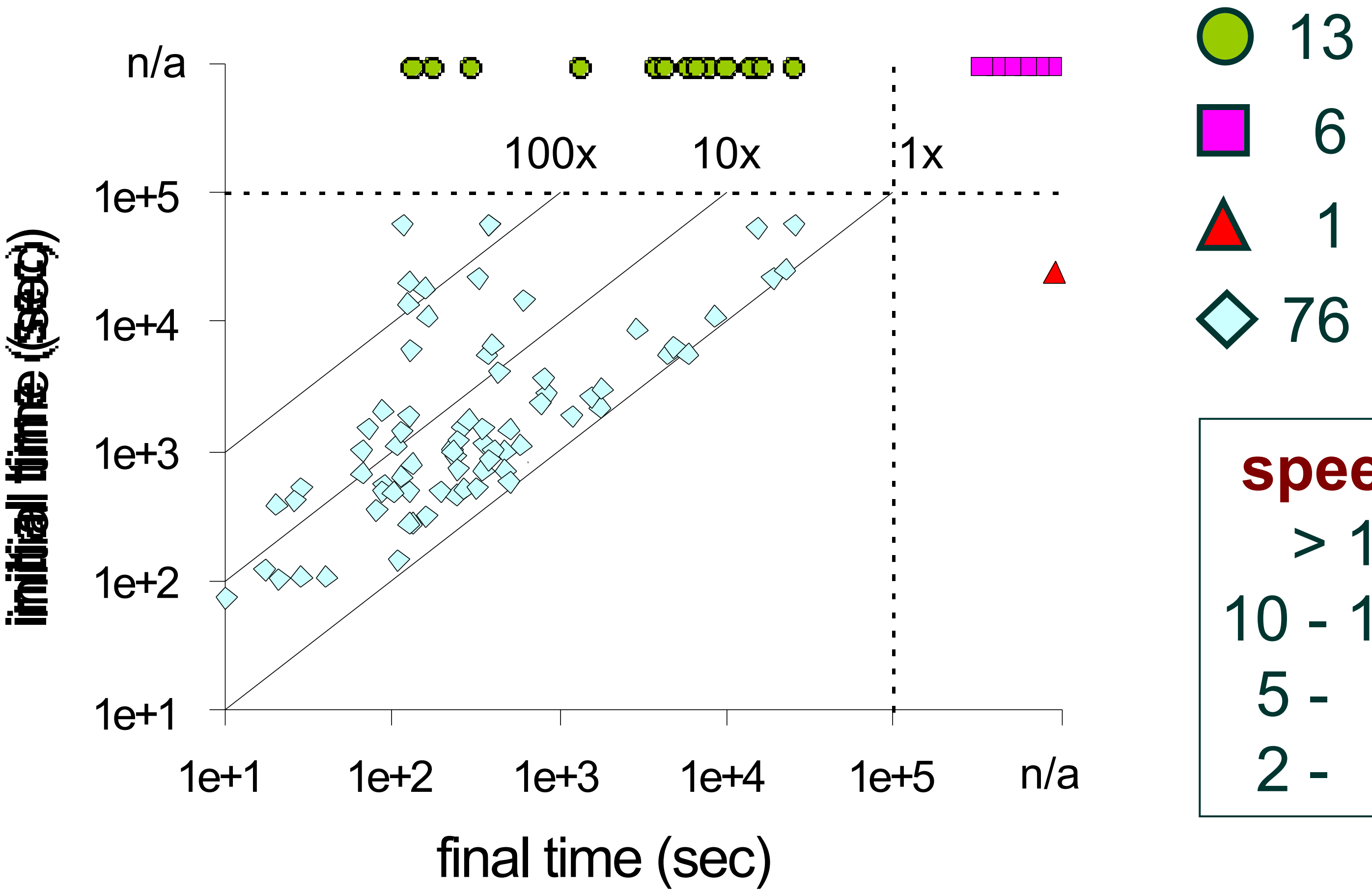
BDD Evaluation Methodology Metrics: Space



Time/Space Trade-Offs

- Cache size
- GC frequency

Phase 1 Results: Initial / Final

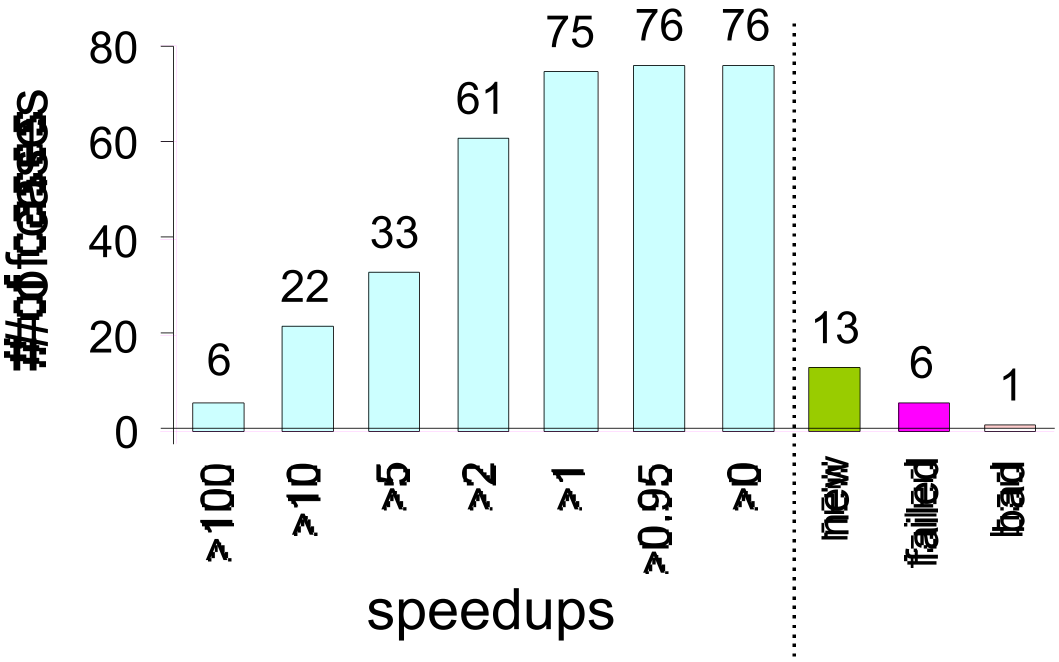


speedups	
> 100:	6
10 - 100:	16
5 - 10:	11
2 - 5:	28

Conclusion: collaborative efforts have led to significant performance improvements

Phase 1: Before/After

Cumulative Speedup Histogram



6 packages * 16 traces = 96 cases

Phase 1: Hypotheses / Experiments

Computed Cache

- effects of computed cache size
- number of repeated sub-problems across time

Garbage Collection

- reachable / unreachable

Complement Edge Representation

- work
- space

Memory Locality for Breadth-First Algorithms

Phase 1:
Hypotheses / Experiments (Cont'd)

For Comparison

- **ISCAS85 combinational circuits (> 5 sec, < 1GB)**
  - c2670, c3540
  - 13-bit, 14-bit multipliers based on c6288

Metric depends only on the trace and BDD algorithms

- **machine-independent**
- **implementation-independent**

Computed Cache Size Dependency

Hypothesis

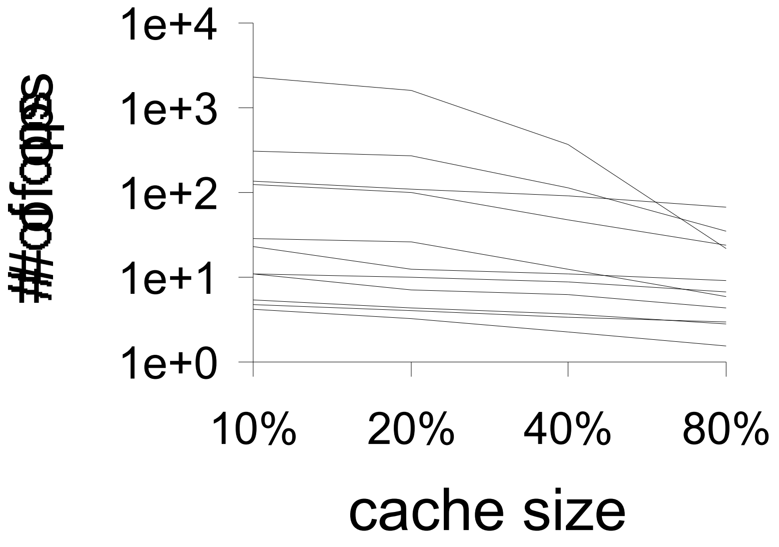
- **The computed cache is more important for MC than for CC.**

Experiment

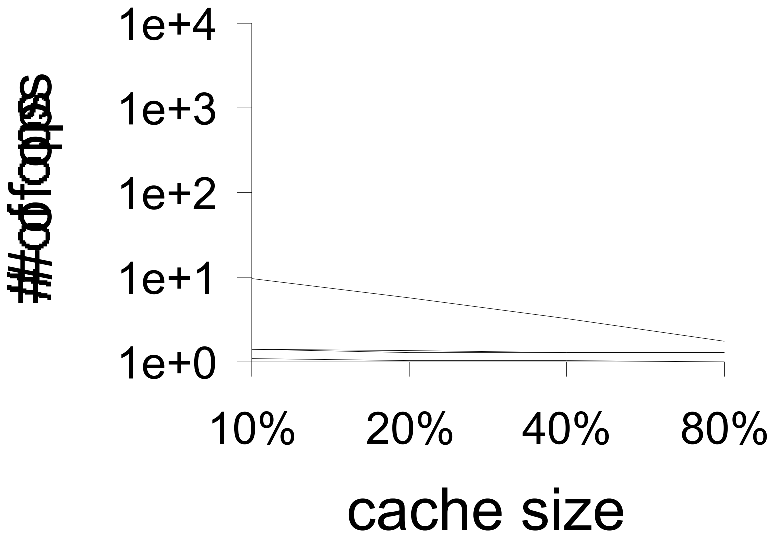
- **Vary the cache size and measure its effects on work.**
  - size as a percentage of BDD nodes
  - normalize the result to minimum amount of work
  - necessary; i.e., no GC and complete cache.

Effects of Computed Cache Size

MC Traces



ISCAS85 Circuits



# of ops: normalized to the minimum number of operations

cache size: % of BDD nodes

Conclusion: large cache is important for MC

Computed Cache:
Repeated Sub-problems Across Time

Source of Speedup

- increase computed cache size

Possible Cause

- many repeated sub-problems are far apart in time

Validation

- study the number of repeated sub-problems across user issued operations (top-level operations).

Hypothesis: Top-Level Sharing

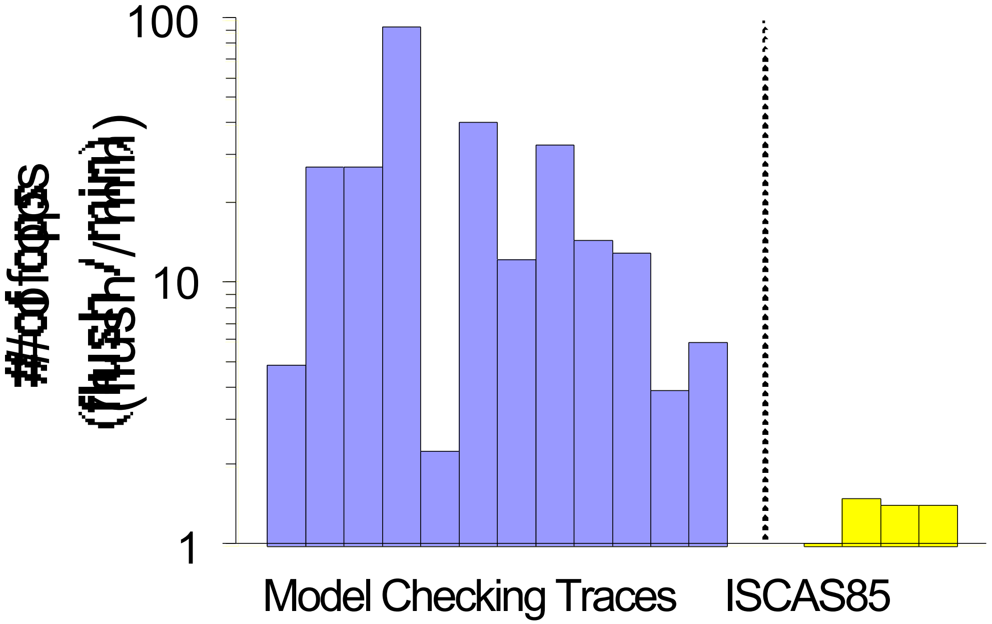
Hypothesis

- MC computations have a large number of repeated
- sub-problems across the top-level operations.

Experiment

- measure the minimum number of operations with GC disabled and complete cache.
- compare this with the same setup, but cache is flushed between top-level operations.

Results on Top-Level Sharing



flush: cache flushed between top-level operations

min: cache never flushed

Conclusion: large cache is more important for MC

Garbage Collection: Rebirth Rate

Source of Speedup

- **reduce GC frequency**

Possible Cause

- **many dead nodes become reachable again (rebirth)**
  - GC is delayed until the number of dead nodes reaches a threshold
  - dead nodes are reborn when they are part of the result of new sub-problems

Hypothesis: Rebirth Rate

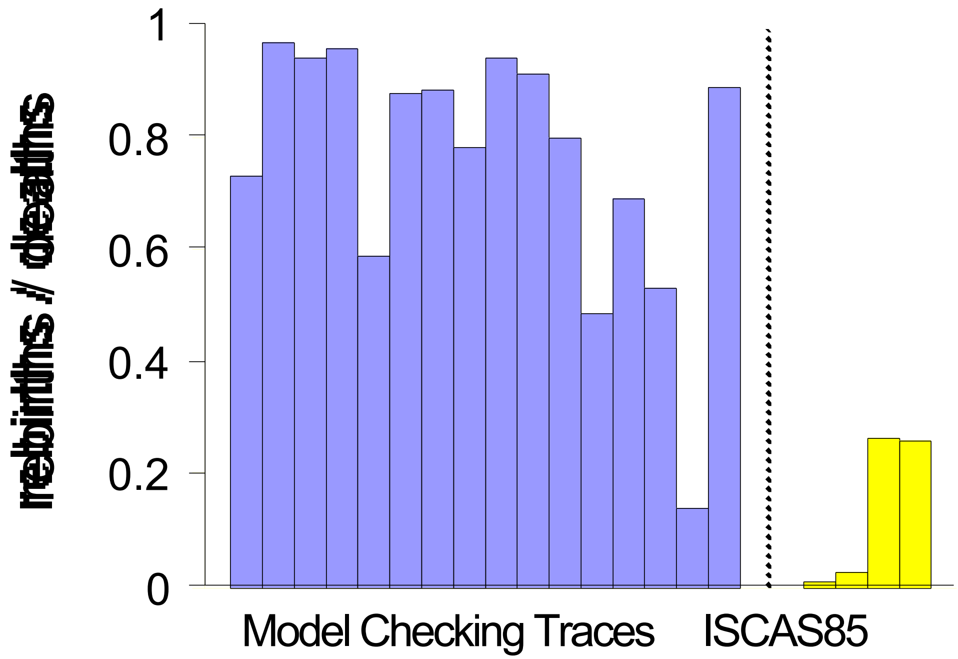
Hypothesis

- MC computations have very high rebirth rate.

Experiment

- measure the number of deaths and the number of rebirths

Results on Rebirth Rate



Conclusions

- **delay garbage collection**
- **triggering GC should not be based only on # of dead nodes**
  - Just because a lot of nodes are dead doesn't mean they're useless
- **delay updating reference counts**
  - High cost to kill/resurrect subgraphs

BF BDD Construction

On MC traces, breadth-first based BDD construction has no demonstrated advantage over traditional depth-first based techniques.

Two packages (CAL and PBF) are BF based.

BF BDD Construction Overview

Level-by-Level Access

- operations on same level (variable) are processed together
- one queue per level

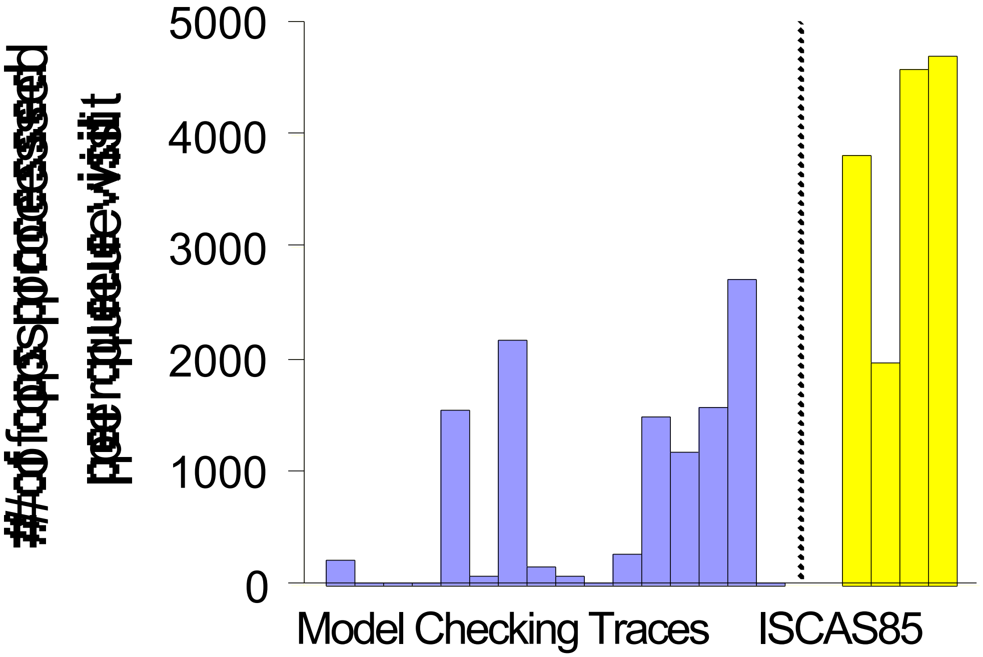
Locality

- group nodes of the same level together in memory

Good memory locality due to BF

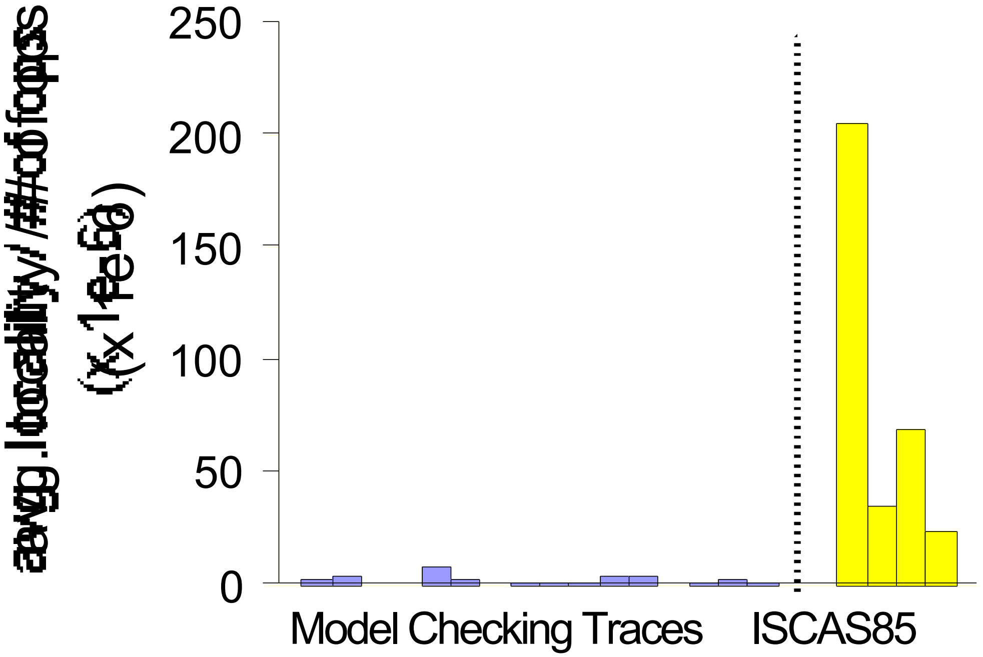
- # of ops processed per queue visit must be high

Average BF Locality



Conclusion: MC traces generally have less BF locality

Average BF Locality / Work



Conclusion: For comparable BF locality,
MC computations do much more work.

Phase 1:
Some Issues / Open Questions

Memory Management

- **space-time tradeoff**
  - computed cache size / GC frequency
- **resource awareness**
  - available physical memory, memory limit, page fault rate

Top-Level Sharing

- **possibly the main cause for**
  - strong cache dependency
  - high rebirth rate
- **better understanding may lead to**
  - better memory management
  - higher level algorithms to exploit the pattern

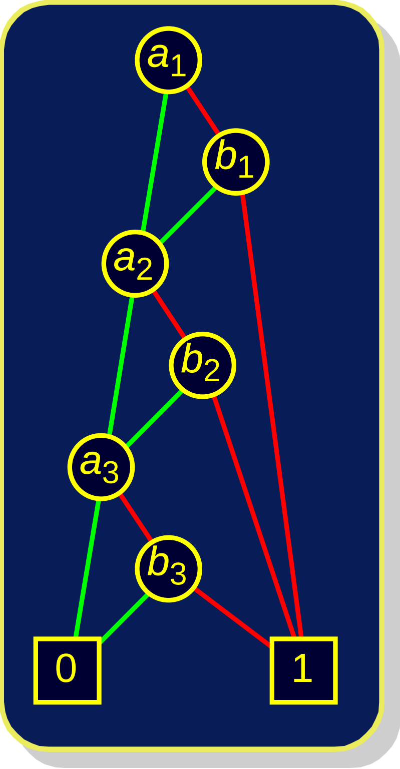
Phase 2:
Dynamic Variable Reordering

BDD Packages Used

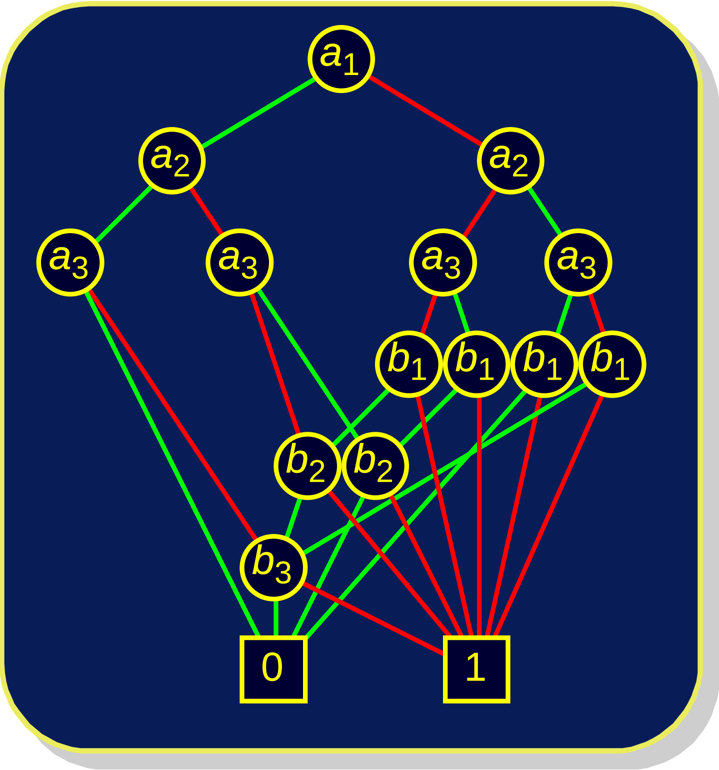
- CAL, CUDD, EHV, TiGeR
- improvements from phase 1 incorporated

Variable Ordering Sensitivity

- **BDD unique for given variable order**
- **Ordering can have large effect on size**
- **Finding good ordering essential**



$$(a_1 \quad b_1)$$
$$(a_2 \quad b_2)$$
$$(a_3 \quad b_3)$$



Dynamic Variable Ordering

- **Rudell, ICCAD ‘93**

Concept

- **Variable ordering changes as computation progresses**
  - Typical application involves long series of BDD operations
- **Proceeds in background, invisible to user**

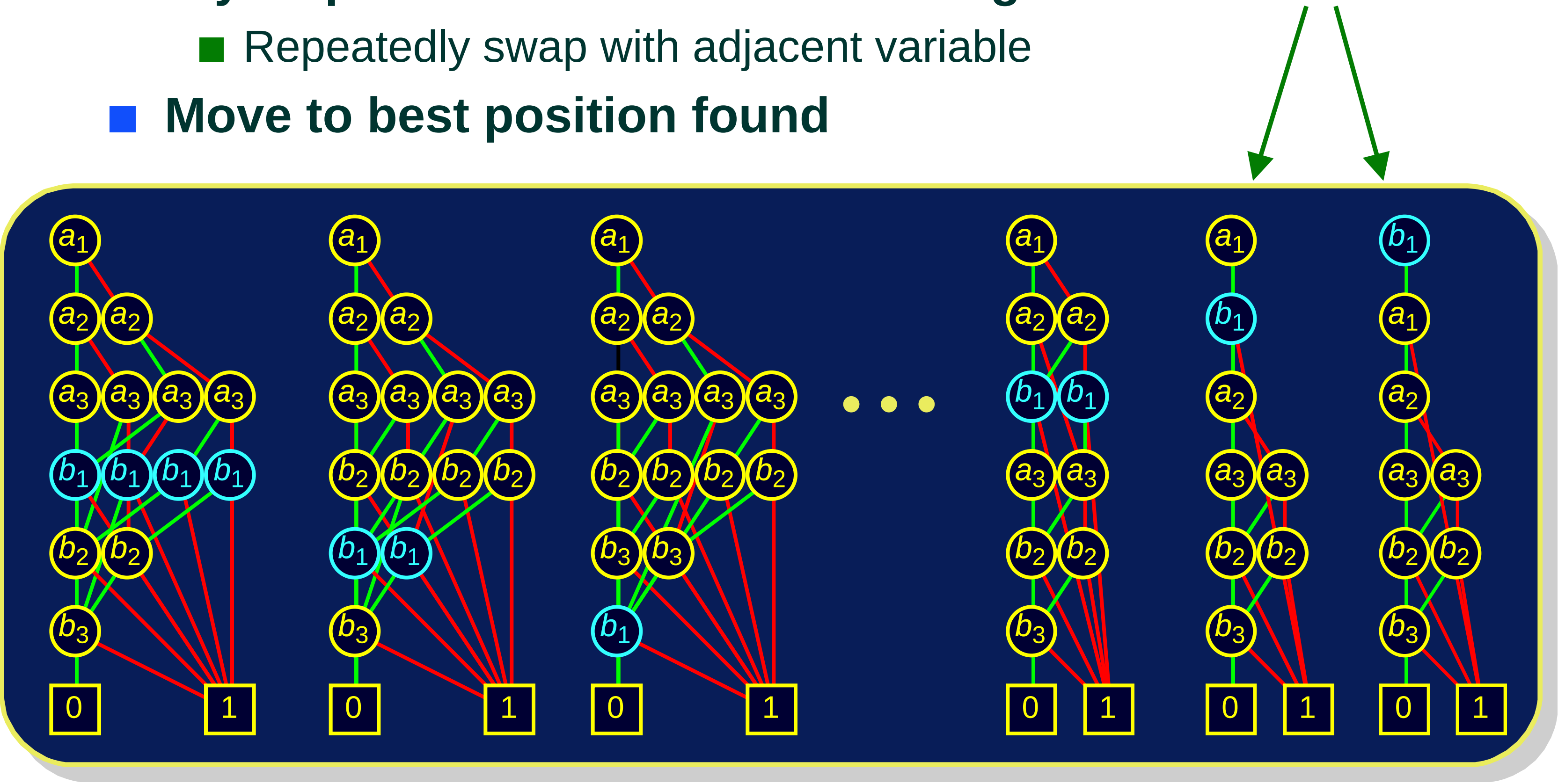
Implementation

- **When approach memory limit, attempt to reduce**
  - Garbage collect unneeded nodes
  - Attempt to find better order for variables
- **Simple, greedy reordering heuristics**
  - Ongoing improvements

Reordering By Sifting

- **Choose candidate variable**
- **Try all positions in variable ordering**
  - Repeatedly swap with adjacent variable
- **Move to best position found**

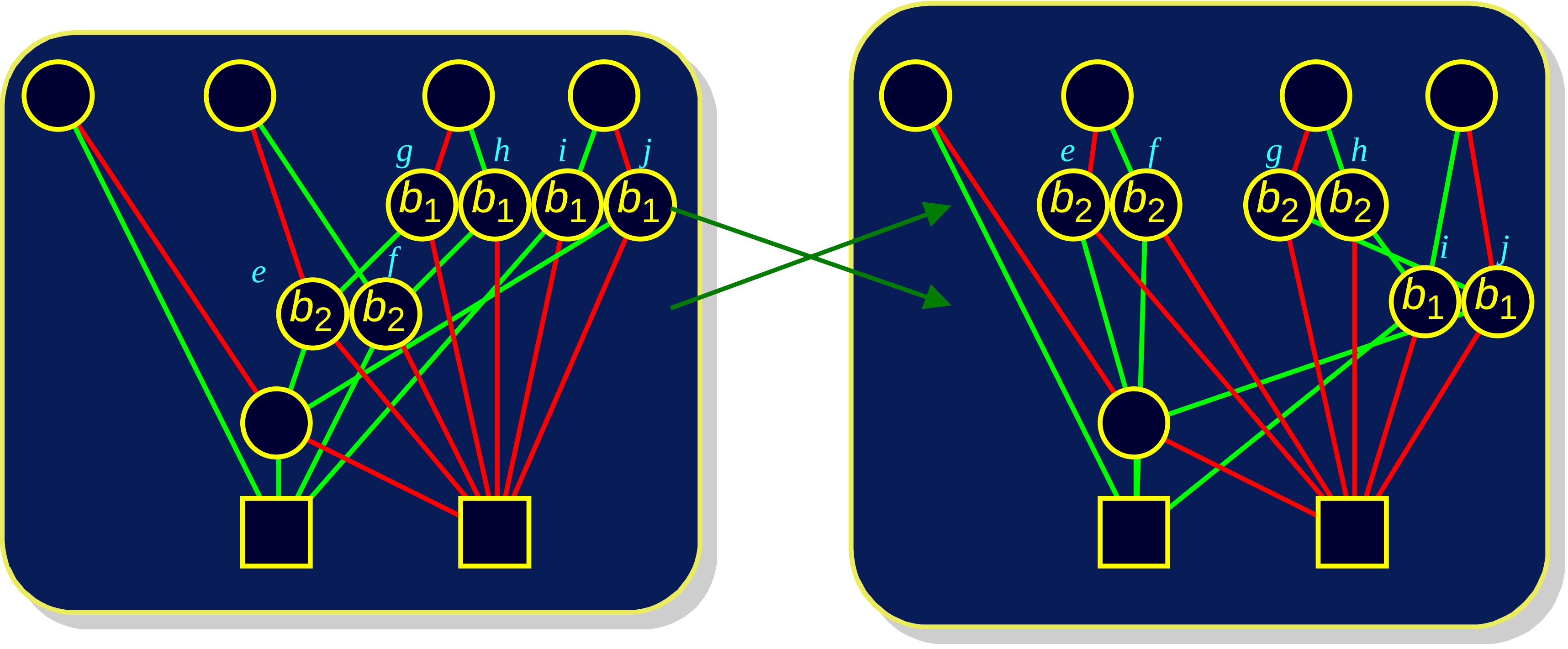
Best Choices



Swapping Adjacent Variables

Localized Effect

- Add / delete / alter only nodes labeled by swapping variables
- Do not change any incoming pointers



Dynamic Ordering Characteristics

Added to Many BDD Packages

- Compatible with existing interfaces
- User need not be aware that it is happening

Significant Improvement in Memory Requirement

- Limiting factor in many applications
- Reduces need to have user worry about ordering
- Main cost is in CPU time
  - Acceptable trade-off
  - May run 10X slower

Compatible with Other Extensions

- Now part of “core technology”

Why is Variable Reordering Hard to Study

Time-space tradeoff

- how much time to spent to reduce graph sizes

Chaotic behavior

- e.g., small changes to triggering / termination criteria
- can have significant performance impact

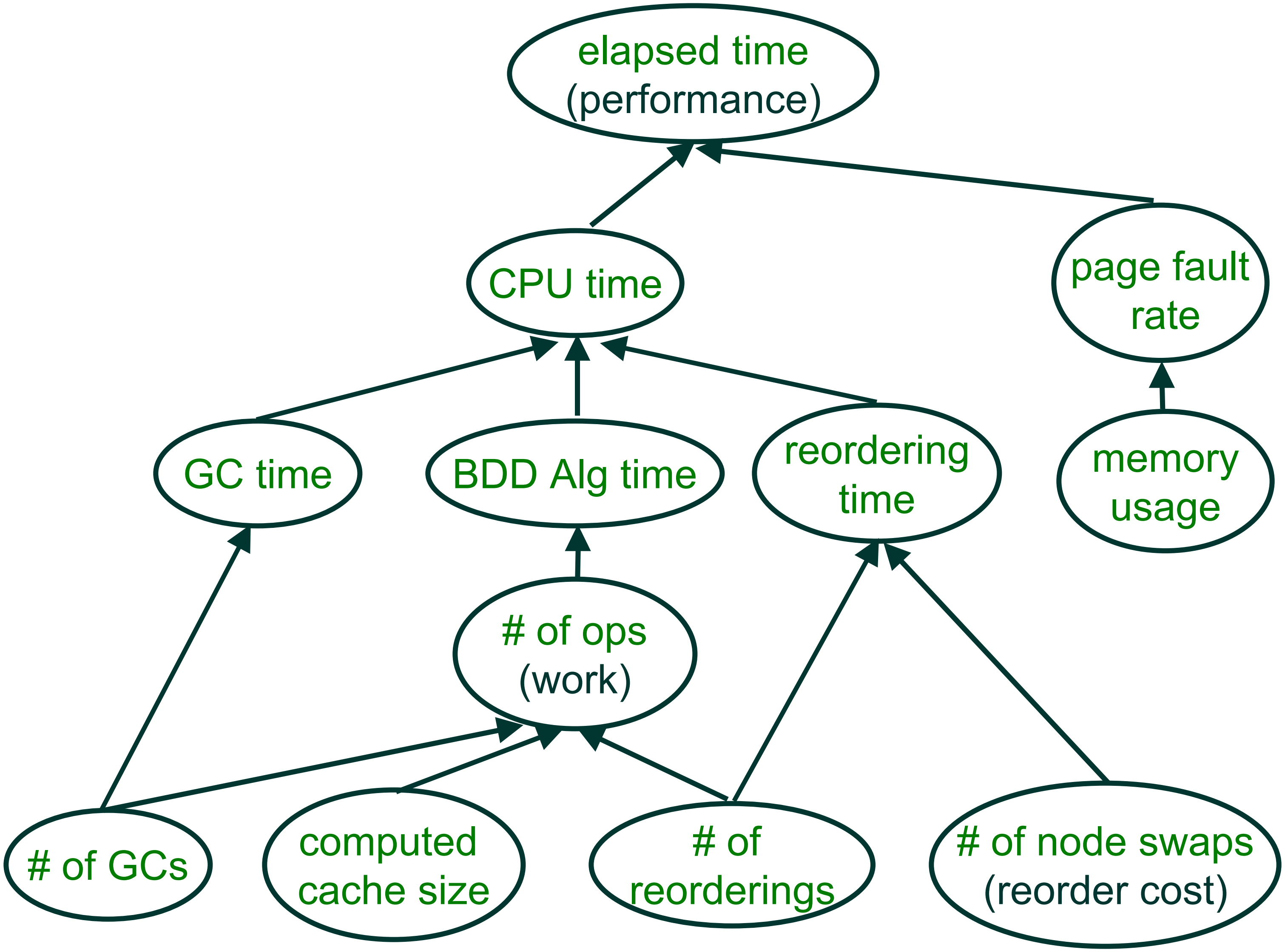
Resource intensive

- reordering is expensive
- space of possible orderings is combinatorial

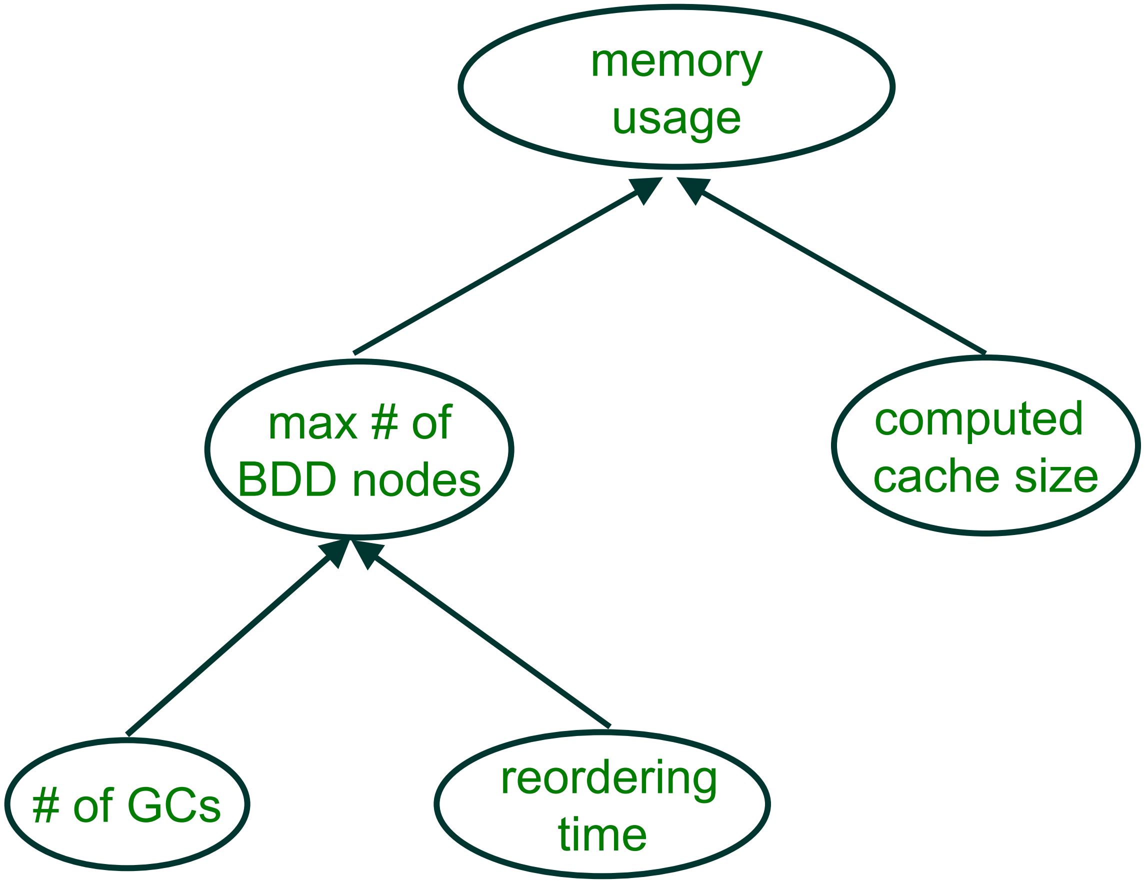
Different variable order different computation

- e.g., many “don’t-care space” optimization algorithms

BDD Evaluation Methodology
Metrics: Time



BDD Evaluation Methodology Metrics: Space



Phase 2: Experiments

Quality of Variable Order Generated

Variable Grouping Heuristic

- keep strongly related variables adjacent

Reorder Transition Relation

- BDDs for the transition relation are used repeatedly

Effects of Initial Variable Order

- with and without variable reordering

Only CUDD is used

Effects of Initial Variable Order: Perturbation Algorithm

Perturbation Parameters (p, d)

- p: probability that a variable will be perturbed
- d: perturbation distance

Properties

- in average, p fraction of variables is perturbed
- max distance moved is 2d
- (p = 1, d = ∞)   completely random variable order

For each perturbation level (p, d)

- generate a number (sample size) of variable orders

Effects of Initial Variable Order: Parameters

Parameter Values

- p: (0.1, 0.2, …, 1.0)
- d: (10, 20, …, 100, ∞)
- sample size: 10

For each trace

- 1100 orderings
- 2200 runs (w/ and w/o dynamic reordering)

Effects of Initial Variable Order: Smallest Test Case

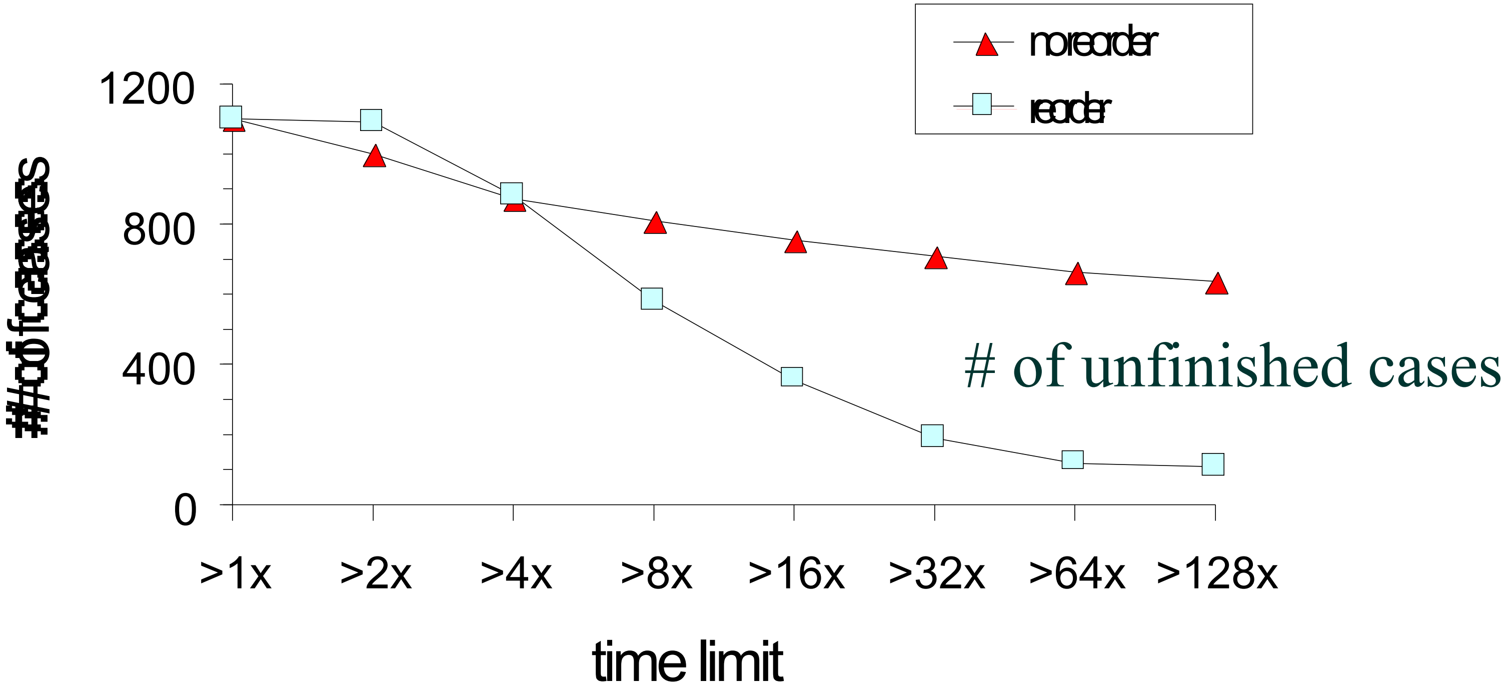
Base Case (best ordering)

- time: 13 sec
- memory: 127 MB

Resource Limits on Generated Orders

- time: 128x base case
- memory: 500 MB

Effects of Initial Variable Order: Result



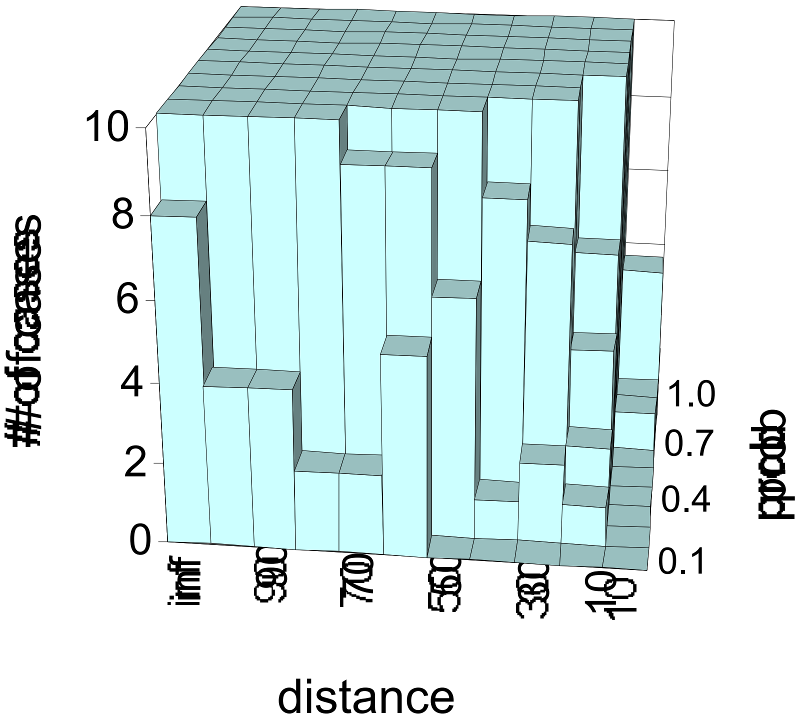
# of unfinished cases

At 128x/500MB limit, “no reorder” finished **33%**,
“reorder” finished **90%**.

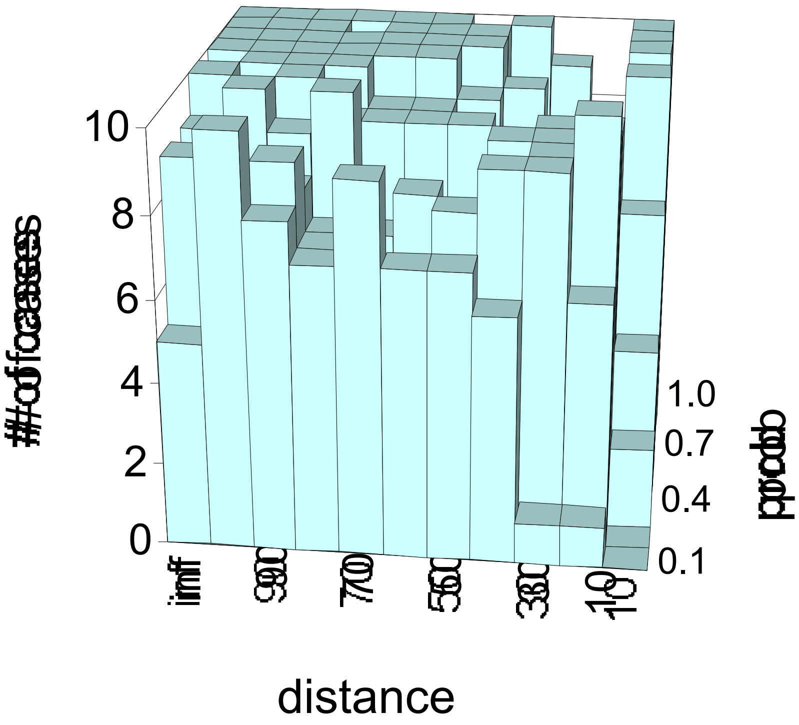
Conclusion: dynamic reordering is effective

> 4x or > 500Mb

No Reorder

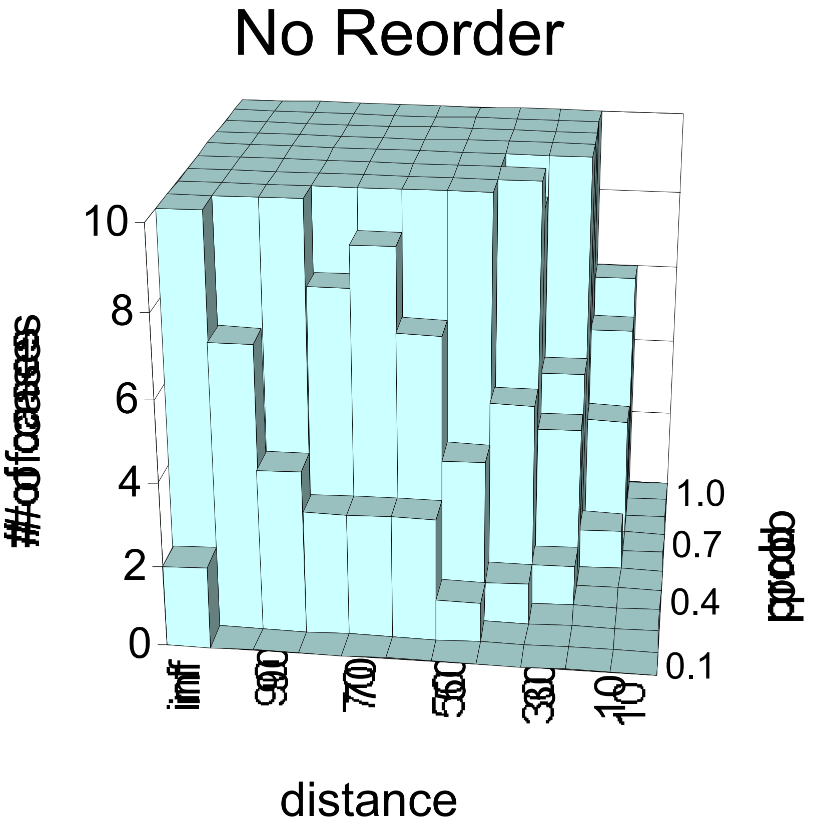


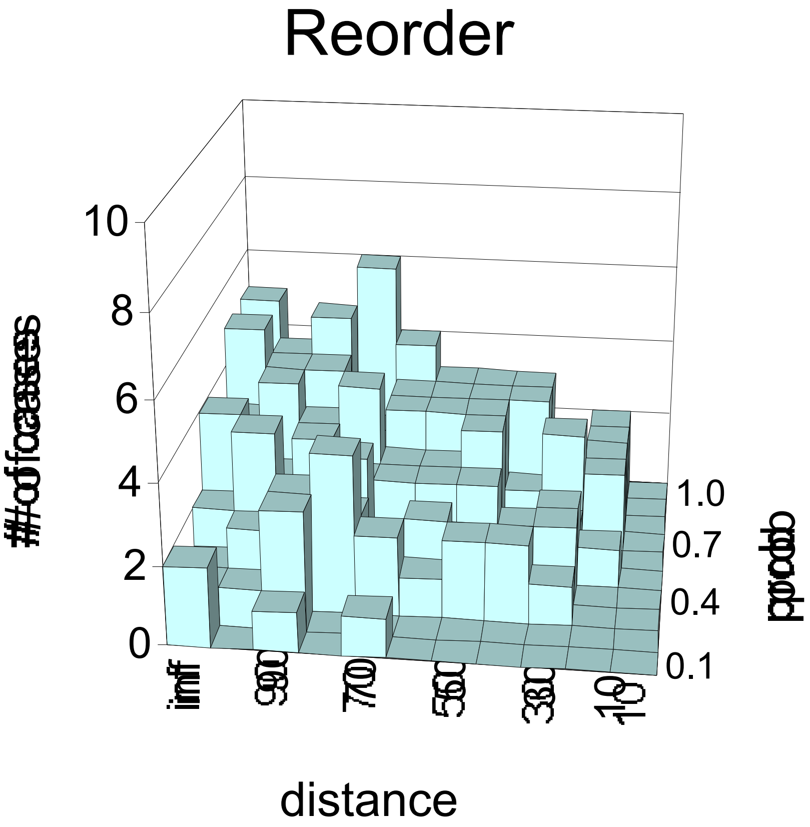
Reorder



Conclusions: For very low perturbation, reordering does not work well.
Overall, very few cases get finished.

> 32x or > 500Mb


No Reorder


Reorder

Conclusion: variable reordering worked rather well

Phase 2:
Some Issues / Open Questions

Computed Cache Flushing

- cost

Effects of Initial Variable Order

- determine sample size

Need New Better Experimental Design

Summary

Collaboration + Evaluation Methodology

- **significant performance improvements**
  - up to 2 orders of magnitude
- **characterization of MC computation**
  - computed cache size
  - garbage collection frequency
  - effects of complement edge
  - BF locality
  - effects of reordering the transition relation
  - effects of initial variable orderings
- **other general results (not mentioned in this talk)**
- **issues and open questions for future research**

Conclusions

Rigorous quantitative analysis can lead to:

- **dramatic performance improvements**
- **better understanding of computational characteristics**

Adopt the evaluation methodology by:

- **building more benchmark traces**
    - for IP issues, BDD-call traces are hard to understand
- **using / improving the proposed metrics for future evaluation**

For data and BDD traces used in this study,
http://www.cs.cmu.edu/~bwolen/fmcad98/