Parallelism: A Siren Song?

Moshe Y. Vardi

Rice University

Siren Song

Definition: "Siren Song" refers to an appeal that is hard to resist but that, if heeded, will lead to a bad result.

An Old Vignette: The ILLIAC IV

- Funded by the US Air Force in the mid 1960s
- High parallelism, with up to 256 processors
- 11 years of development
- 4X over budget
- When launched in 1976, it was outperformed by the Cray I, which was commercially available!

Faster and Faster

Question: How do we speed up computing?

Wrong Answer: Moore's Law

- Moore's Law is about transistor density, not about performance!

Right Answer: It's complicated!

- Reduced transistor-switching time – but, for CMOS, *this is essentially over!*
- Parallelism – *many transistors working in parallel!*

Leveraging Transistor Parallelism

How does transistor parallelism yield performance?

- Cache memory
- Bit-level parallelism: 4, 8, 16, 32, 64
- Inter-instruction parallelism
  - Pipelines
  - Branch prediction
  - Speculative execution
  - Out-of-order execution

But: We have maxed out!

- Diminished ROI
- Energy constraints (e.g., Intel's 2004 "Tejas Affair")

“The Future is Parallel Computing”

Fact: Processors’ speed increase is slow!

What Now?

- *Parallelism on a Chip:* multi-core processors, local and shared caches
- *Parallelism on a Network:* Beowulf cluster
  - Commodity (multi-core) processors
  - Ethernet and/or Infiniband communication
  - Local memories, shared secondary storage

Theory of Parallel Computation

Computational Model: *CRCW PRAM*

- Multiple processors, each with its own program
- Local and shared memories, one cycle for communication
- Concurrent reads, concurrent writes (with conflict resolution)

Complexity Class: *NC*

- Polynomial number of processors
- Polylogarithmic time

Example: Graph Reachability

Input: Digraph $G = (V, E)$
Output: Reachability Digraph $G = (V, R)$

Processors: $P_{uvw} : u, v, w \in V$ - cubic

Initial: $R(u, v) \leftarrow E(u, v)$

Algorithm: $R(u, w) \leftarrow R(u, v) \wedge R(v, w)$

Running Time: logarithmic (NC^1)

We have: numerous NC algorithms, many sublogarithmic

What Cannot Be Parallelized?

Working Conjecture: $P \neq NC$

Corollary: P-complete problems cannot be parallelized.

- **Examples:** reachability games, Horn SAT

But: flimsy evidence for conjecture!

Critique

Claim: PRAM/NC theory suffers from several fundamental flaws.

- Model ignores communication costs.
- Number of processors cannot be practically scaled.
- Focus on polylogarithmic time unjustified.
- Unmotivated choice of problems.

Optimal Speedup

Definition:

- Let $Seq(P, n)$ be the fastest known worst-case running time of a sequential algorithm to solve a problem P of size n.
- The best upper bound on the parallel time achievable using p processors is $O(Seq(P, n)/p)$.
- A parallel algorithm that achieves this running time is said to have *optimal speedup*.

Example: An $O((n^3 \log n)/p)$ (for $p \leq n$) parallel Max-Flow algorithm [Shiloach&Vishkin, 1982] - almost optimal!

Reality of Parallel Computing

Claim: Parallel programming is hard – very few success stories:

- *Inter-instruction parallelism:* but this technique has maxed out, due to energy constraints
- *Data parallelism:* multiple processors perform the same operation on multiple data simultaneously

Non-success Story: *task parallelism* (MIMD) – distributing execution threads across different processors – *too hard* to coordinate multiple threads and pass information between them – *Amdahl's Law* and *Gustafson's Law*

Data Parallelism

SIMD: Single Instruction, Multiple Data
SPMD: Single Program, Multiple Data

Examples:

- *Wide registers:* 4, 8, 16, 32, 64
- Vector processing: single instruction operating on arrays of data, e.g., multiply two arrays of floating-point numbers
  - *Cray-1:* eight "vector registers," sixty-four 64-bit words each.
  - x86: Streaming SIMD Extensions (SSE)
- *GPUs:* graphics-processing units – data parallelism specialized for graphics.

MapReduce

MapReduce, 2005: First successful framework of task parallelism:

- *Map:* Master node partitions input up into smaller sub-problems, and distributes to worker nodes. Worker nodes may repeat, recursively.
- *Process:* Worker nodes process sub-problems, and pass answers back to master nodes up the tree.
- *Reduce:* Master node takes answers to sub-problems and combines them to get answer.

Example: IBM's Watson winning in Jeopardy!

Widely available: libraries in C++, C#, Erlang, Java, OCaml, Perl, Python, PHP, Ruby, F#, R

Ensemble Algorithms: Embarrassing Parallelism

Basic Observation: For many problems there are many potential algorithms, but no "best" algorithm

Solution: run them all in parallel, terminating when first terminates!

Example: *symbolic LTL satisfiability* [Rozier&V., FM 2011]

LTL Satisfiability Checking Reduces to Model Checking

- Let f be a LTL formula over a set $Prop$ of propositions.

- Let the system model M be *universal* – containing all possible traces over $Prop$.

- Then f is *satisfiable* precisely when M does *not* satisfy $\neg f$.

LTL Satisfiability in SMV

1. Model check $\neg f$ against a *universal SMV model*.

```
MODULE main
  VAR
    a : boolean;
    b : boolean;
    c : boolean;
  LTLSPEC !f
  FAIRNESS TRUE
```

2. SMV:
   (a) Negates the property, $\neg f$.
   (b) Symbolically compiles f into A_f and conjoins with the universal model.
   (c) Searches for a fair path that satisfies f.

LTL Satisfiability Checking via Symbolic Model Checking

$f = (p\mathcal{U}q)$

```
module main() {
  VAR
    p: boolean;
    q: boolean;
    EL_X__p_U_q : boolean;
  DEFINE S__p_U_q := q | (p & EL_X__p_U_q);
  TRANS  ( EL_X__p_U_q = (next(S__p_U_q) ))
  FAIRNESS (!S__p_U_q | q)
  SPEC !(S__p_U_q & EG TRUE) }
```

```
module main() {
  VAR
    p: boolean;
    q: boolean;
  FAIRNESS  TRUE; }
```

symbolic $A_{\neg f}$ $\otimes$ universal M

$A_{M,\neg f} \longrightarrow$ CadenceSMV:EMPTY?

Key: The encoding of $A_{\neg f}$ has a major impact on complexity.

Symbolic Encodings

Fact: Since 1994, there has been *only one* encoding for LTL-to-symbolic automata, due to Clarke, Grumberg &Hamaguchi (CGH) – used by *all* symbolic model checkers

Questions:

- Can we do it differently?
- Can we do it better?

A Set of 30 Symbolic Automata Encodings

Novel encodings are combinations of four components:

1. *Normal Form:* BNF or NNF
2. *Automaton Form:* GBA or TGBA
3. *Transition Form:* fussy or sloppy
4. *Variable Order:* default, naïve, `LEXP`, `LEXM`, `MCS-MIN`, `MCS-MAX`

Note: CGH = BNF/GBA/fussy/default

Normal Forms

- BNF: $\neg$, $\vee$, $next$, $until$
- NNF:
  - Add $\wedge$, $release$
  - push negations all the way to atomic propositions

TGBA: A New Symbolic Automaton Form

- Requires NNF
- Avoid declaring variables for *eventuality expansion rules*
  CGH/GBA: $p\,\mathcal{U}\,q \equiv$ `q | (p & VAR_X__p_U_q)`
- Ensure eventualities using *promise variables*
  $p\,\mathcal{U}\,q \equiv$ `( (q) | (p &  P__p_U_q & (next(VAR__p_U_q)))`
- Simpler transitions
- Fairness means promise fulfilled: `FAIRNESS (!P__p_U_q)`

Sloppy: A New Transition Form

- *Fussy: iff* transitions–more constrained

  `TRANS (EL_(p&q) = EL_p&EL_q)`

- *Sloppy: if* transitions–less constrained

  `TRANS (EL_(p&q) -> EL_p&EL_q)`

  – Requires NNF

30 Combinations

<table>
<tr><th>Automaton Form</th><th>Normal Form</th><th>Transition Form</th><th>Variable Order</th></tr>
<tr><td rowspan="2">GBA</td><td>BNF</td><td>fussy</td><td rowspan="4"><code>default</code>
<code>naïve</code>
<code>LEXP</code>
<code>LEXM</code>
<code>MCS-MIN</code>
<code>MCS-MAX</code></td></tr>
<tr><td rowspan="3">NNF</td><td rowspan="2">fussy</td></tr>
<tr><td rowspan="2">TGBA</td></tr>
<tr><td>sloppy</td></tr>
</table>

Input Formulas

Rozier & V., 2007:

- *Random Formulas:* 60,000 instances
- *Scalable Pattern Formulas:* 8,000 instances
- *Scalable Counter Formulas:* 60 instances

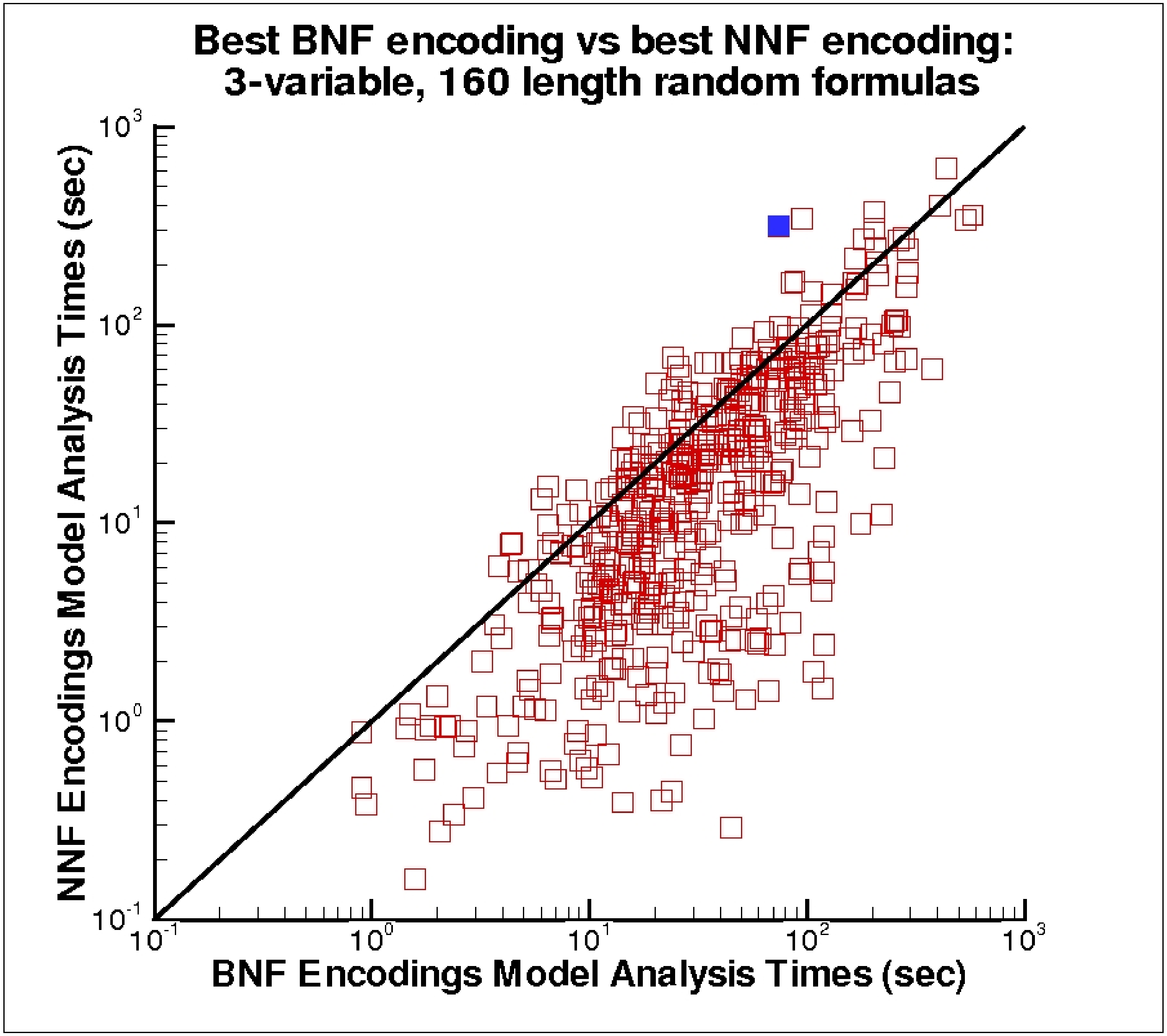
Experimental Results

- Seven configurations are not competitive.
- NNF is the best normal form, most (but not all) of the time.
- No automaton form is best.
- No transition form is best.
- No variable order is best; `LEXM` is not competitive.
- A formula class typically has a best encoding, but predictions are difficult.

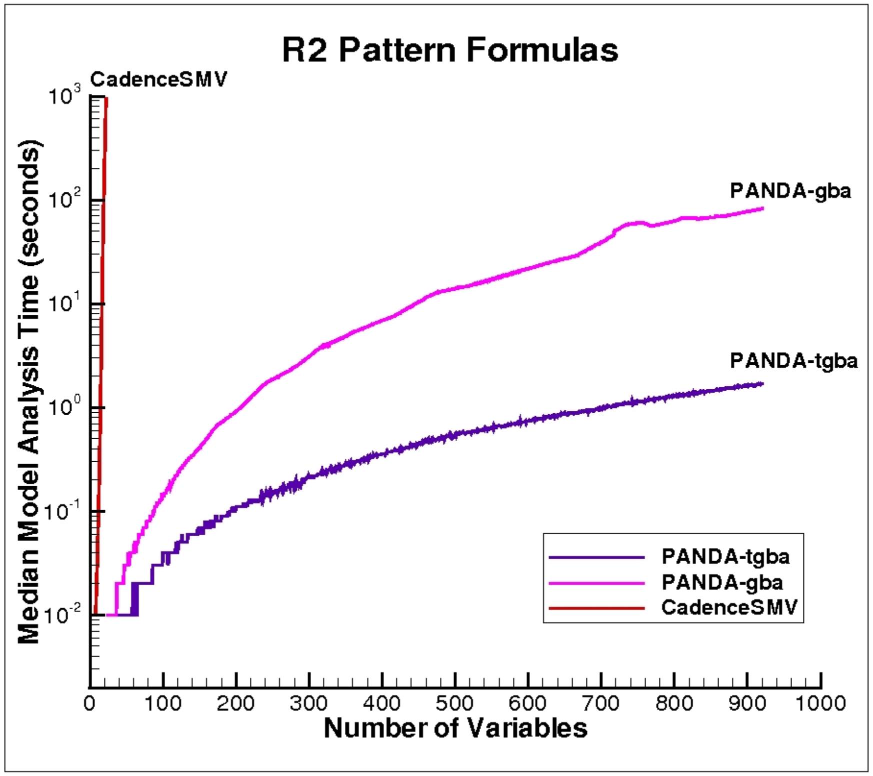
Tool: *PANDA* – implements all 30 encodings

NNF is the best normal form, most (not all) of the time

- NNF encodings were always better for all counter and pattern formulas.
- BNF encodings were optimal for a nontrivial portion of our random formulas.


Best BNF encoding vs best NNF encoding:
3-variable, 160 length random formulas
NNF Encodings Model Analysis Times (sec)
BNF Encodings Model Analysis Times (sec)
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}

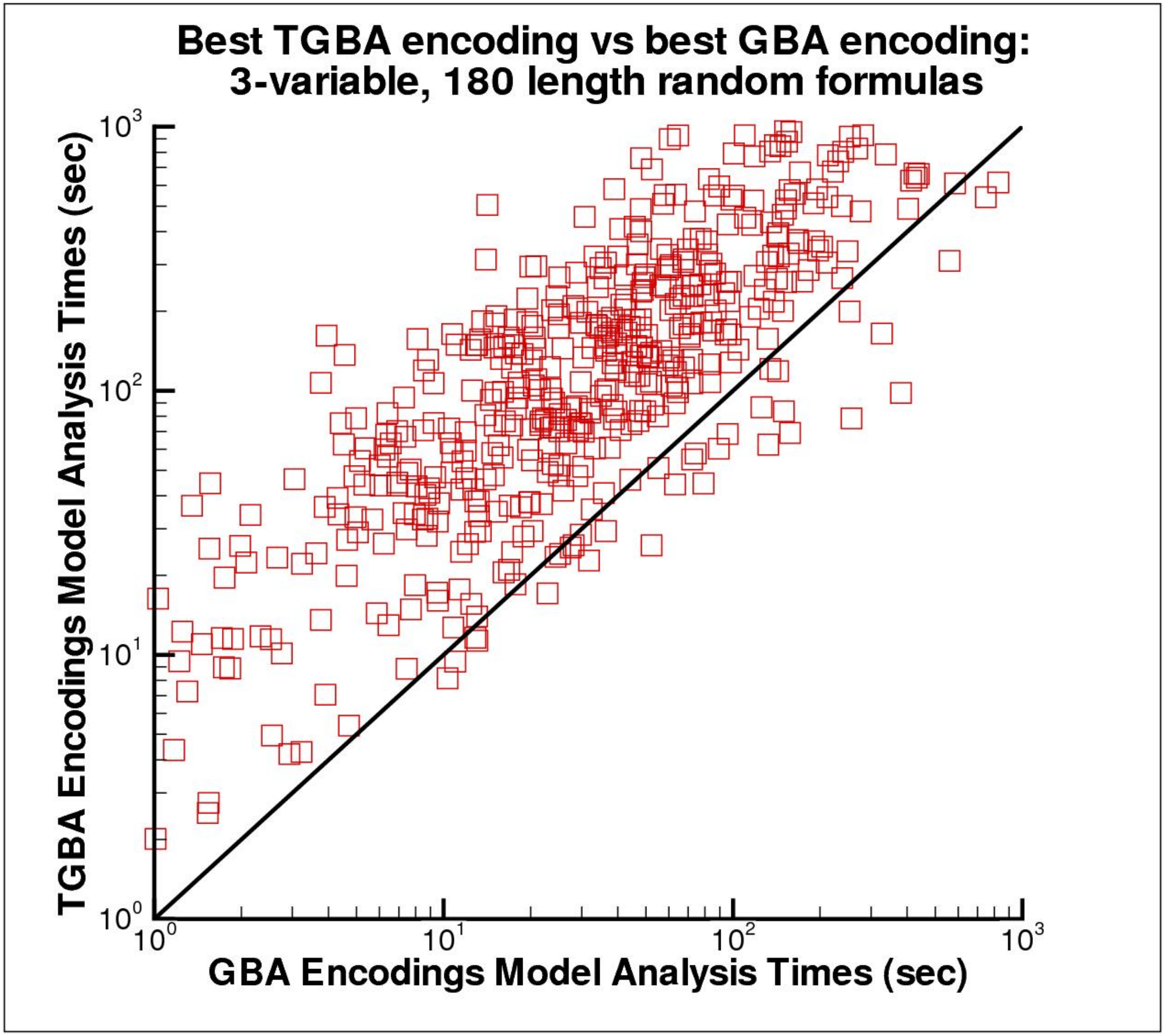
TGBAs can beat CGH/CadenceSMV



$R_2(n) = (..(p_1 \, \mathcal{R} \, p_2) \, \mathcal{R} \, \ldots) \, \mathcal{R} \, p_n.$

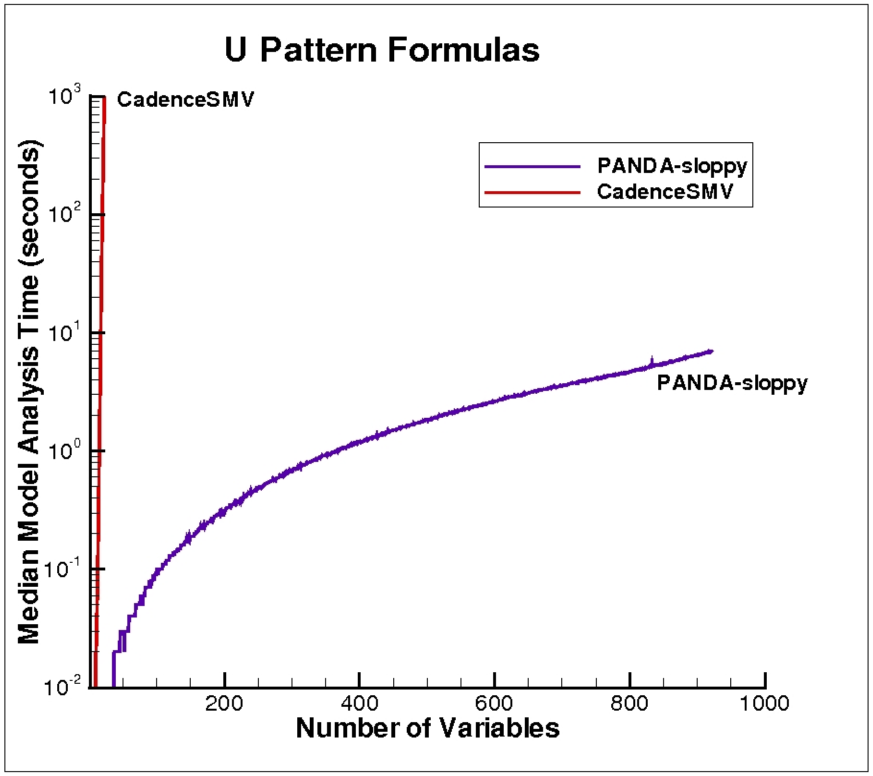
No automaton form is best

- TGBA encodings are better for $C2$, $R2$, U, and $C1$ pattern formulas.
- GBA encodings are better for R-pattern formulas, majority of random formulas.
- TGBA is better for 3-variable counters.
- GBA is better for 2-variable linear counters.



Best TGBA encoding vs best GBA encoding:
3-variable, 180 length random formulas
TGBA Encodings Model Analysis Times (sec)
GBA Encodings Model Analysis Times (sec)
10^0
10^1
10^2
10^3

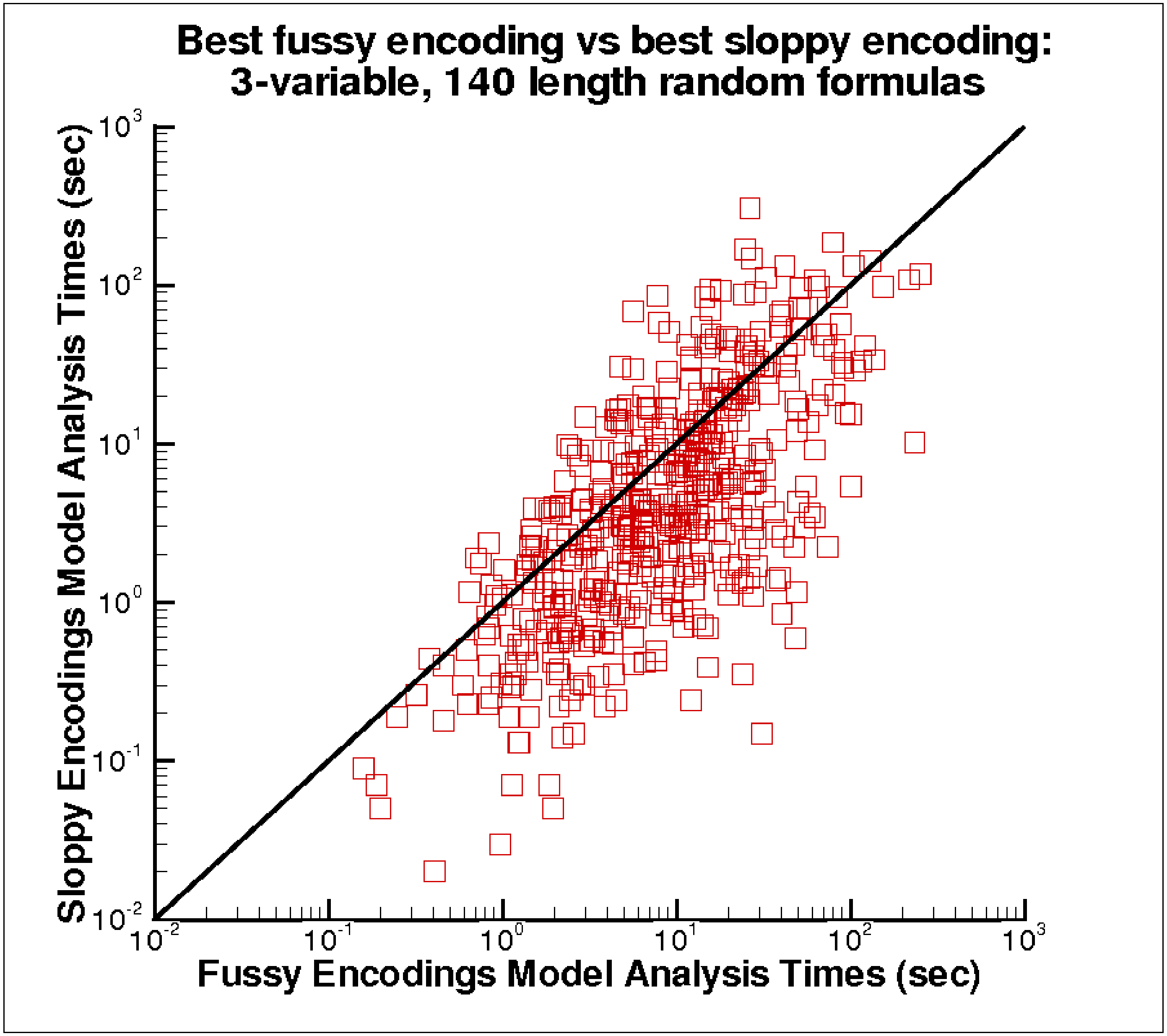
Sloppy transitions can beat CGH/CadenceSMV



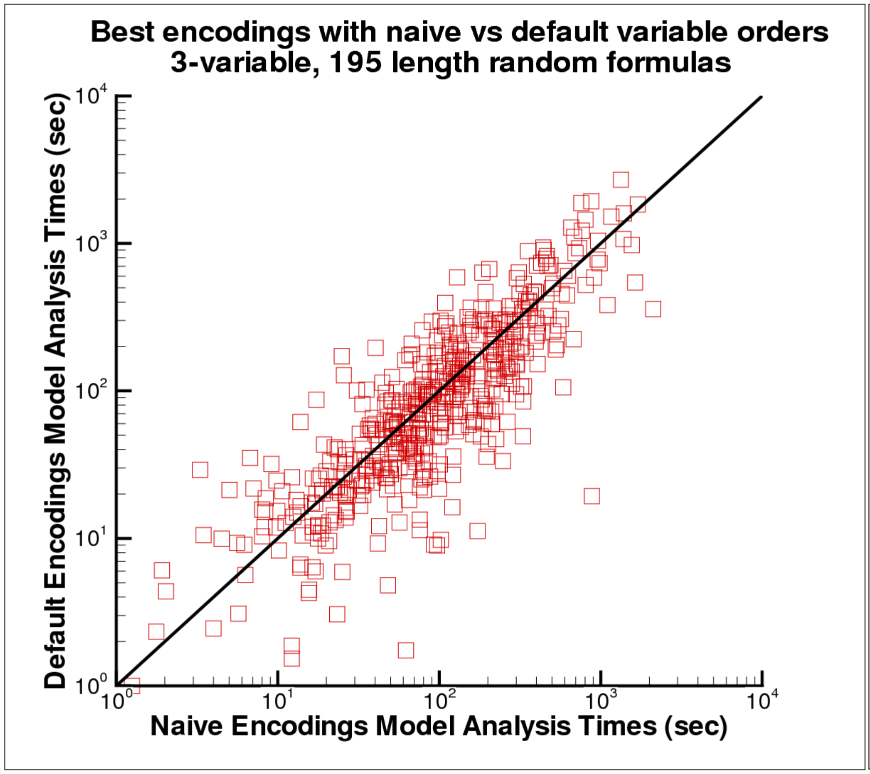
$$U(n) = (\ldots(p_1\, \mathcal{U}\, p_2)\, \mathcal{U}\ \ldots)\, \mathcal{U}\, p_n.$$

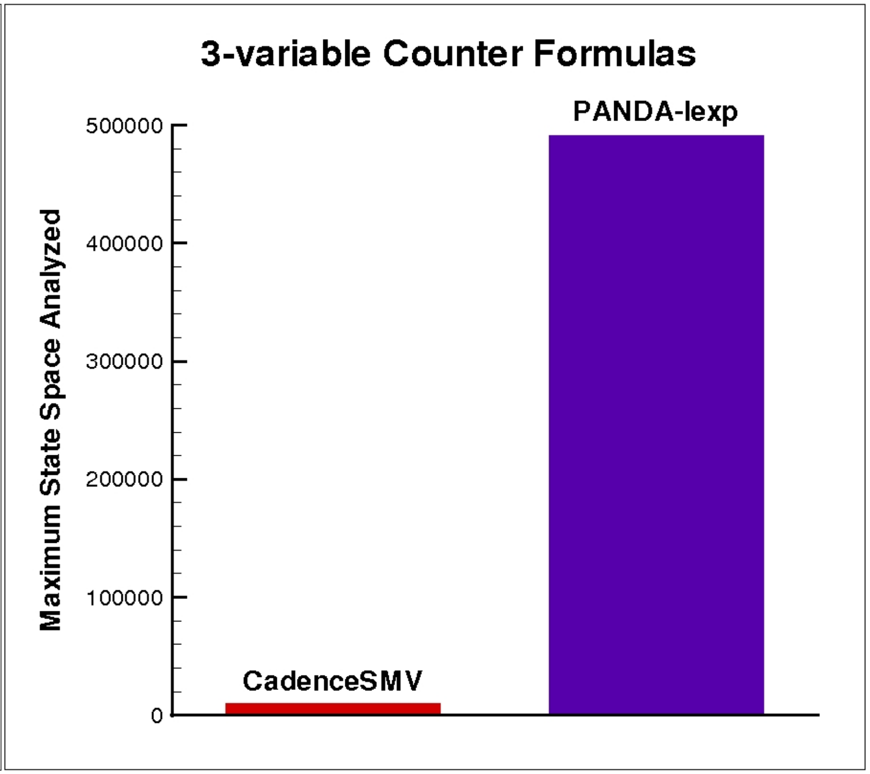
No transition form is best

- Sloppy encoding is the best transition form for all pattern formulas.
- Fussy encoding is better for all counter formulas.


Best fussy encoding vs best sloppy encoding:
3-variable, 140 length random formulas
Sloppy Encodings Model Analysis Times (sec)
Fussy Encodings Model Analysis Times (sec)
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}

No variable order is best, but `LEXM` is worst





A Multi-Encoding Approach

New tool: *PANDA* – Portfolio Approach to Navigate the Design of Automata

- *Multi-encoding approach:*
  - runs 23 PANDA encodings in parallel
  - terminates when the first job completes

Bottom Line: *exponential* improvement in performance over current techniques

Discussion

- Parallel computing has been a siren song in computer science for almost 50 years!

- While there are some success stories, parallelism, in general, has under-delivered.

Question: What does work?

Answer: Embarrassing parallelism!

My Advice: Do not be embarrassed to pick low-hanging fruit. It is the easiest to pick!