Performance of a SystemVerilog Sudoku Solver with Synopsys VCS

Jeremy Ridgeway

Avago Technologies, Ltd.

September 18, 2015

SNUG AUSTIN

Agenda

- Why do we care about Sudoku?
- Constraint Composition
- Results
- Conclusions

Constraint Solver Efficiency

- Write a bunch of constraints in the test bench
- No real concern with how well the solver will perform

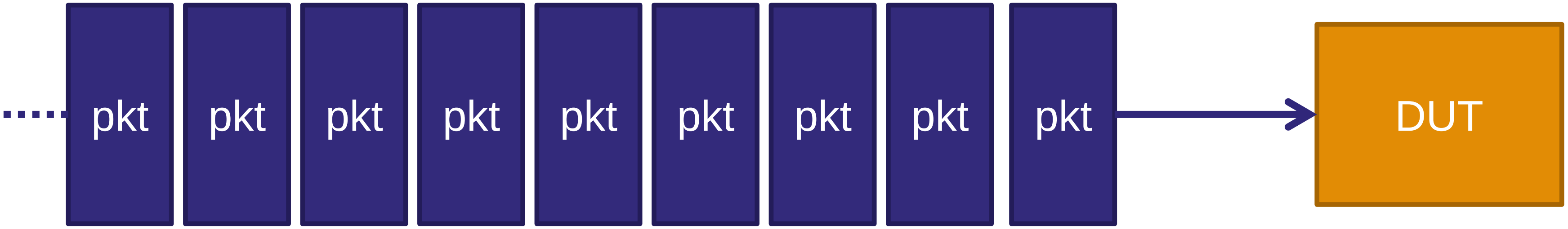
… until …

… much later in the project …

- Hey we're spending a lot of time in the constraint solver!
- Why is that?

Constraints in a Packet

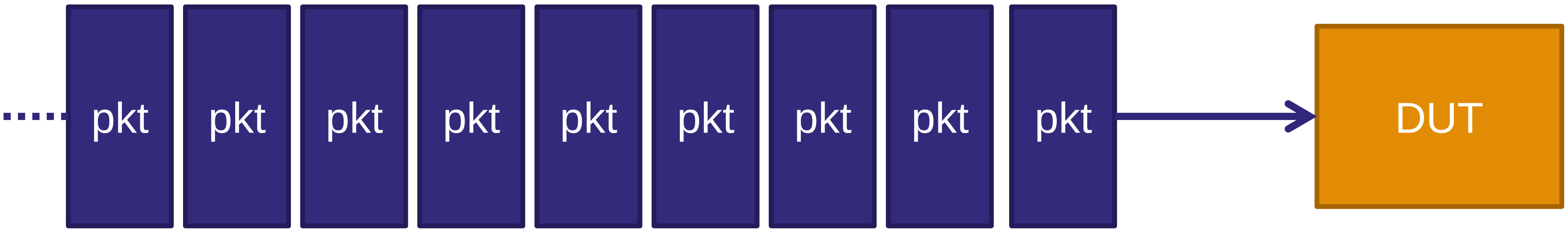
- How many packets are generated?
  - 10? 50,000?
- How long does each packet take to solve?
- Why does the solver take time?



Constraints in a Packet

- How many packets are generated?
- How long does each packet take to solve?
- Why does the solver take time?

- Seemingly simple constraint construction could lead to an inefficient constraint formula to solve
- Proprietary methods to analyze and solve formula



Proprietary Analysis

- Boolean Satisfaction is an Art
  - NP-complete problem
    - Nondeterministic Polynomial time

- Different solvers' approaches may be style-dependent

- Fast & efficient for one simulator may be slow for another

Why Sudoku?

- We don’t care about Sudoku

- Sudoku as testing vehicle
- Concrete Boolean problem
- Simple constraint composition

- Scalable
  - Boolean clauses are scalable (artifact of the CNF formula)
- Scales exponentially

- Saturate the solver
  - Small changes should have a noticeable affect

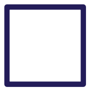
Sudoku Solving Performance

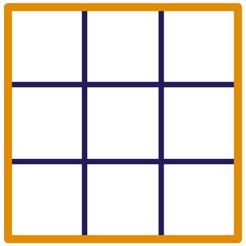
- Dissect the Sudoku constraints in SystemVerilog
- Compare VCS solver efficiency over differing constraint compositions
- Compare VCS solver efficiency against public solvers
  - MathSAT5, University of Trento, Trento, Italy
  - Yices, SRI International, Menlo Park, CA
  - Z3, Microsoft Research, Redmond, WA

Synopsys vs. Microsoft

Playing the game of Sudoku

Sudoku Defined

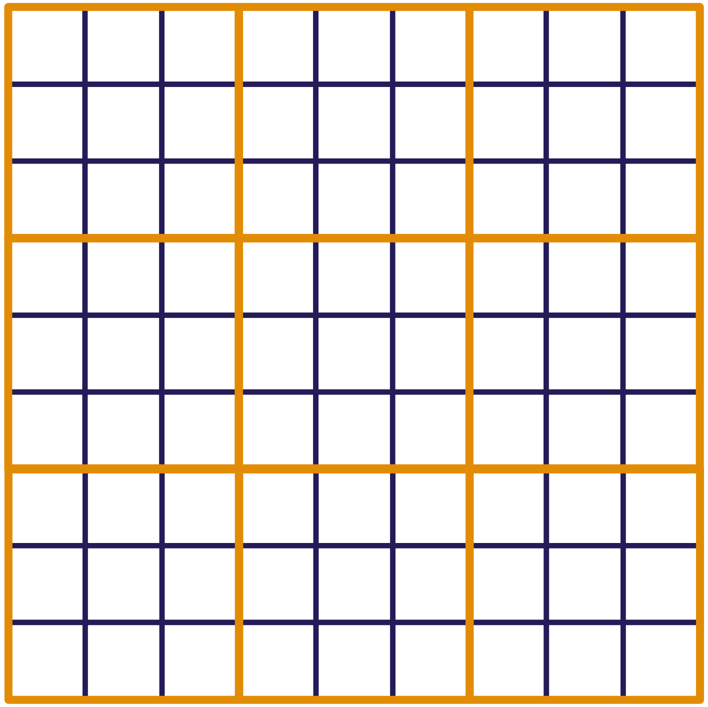




n = number of cells per square side

n^2 = number of cells in Latin Square

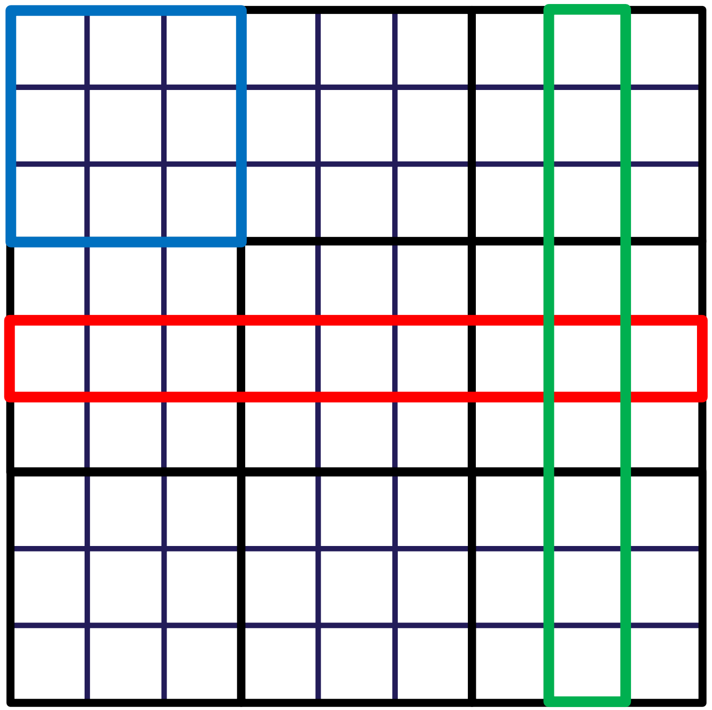
Each cell contains a unique value



9x9 Sudoku Board
$n^2 = 9$

Sudoku Defined

- Cell value is unique in:
  - Latin Square
  - Row
  - Column



9x9 Sudoku Board
$n^2 = 9$

Playing Sudoku

- Subset of cells have an initial value -- hints

	2			3	5			
			6		4		1	3
	6							5
			4			5		
3	9			6			7	2
		6			2			
2							8	
8	1		2		7			
			8	4			2	

Playing Sudoku

- Subset of cells have an initial value -- hints
- Fill in the empty cells with valid values to solve

	2			3	5			
			6		4		1	3
	6							5
			4			5		
3	9			6			7	2
		6			2			
2							8	
8	1		2		7			
			8	4			2	

Solving SystemVerilog Sudoku

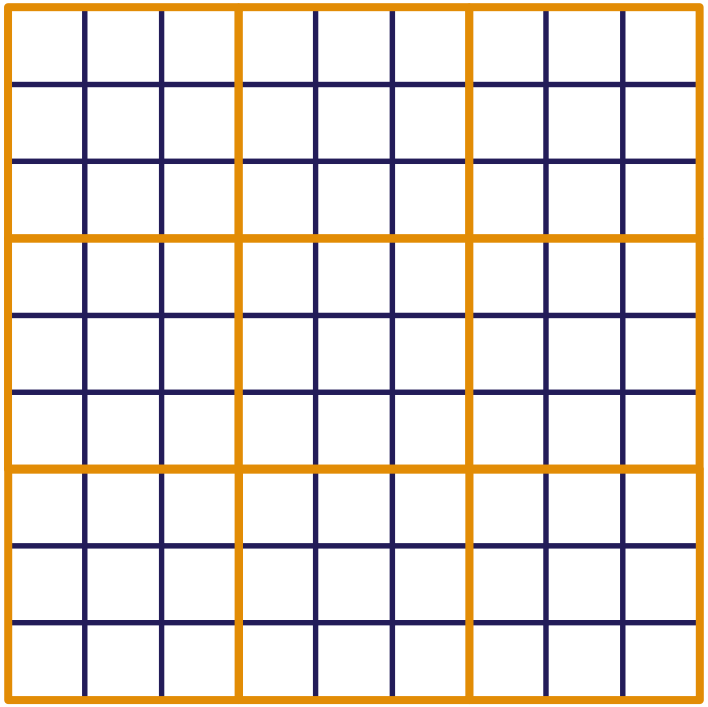
- Hint cell values reduce number of constraints for simulator’s solver to solve

	2			3	5			
			6		4		1	3
	6							5
			4			5		
3	9			6			7	2
		6			2			
2							8	
8	1		2		7			
			8	4			2	

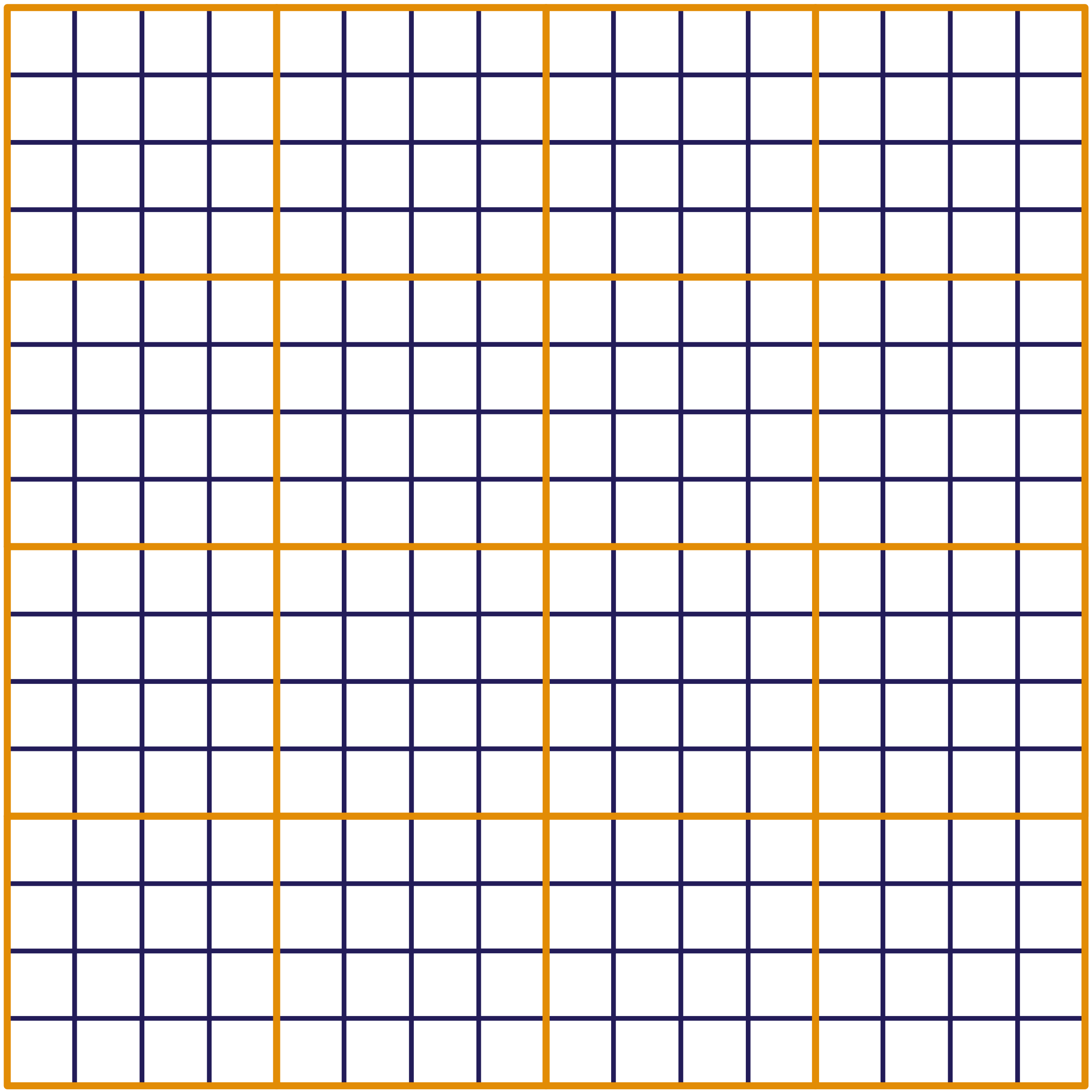
Solving SystemVerilog Sudoku

- Hint cell values reduce number of constraints for simulator's solver to solve

- Instead, we used a blank board and a random seed
  - Total 9x9 grids: 6.67×10^{21}



Sudoku Scaled



Scales in
Latin-Squares

16x16
Sudoku Board
$n^2 = 16$

Constraint Composition

Object Oriented Game Board

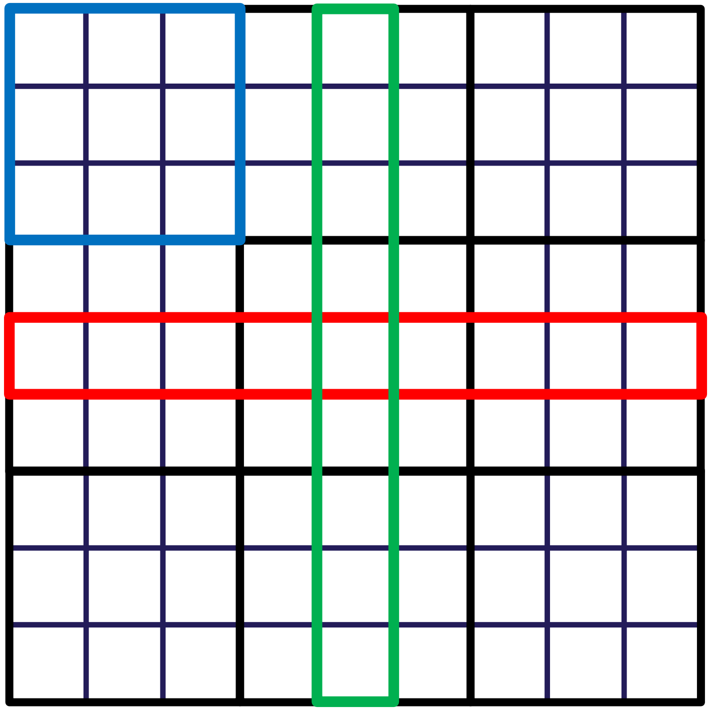
Composing Sudoku Formula

- Assume cell takes integer value greater than zero

- Cell constraint

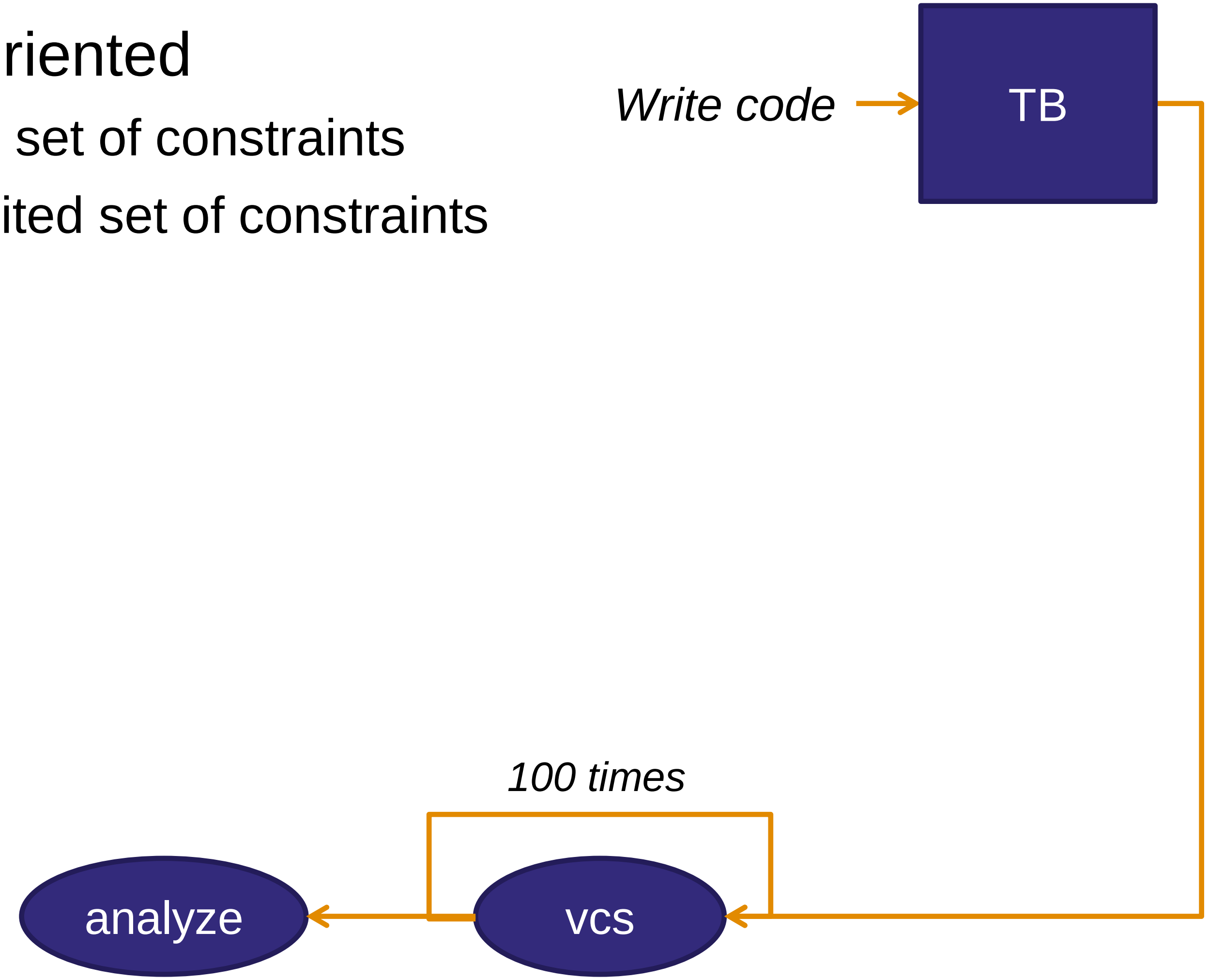
  cell.value **inside** {[1:n^2]};

- Square constraint
- Row constraint
- Column constraint



Sudoku Test benches

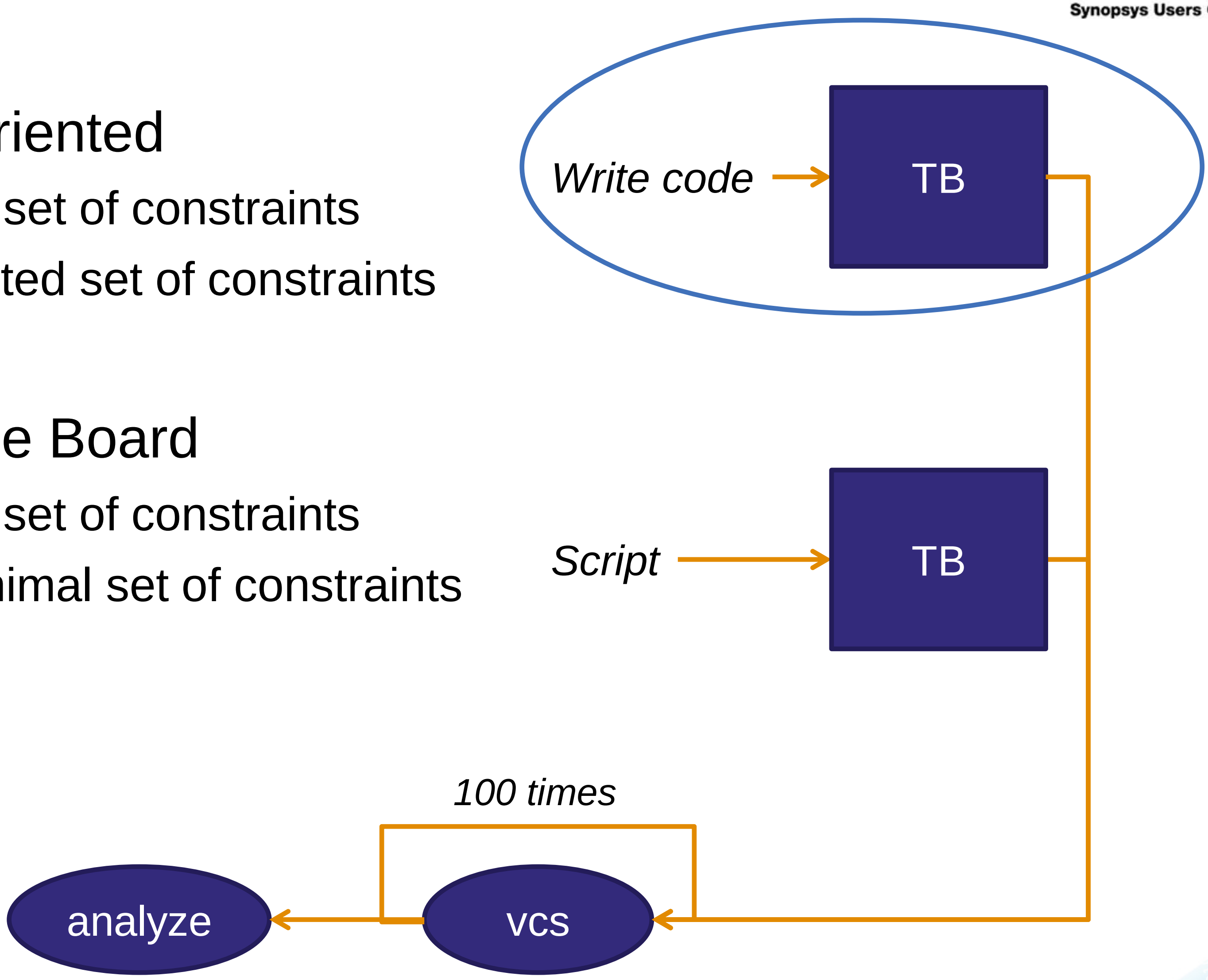
1. Object Oriented
   a) with full set of constraints
   b) with limited set of constraints



Sudoku Test benches

1. Object Oriented
   a) with full set of constraints
   b) with limited set of constraints

2. Flat Game Board
   a) with full set of constraints
   b) with minimal set of constraints



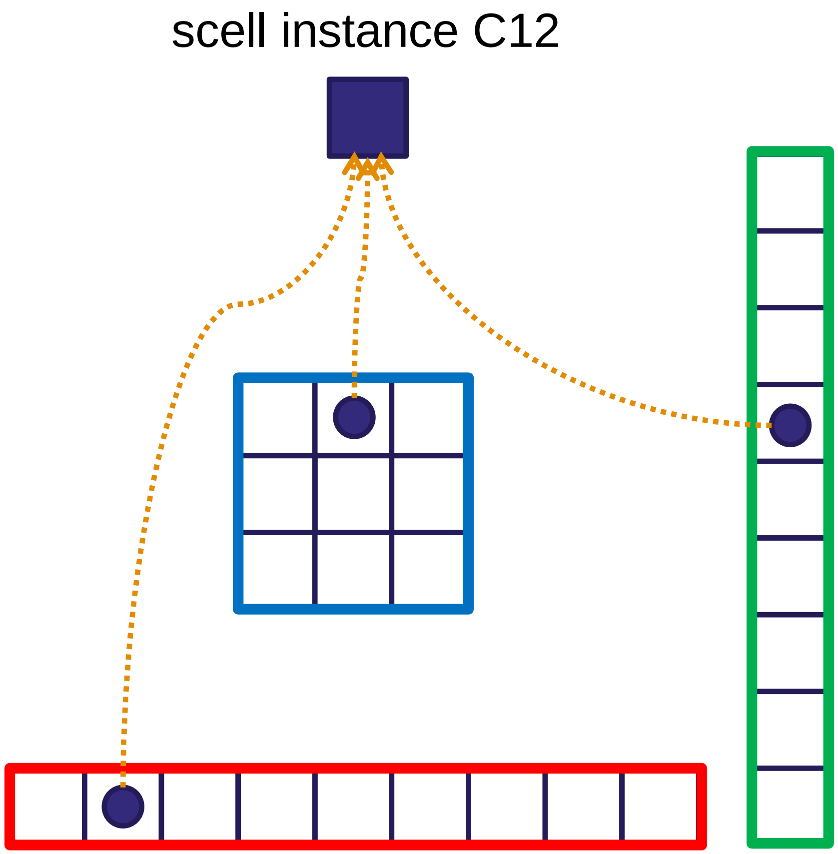
#1 Object Oriented Test Bench

Single cell instance in Sudoku board

```
`define N2 9

class scell;
  rand int val;
  constraint c {
    val inside { [1:`N2] }; }
endclass


class slist;
  rand scell cel[ ];
  constraint c {
    foreach ( cel[i] )
      foreach ( cel[j] )
        i != j -> cel[i].val != cel[j].val;
  }
endclass
```



#1 Object Oriented Test Bench

Boolean formula to solve

```
`define N2 9

class scell;
  rand int val;
  constraint c {
    val inside { [1:`N2] }; }
endclass

class slist;
  rand scell cel[ ];
  constraint c {
    foreach ( cel[i] )
      foreach ( cel[j] )
        i != j -> cel[i].val != cel[j].val;
  }
endclass
```

CNF formula

val >= 1 && val <= `N2

Clause   Clause

#1 Object Oriented Test Bench

Boolean formula to solve

```
`define N2 9

class scell;
  rand int val;
  constraint c {
    val inside { [1:`N2] }; }
endclass

class slist;
  rand scell cel[ ];
  constraint c {
    foreach ( cel[i] )
      foreach ( cel[j] )
        i != j -> cel[i].val != cel[j].val;
  }
endclass
```

CNF formula

val >= 1 && val <= `N2

val > 1 || val == 1 ←- - -→ !(val < 1)

predicate a || b ←- - -→ !(x)

literal A || B ←- - -→ X

#1 Object Oriented Test Bench

Boolean formula to solve

```
`define N2 9

class scell;
  rand int val;
  constraint c {
    val inside { [1:`N2] }; }
endclass

class slist;
  rand scell cel[ ];
  constraint c {
    foreach ( cel[i] )
      foreach ( cel[j] )
        i != j -> cel[i].val != cel[j].val;
  }
endclass
```

CNF formula

val >= 1 && val <= `N2

val > 1 || val == 1

predicate　a || b

literal　A || B

unit clause

(~~A~~ || B) && (C || D) && (B)

C18: val == 1

Lazy Solving Approach

#1 Object Oriented Test Bench

a) full set of constraints

```
`define N2 9

class scell;
  rand int val;
  constraint c {
    val inside { [1:`N2] }; }
endclass
```

val >= 1 && val <= `N2

Crow column: C12, C18
Crow_index col_index: C01, C07

```
class slist;
  rand scell cel[ ];
  constraint c {
    foreach ( cel[i] )
      foreach ( cel[j] )
        i != j -> cel[i].val != cel[j].val;
  }
endclass
```

i	j	cel[i]	cel[j]
~~(0 == 0 || C11 != C11) &&~~			
(0 == 1 || C11 != C12) &&			
…			
~~(1 == 0 || C12 != C11) &&~~			
…			

Row 1 Constraint

reduce?

optimize?

#1 Object Oriented Test Bench

b) limited set of constraints

```
`define N2 9

class scell;
  rand int val;
  constraint c {
    val inside { [1:`N2] }; }
endclass

class slist;
  rand scell cel[ ];
  constraint c {
    foreach ( cel[i] )
      foreach ( cel[j] )
        i > j -> cel[i].val != cel[j].val;
  }
endclass
```

val >= 1 && val <= `N2

Crow column: C12, C18
Crow_index col_index: C01, C07

i	j	cel[i]	cel[j]

~~(0 <= 0 || C11 != C11) &&~~ reduce?
~~(0 <= 1 || C11 != C12) &&~~ reduce?
…
(1 <= 0 || C12 != C11) &&
…

Row 1 Constraint

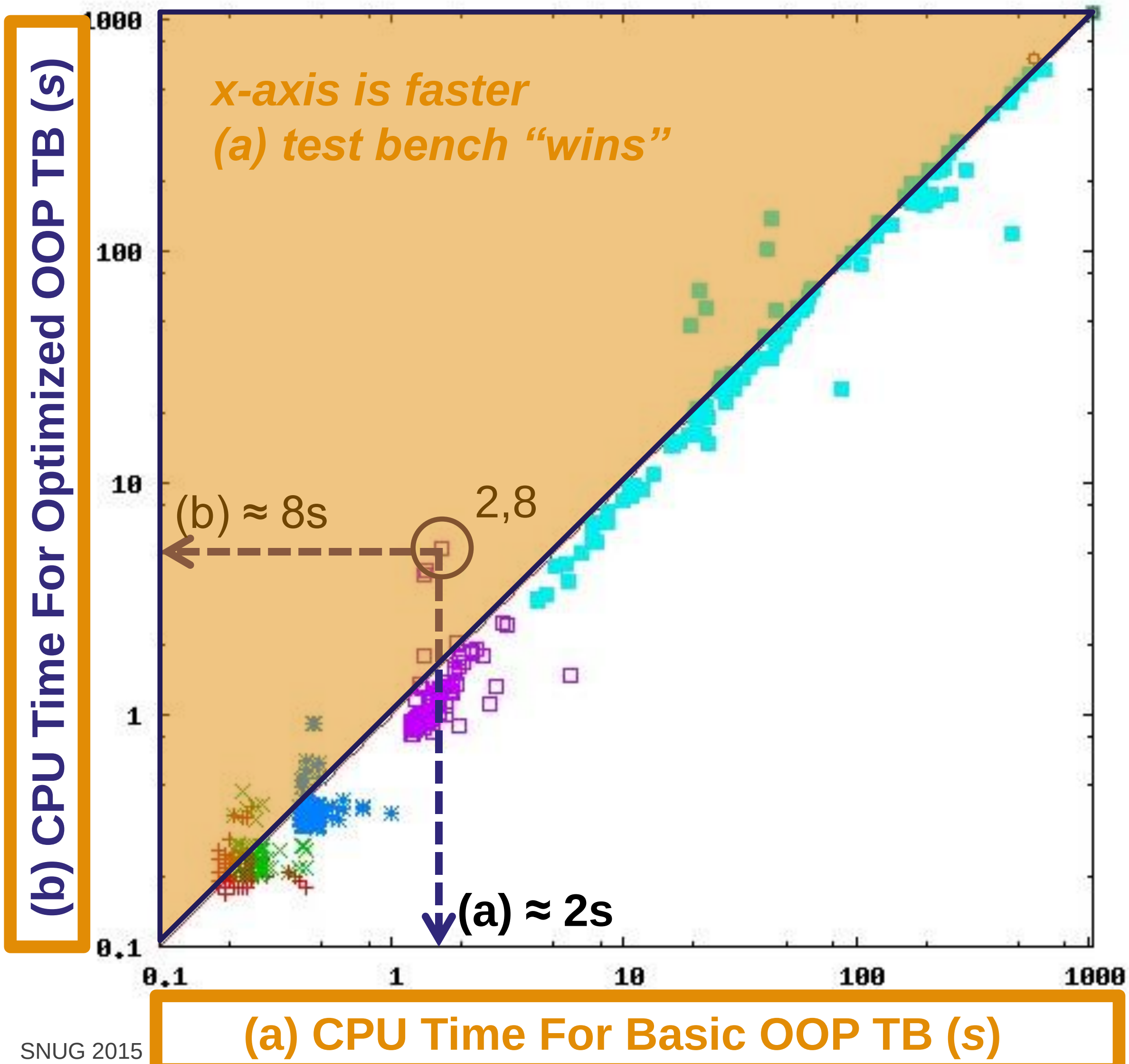
#1 Object Oriented Test Bench

Optimized TB halves the solving space

	Sudoku Board	# of Cells	Tot # scell clauses	# slist clauses	# Total clauses	Reduction
(a) Basic OOP Test bench (!=)	4x4	16	32	64	224	= 32 + (64*3)
	9x9	81	162	729	2,349	-
	16x16	256	512	4,096	12,800	-
	25x25	625	1,250	15,625	48,125	-
	36x36	1,296	2,592	46,656	142,560	-
	49x49	2,401	4,802	117,649	357,749	-
(b) Optimized OOP Test bench (>)	4x4	16	32	24	104	53.67%
	9x9	81	162	324	1,134	51.7%
	16x16	256	512	1,920	6,272	51%
	25x25	625	1,250	7,500	23,750	50.7%
	36x36	1,296	2,592	22,680	70,632	50.5%
	49x49	2,401	4,802	57,624	177,674	50.3%

#1 Object Oriented Test Bench

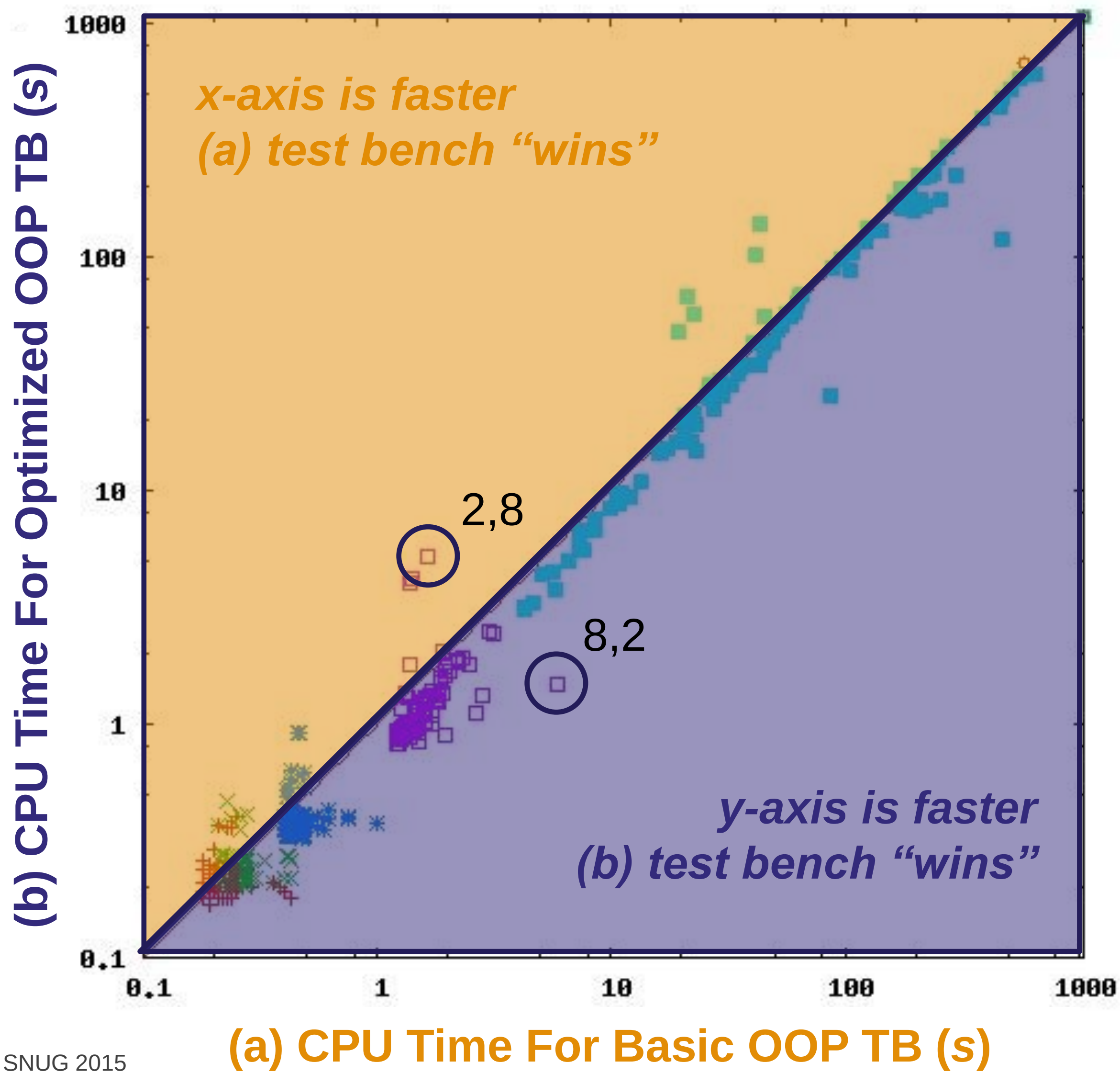
Basic vs. Optimized 4 using a const variable




```
class scell;
  rand int val;
  const int max = `N2;
  constraint c {
    val inside { [1:max] }; }
endclass
```

#1 Object Oriented Test Bench

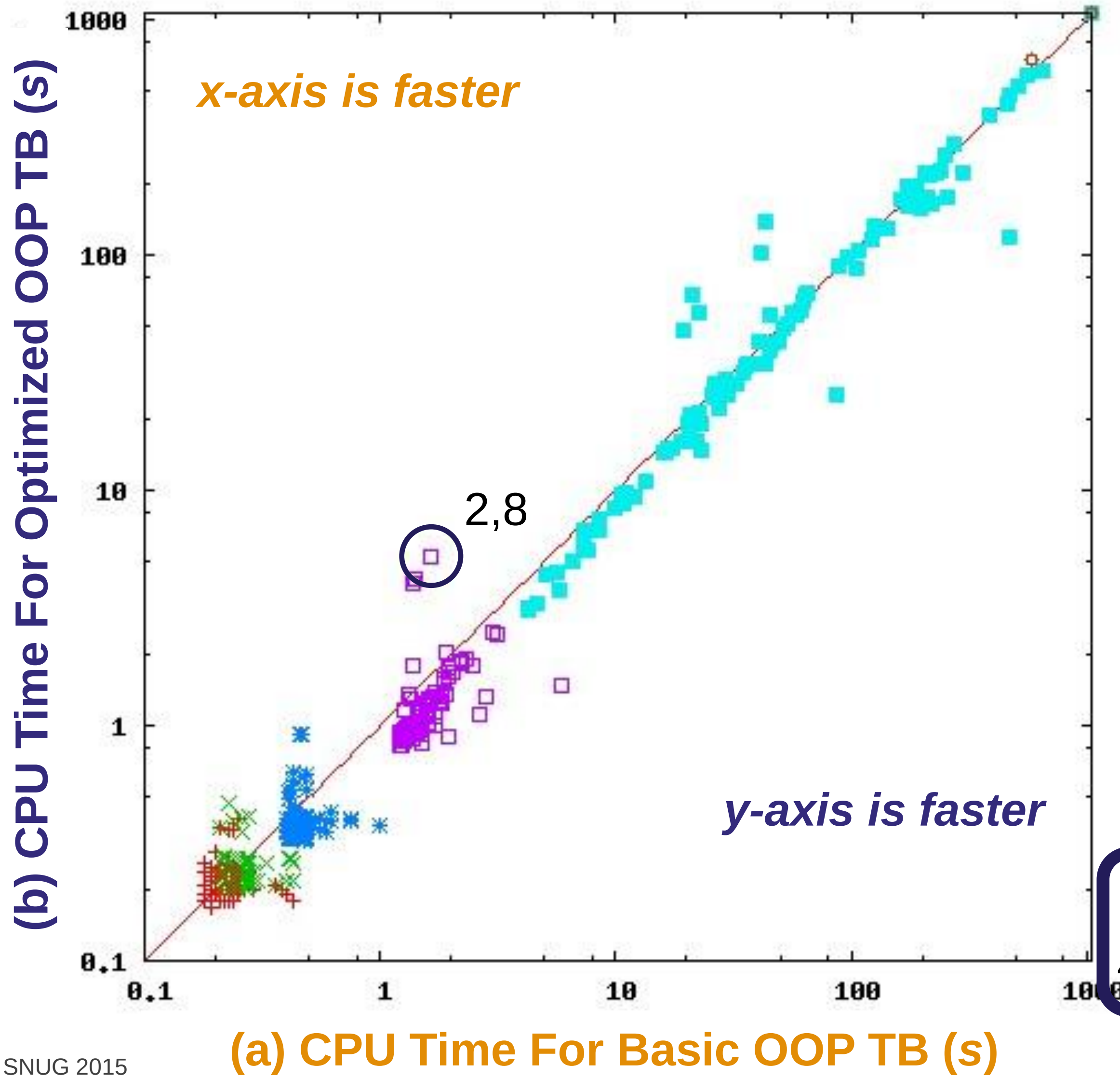
Basic vs. Optimized 4 using a const variable




```
class scell;
  rand int val;
  const int max = `N2;
  constraint c {
    val inside { [1:max] }; }
endclass
```

#1 Object Oriented Test Bench

Basic vs. Optimized 4 using a const variable



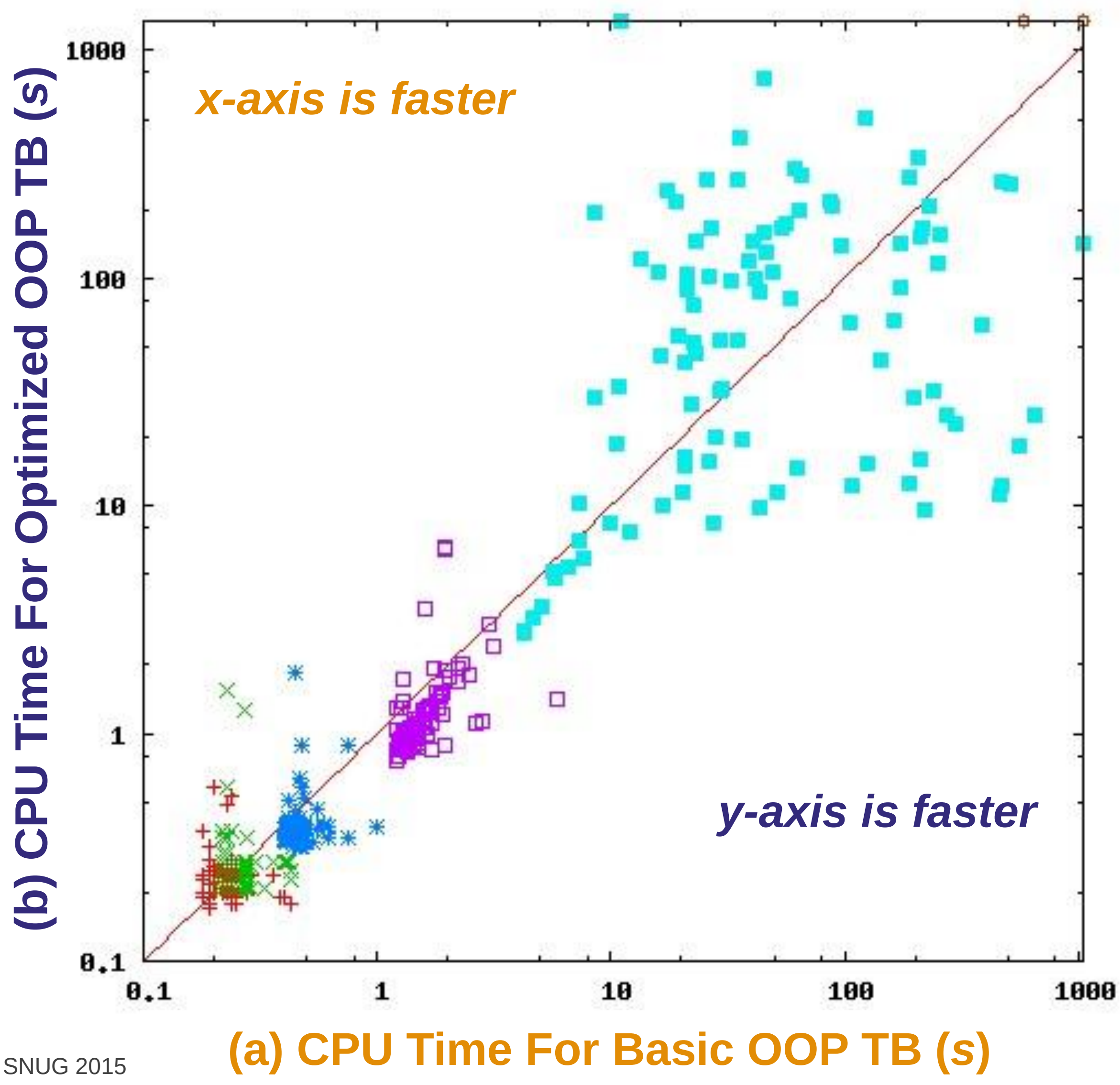
Average Solving Time with VCS (seconds)

Board	(a)	(b)
4x4	0.23	0.22
9x9	0.26	0.24
16x16	0.47	0.39
25x25	1.60	1.25
36x36	48,125	23,750
49x49	*587.48*	*679.28*

49x49 = 1 solved (a), 1 solved (b)

#1 Object Oriented Test Bench

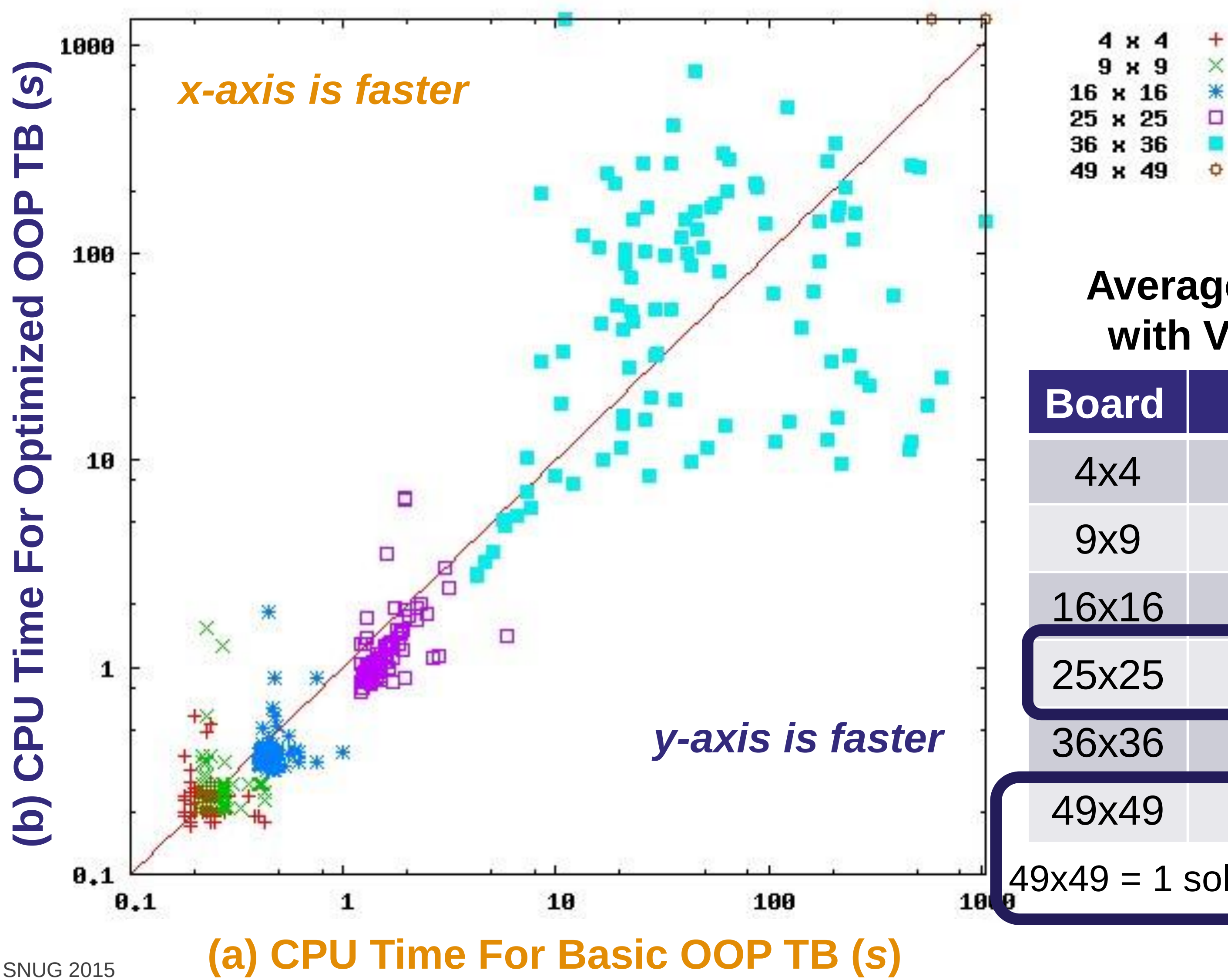
Basic vs. Optimized 4 using a `define




```
class scell;
  rand int val;
  constraint c {
    val inside { [1:`N2] }; }
endclass
```

#1 Object Oriented Test Bench

Basic vs. Optimized 4 using a `define



Average Solving Time with VCS (seconds)

Board	(a)	(b)
4x4	0.23	0.23
9x9	0.26	0.28
16x16	0.47	0.40
25x25	1.60	1.27
36x36	48,125	23,750
49x49	*587.48*	-

49x49 = 1 solved (a), 0 solved (b)

#1 Object Oriented Test Bench

b) limited set of constraints

Synopsys Users Group

```
`define N2 9

class scell;
  rand int val;
  constraint c {
    val inside { [1:`N2] }; }
endclass
```

val >= 1 && val <= `N2

Crow column: C12, C18
Crow_index col_index: C01, C07

Empirically, “yes” formula is reduced. No benefit rewriting.

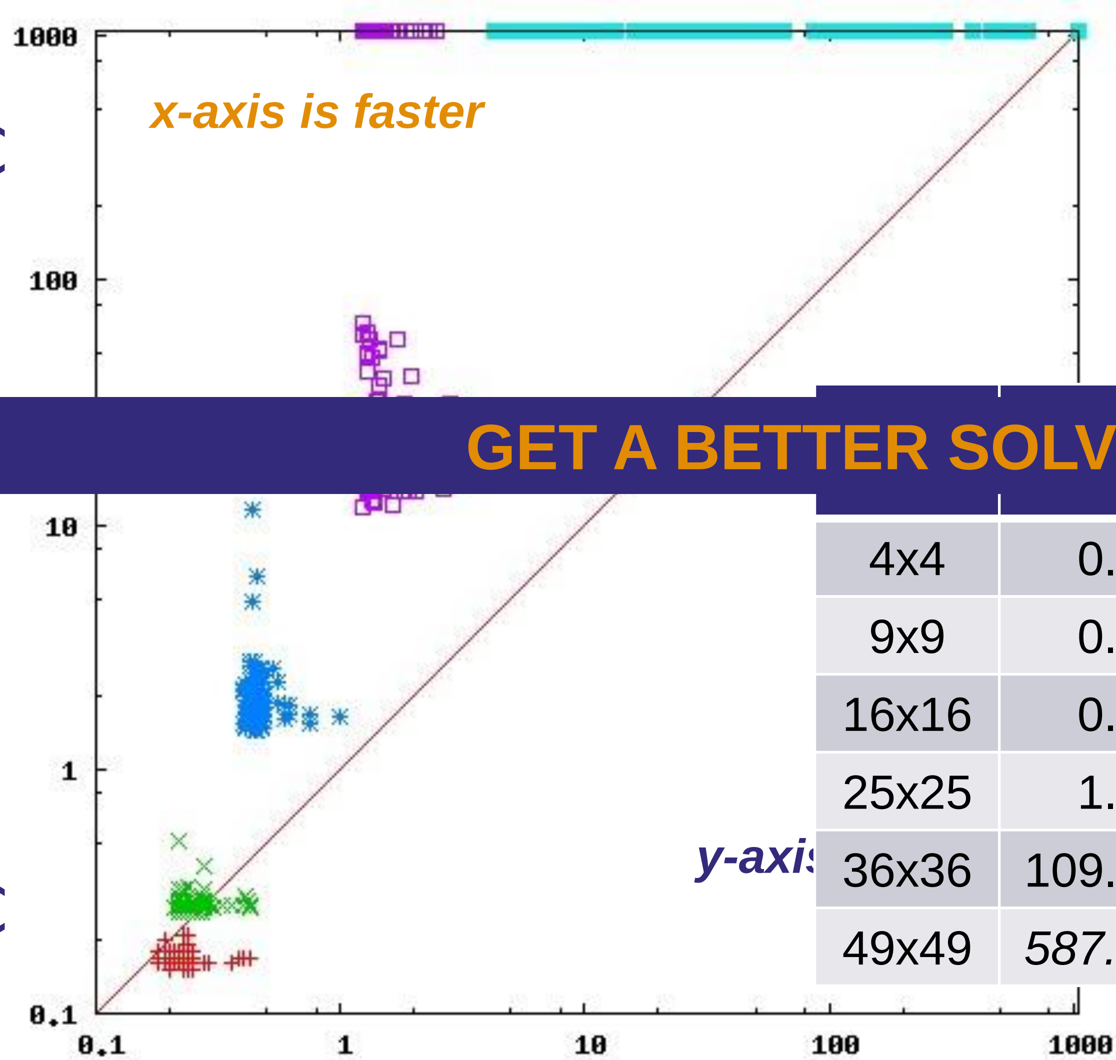
```
class slist;
  rand scell cel[ ];
  constraint c {
    foreach ( cel[i] )
      foreach ( cel[j] )
        i > j -> cel[i].val != cel[j].val;
  }
endclass
```

~~(0 <= 0 || C11 != C11) &&~~ reduce?
~~(0 <= 1 || C11 != C12) &&~~ reduce?
…
(1 <= 0 || C12 != C11) &&
…

Row 1 Constraint

#1 Object Oriented Test Bench

VCS 2014 is the clear winner



Average Solving Time with VCS (seconds)

GET A BETTER SOLVER

4x4	0.23	100	0.17	100
9x9	0.26	100	0.28	100
16x16	0.47	100	2.03	100
25x25	1.60	100	25.58	63
36x36	109.32	100	-	0
49x49	*587.48*	1	-	0

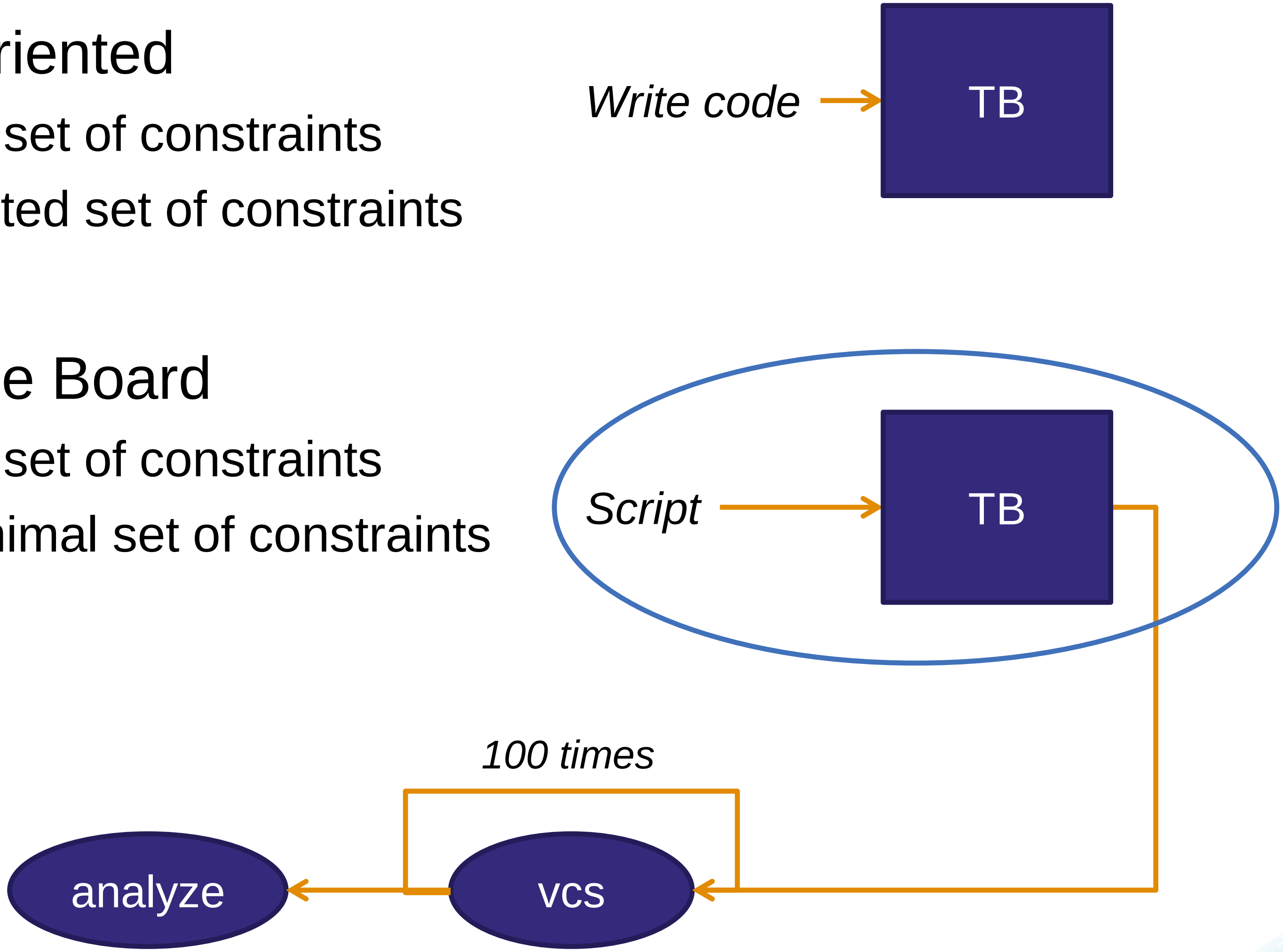
Constraint Composition

Flat Game Board

Sudoku Test benches

1. Object Oriented
   a) with full set of constraints
   b) with limited set of constraints

2. Flat Game Board
   a) with full set of constraints
   b) with minimal set of constraints



#2 Flat Test Bench

a) full set of constraints

```
class sboard;
  rand int cel_1_1;
  rand int cel_1_2;
  …
  rand int cel_9_9;

  constraint cells {
    cel_1_1 inside { [1:9] };
    cel_1_2 inside { [1:9] };
    …
    cel_9_9 inside { [1:9] };
  }
constraint rows {
    cel_1_1 != cel_1_2;
    cel_1_1 != cel_1_3;
    …
    cel_1_2 != cel_1_1;
    …
  }
```

```
  constraint cols {
    cel_1_1 != cel_2_1;
    cel_1_1 != cel_3_1;
    …
    cel_2_1 != cel_1_1;
    …
  }
  constraint sqrs {
    cel_1_1 != cel_1_2;
    cel_1_1 != cel_1_3;
    cel_1_1 != cel_2_1;
    …
    cel_2_1 != cel_1_1;
    …
  }
endclass
```

optimize between blocks?

optimize within block?

optimize within block?

optimize between blocks?

optimize within block?

#2 Flat Test Bench

b) minimal set of constraints

```
class sboard;
  rand int cel_1_1;
  rand int cel_1_2;
  …
  rand int cel_9_9;

  constraint cells {
    cel_1_1 inside { [1:9] };
    cel_1_2 inside { [1:9] };
    …
    cel_9_9 inside { [1:9] };
  }
constraint rows {
    cel_1_1 != cel_1_2;
    cel_1_1 != cel_1_3;
    …
}
  constraint cols {
    cel_1_1 != cel_2_1;
    cel_1_1 != cel_3_1;
    …
  }
  constraint sqrs {
    cel_1_1 != cel_1_5;
    cel_1_1 != cel_1_6;
    …
  }
endclass
```

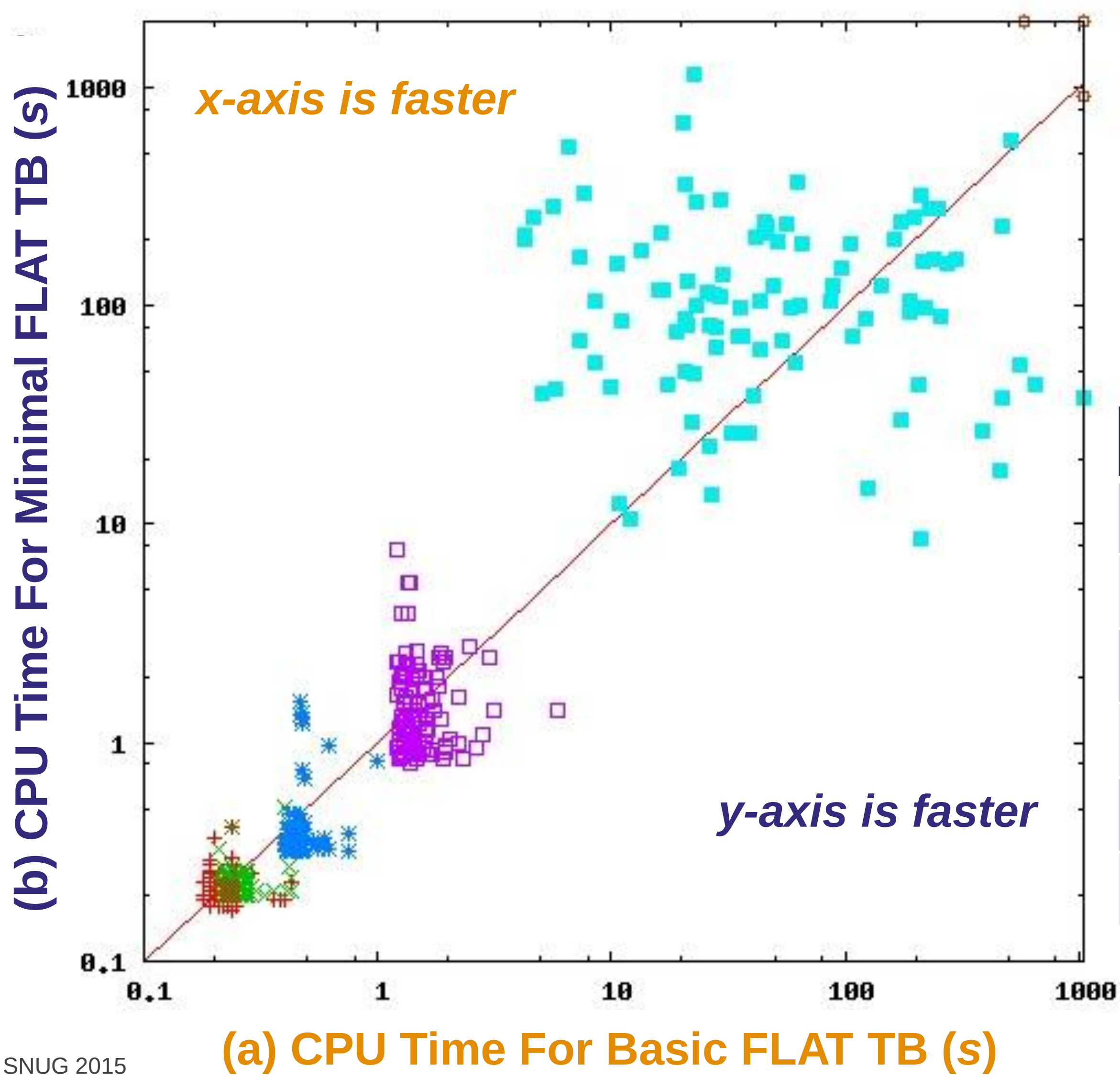
#2 Flat Test Bench

Minimal TB at least halves the solving space

	Sudoku	# of cells	# cell clauses	# row/col clauses	# sqr clauses	# Total clauses	Reduction
(a) Basic FLAT Test bench	4x4	16	32	48	48	176	-
	9x9	81	162	648	648	2,106	-
	16x16	256	512	3,840	3,840	12,032	-
	25x25	625	1,250	15,000	15,000	46,250	48,125
	36x36	1,296	2,592	43,360	43,360	138,672	-
	49x49	2,401	4,802	115,248	115,248	350,546	-
(b) Minimal FLAT Test bench	4x4	16	32	24	8	88	50.00%
	9x9	81	162	324	162	972	53.85%
	16x16	256	512	1,920	1,152	5,504	54.26%
	25x25	625	1,250	7,500	5,000	21,250	23,750 05%
	36x36	1,296	2,592	22,680	16,200	64,152	53.74%
	49x49	2,401	4,802	57,624	43,218	163,268	53.42%

#2 Flat Test Bench

Basic vs. Optimized 3 with unique clauses



Average Solving Time with VCS (seconds)

Board	(a)	(b)
4x4	0.28	0.22
9x9	0.25	0.22
16x16	0.58	0.42
25x25	1.90	1.61
36x36	46,250	21,250
49x49	*1,700.46*	*1,459.09*

49x49 = 2 solved (a), 2 solved (b)

#2 Flat Test Bench

OOP vs FLAT

- OOP test bench outperforms larger FLAT test benches
- FLAT solved more of the largest test benches

Average Solving Time with VCS (seconds)

Sudoku	OOP (a)	OOP (b)	FLAT(a)	FLAT(b)
4x4	0.23	0.22	0.28	0.22
9x9	0.26	0.24	0.25	0.22
16x16	0.47	0.39	0.58	0.42
25x25	1.60	1.25	1.90	1.61
36x36	109.32	104.32	137.72	148.51
49x49	*587.48*	*679.28*	*1,700.46*	*1,459.09*

49x49 = OOP(a) 1 solved, OOP (b) 1 solved,
FLAT(a) 2 solved, FLAT (b) 2 solved

#2 Flat Test Bench

OOP vs MATHSAT5 vs YICES

- Same flat board in SMT2 language
- Showing unique case for Mathsat5, YICES, and Z3

Average Solving Time (seconds)

Sudoku	OOP (a)	OOP (b)	MSAT5	YICES	Z3
4x4	0.23	0.22	0.02	0.00	0.01
9x9	0.26	0.24	21.99	7.70	174.64
16x16	0.47	0.39	-	-	
25x25	1.60	1.25	-	-	
36x36	109.32	104.32	-	-	
49x49	*587.48*	*679.28*	-	-	

49x49 = OOP(a) 1 solved, OOP (b) 1 solved, MSAT, YICES, and Z3 did not solve any board greater than 9x9 in time window.

Conclusions

- Hierarchical constraints about as good as flat
  - Object oriented vs. all in one class
- Constant variable access about as good as hard values
  - const vs. `define (parameter)
- Composing the constraint to limit its size has very little impact on solving time
  - Solver effectively reduces the formula

Take Aways

- Readability of constraint is most important
- Maintainability of constraint is important
- Obtaining an efficient solver is important
  - VCS 2014

Pictures

Man Playing Sudoku on Subway While Women Look Over His Shoulder,
https://www.flickr.com/photos/wnyc/19953592074

el viejo, el mar y el sudok,
https://www.flickr.com/photos/altamar/2652472523

last august friday, with Flor,
https://www.flickr.com/photos/perfumedsecrets/7934536812/

Art in Embassies - Jonathan Anderson, Construction (no. 6), 2009,
https://www.flickr.com/photos/us_embassy_newzealand/6071070490/

How do students measure up?,
https://www.flickr.com/photos/venosdale/4376443940/

summer 4,
https://www.flickr.com/photos/26315381@N06/3403505001/

Thank You

Constraint Solver Efficiency

- Random verification relies on efficient and constrained stimulus generation
- Seemingly simple constraint construction can lead to an inefficient constraint formula to solve
- Proprietary methods to analyze and solve formula