

# **SESS**

## **Rapid3D 20X Performance Improvement**

Greg Rollins

7/19/2010

# Abstract

- Due to changing product goals the Raphael-NXT product was completely rewritten and (renamed Rapid-3D)
- Surprisingly good results were obtained
  - 20X speed increase
  - 3X memory reduction
  - Reduction in lines of code from 60K to 10K
  - Equal accuracy
  - Multi threading added
- This presentation will outline how it was done

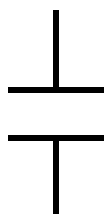
# Overview

- Product history
- Random walk capacitance extraction
- The change in product goals
- Changes to “Math & Physics”
- Changes to “Software”
- Final results
- How to tell if it is possible to do better?

# Raphael-NXT Product history

- 1990 Random walk method “invented” by Iverson and LaCoz at RPI
- 1993 Quickcap & Random Logic Corp launched
- ?-2005 Quickcap marketed by Synopsys through OEM
  - Peak sales around \$2.5M
- 2002 development of Raphael-NXT started by ATG mainly in India.
  - ~6 man years invested
- 2005 Random Logic Corp acquired by Magma
- 2005 Raphael-NXT transferred to TCAD group
  - Sales of >\$10M expected
  - ~2 man years invested
- 2006 Raphael-NXT replaces Quickcap at Synopsys
- 2009 Silicon Front Line Tech offers fast random walk solver
- 2009 Raphael-NXT transferred to Star-RCXT group,
  - Peak sales only reached \$1.5M
  - Discontinued as standalone product and rolled into STAR
  - Rewrite begins
- 2010.06 Rapid-3D, Synopsys fast random walk solver offered to customers.

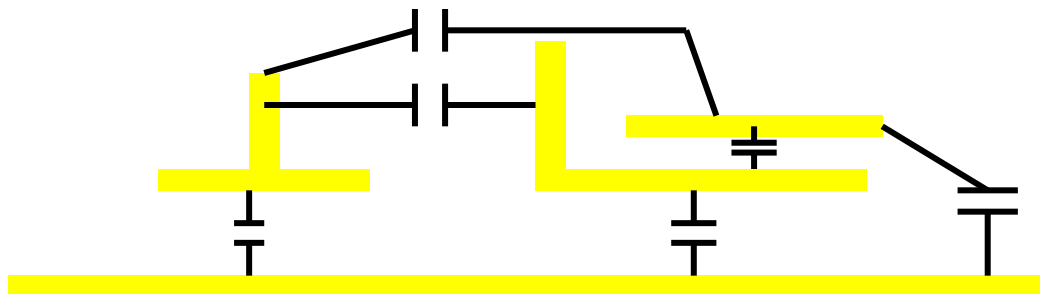
# CAPACITANCE

Capacitance  $\rightarrow$    $\frac{\epsilon A}{d}$

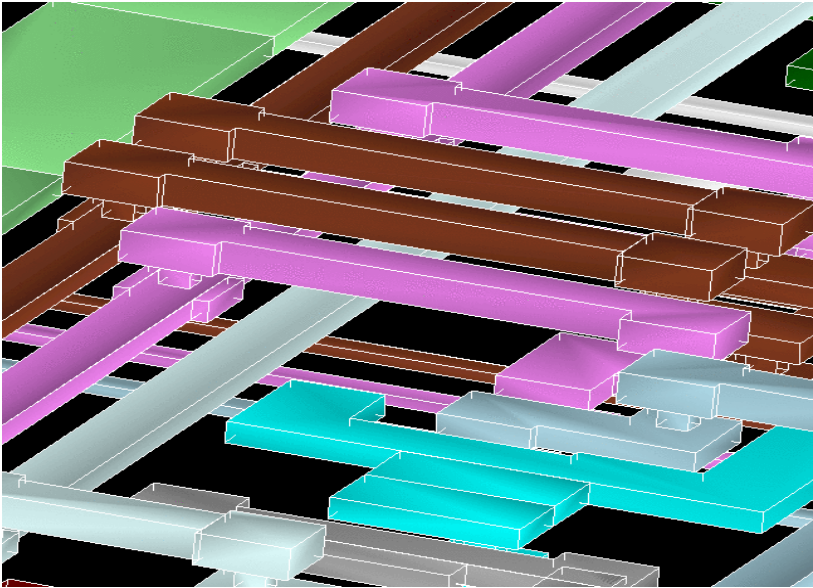
Capacitance exists between each pair of conductor in space & is function of conductor and dielectric geometry

Assumed to be independent of voltage

N Conductors  $\rightarrow$   $N*(N-1)/2$  capacitances



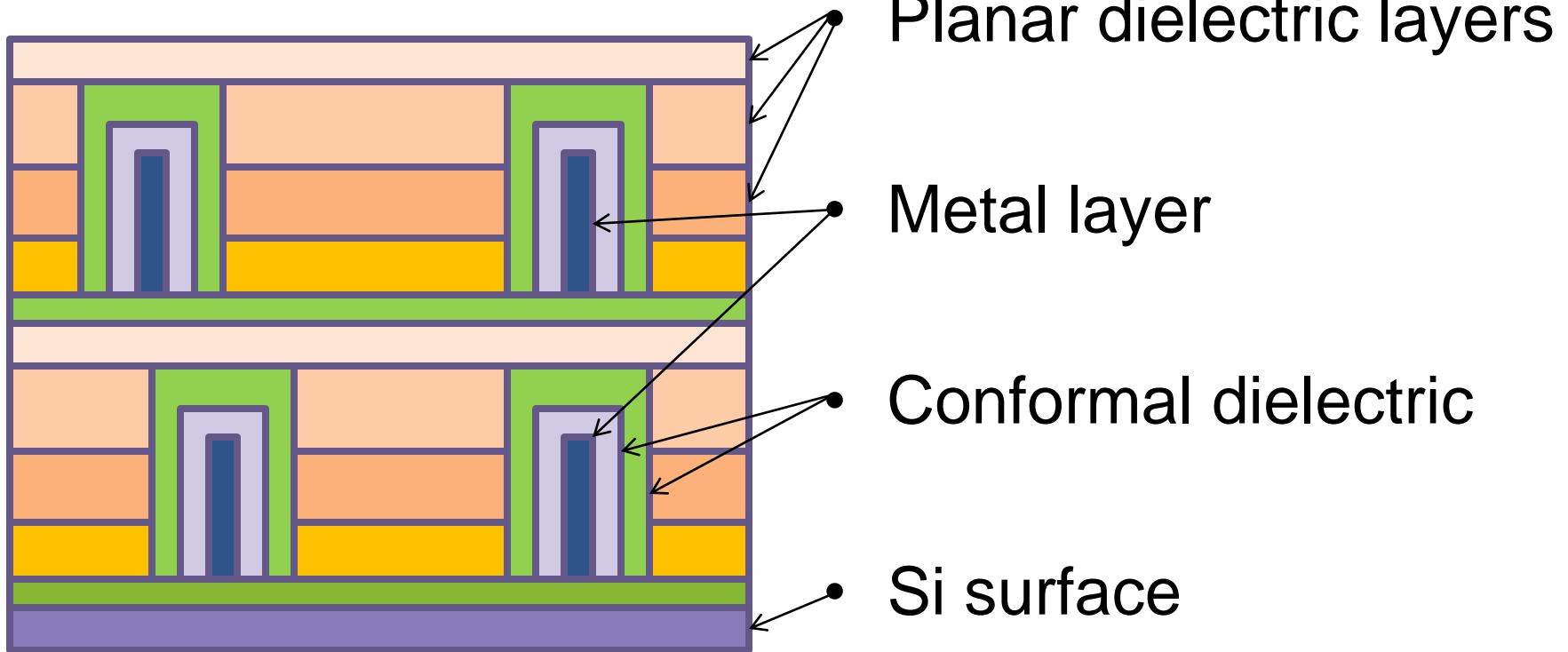
# Creating the structure



1. The hierarchy is flattened
2. Touching metal is connected into nets
3. Polygons are extruded in the “z” direction (perpendicular to the silicon surface)
4. Etch-vs-width&spacing and density based corrections are applied
5. Planar dielectrics are added
6. Conformal dielectrics are placed around the metal

# Chip structure showing dielectrics

## Cross section view



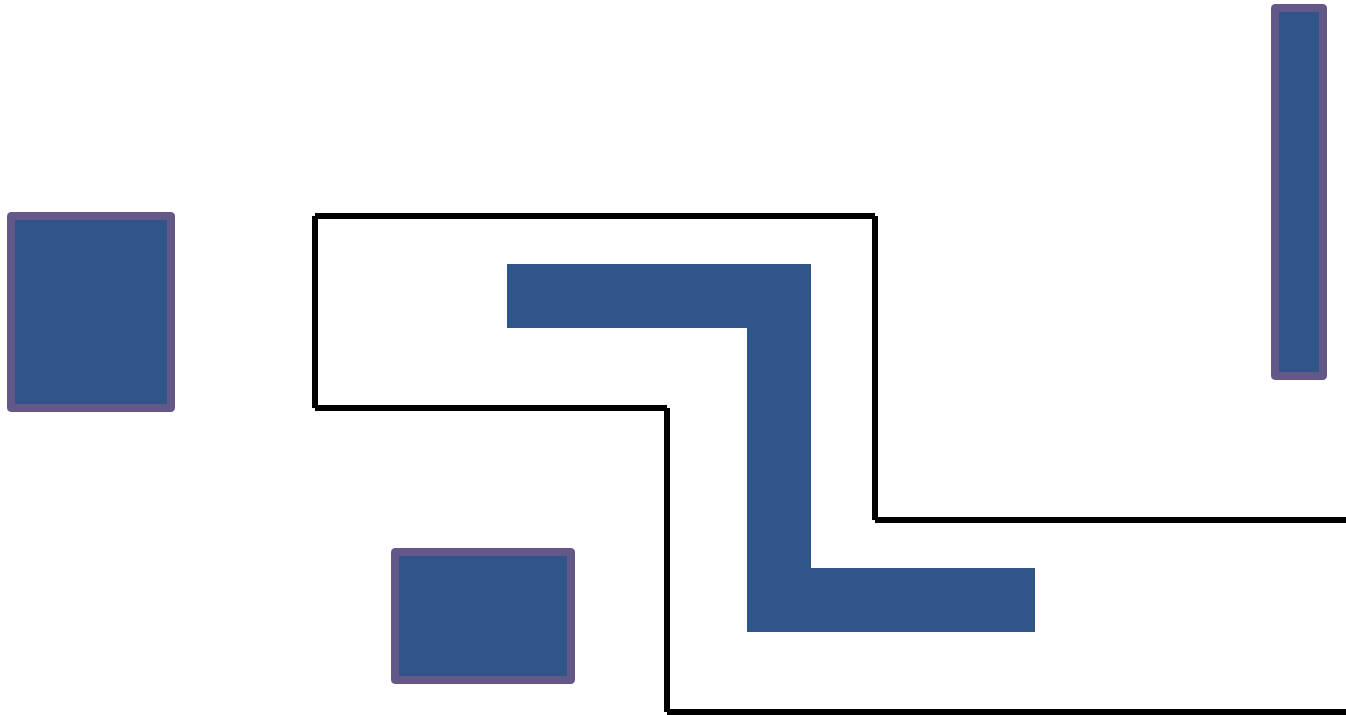
# Random walk method

- Calculates capacitance in 3D space
- Geometries consist of 3D boxes
- All layers must be considered at once
- Dielectric layers add to complexity
- Main advantage is ability to handle designs with millions of polygons and produce accuracy  $\sim 1\%$
- Memory proportional to design size
- $\text{CPU} \propto 1/(\text{accuracy})^2$

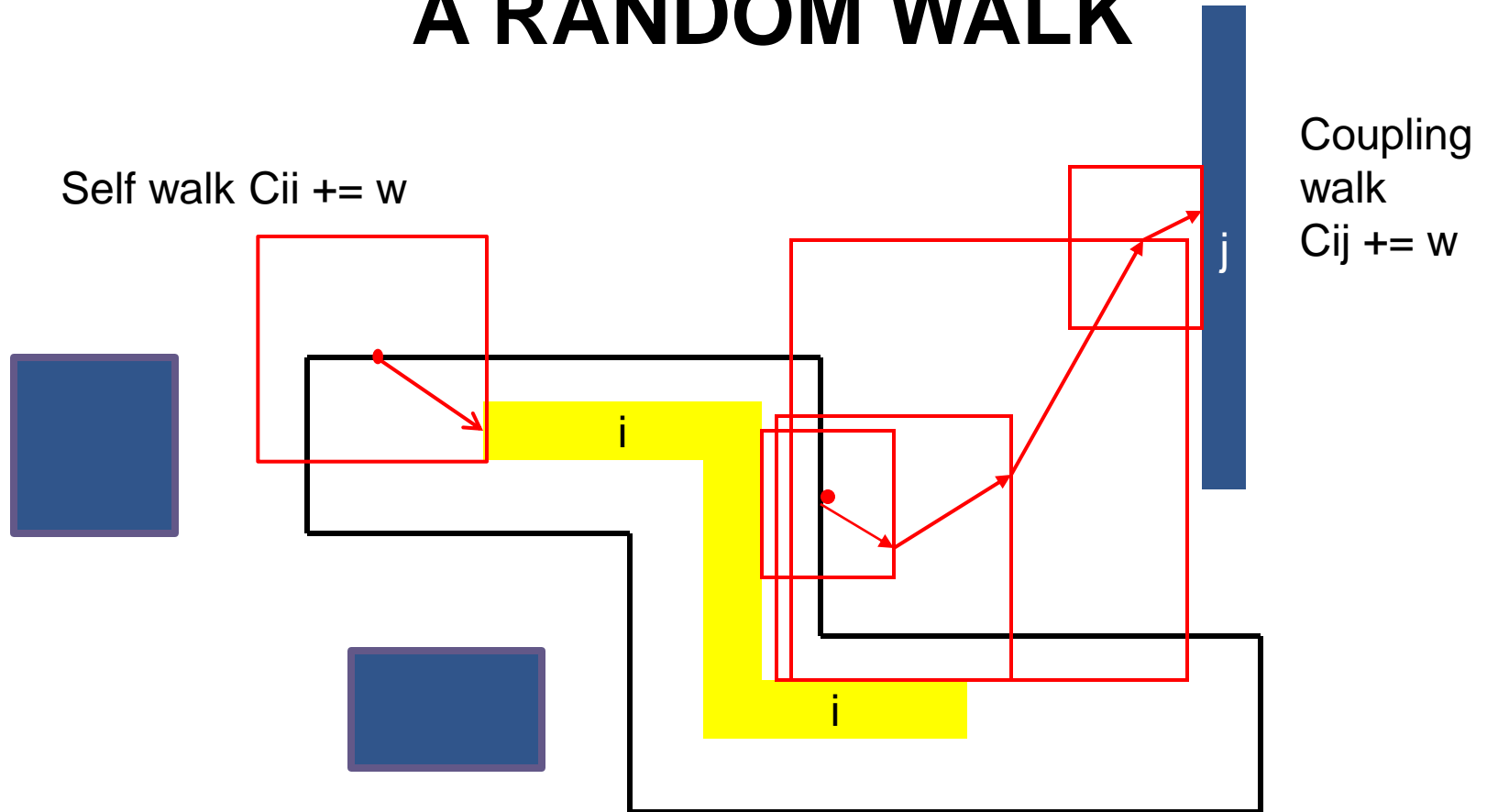


# GAUSSIAN SURFACE

- GS is a closed box that
  - completely contains the net
  - does not contain any other net
  - does not touch any net
- Need to find distance to closest box on each side then inflate the target net by  $\frac{1}{2}$  that amount



# A RANDOM WALK



$$w = A_{gs} * G_v * G_p * G_p * G_p \dots$$

With correct Greens functions ( $G_v$  &  $G_p$ ) can hop across dielectric boundaries!

# Random walk pseudo code

```
For each (net) {  
    Build GS(net)  
    do { // ~200,000 walks/net  
        pick point P on GS  
        find largest cube centered on P  
        hop to surface = P'  
        compute  $w = G_v(P, P')$   
        do { // ~ 15 hops/walk  
             $P = P'$   
            find largest cube centered on P  
            hop to surface = P'  
            compute  $w = w * G_p(P, P')$   
        } until(hits metal)  
         $C_{ij} += w$   
    } until (converged)  
}
```

Typical run:  
100 nets  
20,000,000 walks  
300,000,000 hops

# High usage tasks

- Find closest box to a point  $P$
- Jump to surface of cube
- Compute Greens functions  $G_v$  and  $G_p$
- Find closest box to each side of a box

# Change in product goals

- Original code
  - Stand alone product
  - Expected to grow into a large multi functional interconnect analysis product
  - High accuracy
  - Could handle conformal layers, but was not part of the original design
  - Client-server DP environment
  - Highly modular code with emphasis on module independence and re-use
  - 60,000 lines in core modules
- New code
  - Captive capacitance solver called from Star-RCXT
  - High speed and high accuracy
  - LSF DP environment and Multi-threading
  - Dedicated code with emphasis on speed and simplicity
  - 10,000 lines in core modules

# Speed Improvements

- ~3X from improved Greens Function methods
  - These are changes to the “math and physics”
- ~4X from changes to geometry engine and reduced volume of code
  - Call tree flattened
  - Search algorithm replaced
  - These are “computer science” changes
- ~ 1.5X from changes in how convergence is calculated and variance reduction
  - More “math and physics”

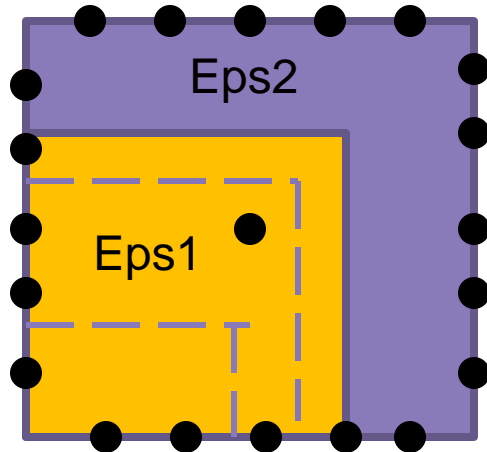
# Green's functions $G_v$ & $G_p$

- Mathematical function which relates potential at center of cube to potential at surface

$$V_c = \iiint G(u, v) V(u, v) du dv$$

- Analytic functions available for simple cases
- Numeric solution needed for all others
- Needs to be computed millions of times
- The primary contribution of the old code was a very efficient table based scheme to compute, store and apply greens functions.

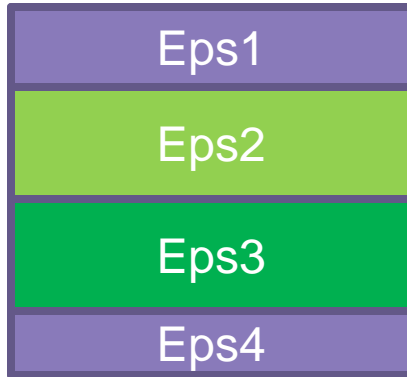
# Greens tables function continued



- A table is stored for each Eps configuration
- The table gives the weight value  $w$  and  $dx, dy, dz$  from center for each point on the surface
- Tables are generated once at start of the run
- Each table needs about 30Kb
- Finite number of surface points introduces discretization error



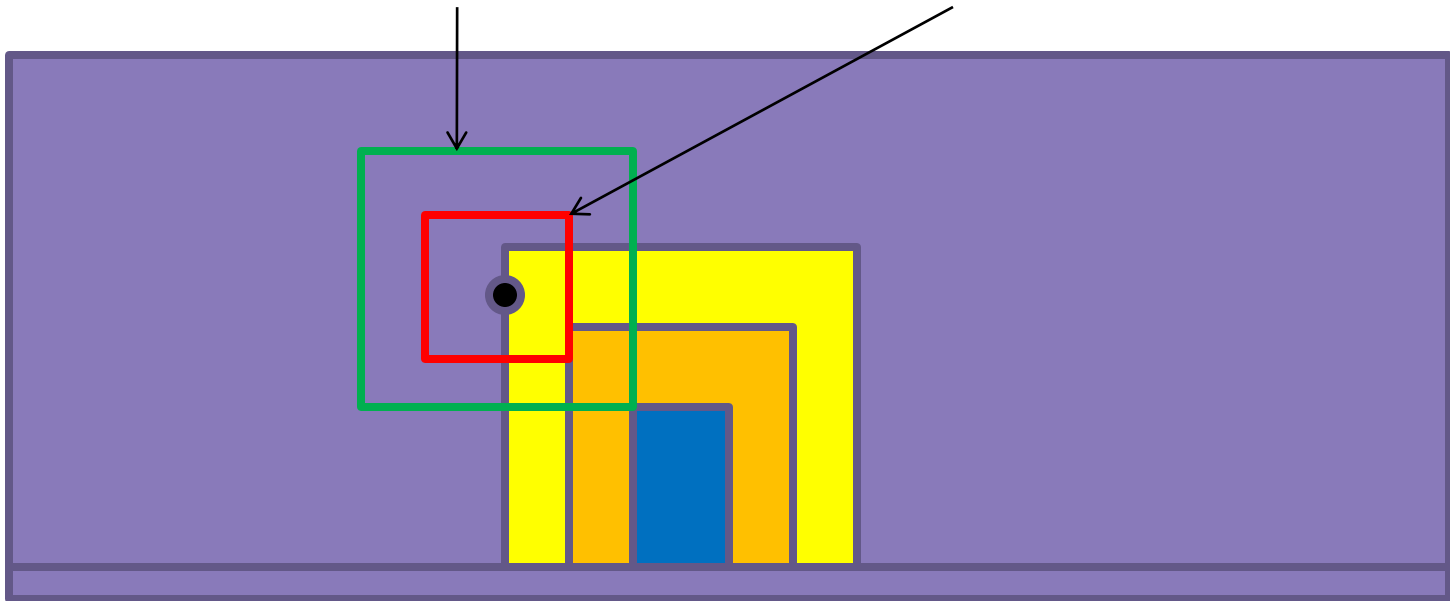
# Type of Greens function tables



- Layered type, matches planar dielectrics in chip (many Eps values allowed)
- Intrusion type matches conformal cubes (only 2 Eps values allowed)

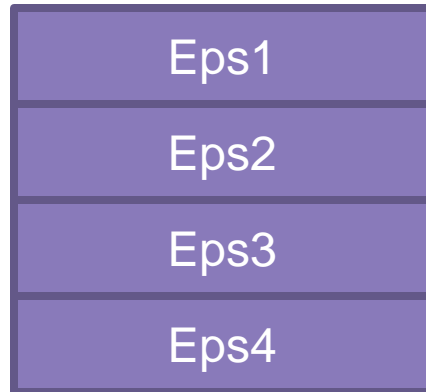
# GF Problem with old code

- Often, the hop cube size must be reduced to get a “match”
- One “bad point” can ruin converge
- runtime  $\propto (\max\_hop\_length / hop\_length)^2$  ← Key! discovery



Key clue: Runtimes varying by as much as 1000X when conformals changed

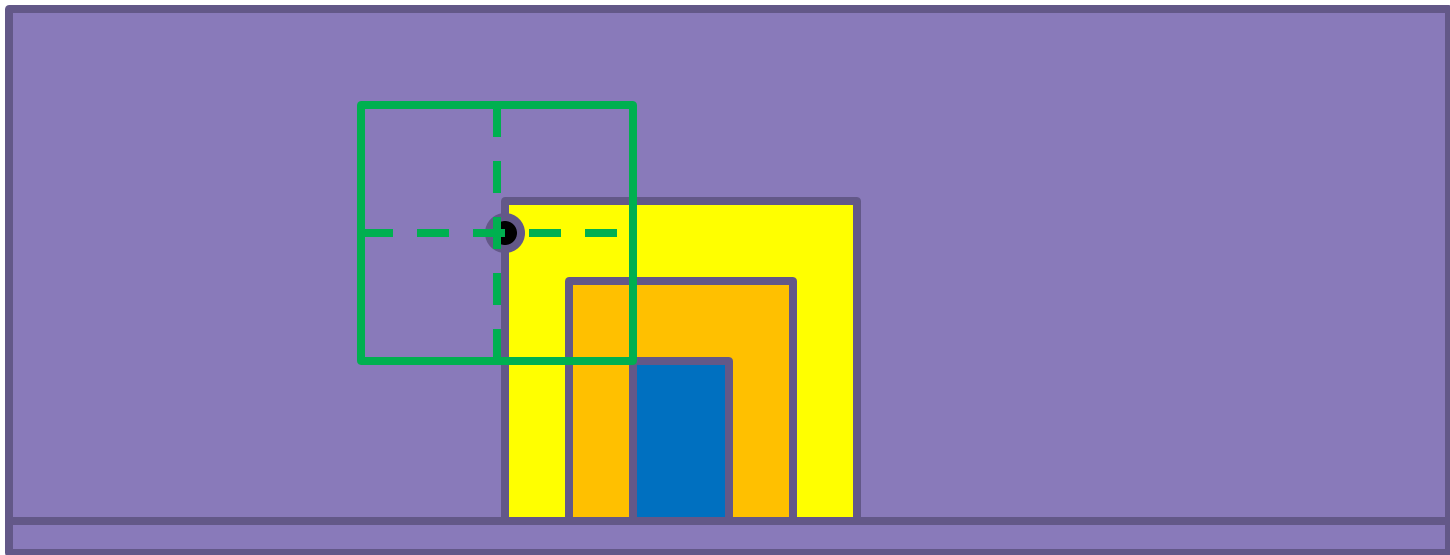
# New Greens function method



- Only two types of GF tables
- 4 volume cube for planar layers. 4 equal thickness layers
- 8 volume cube for conformals, divided into octants
- Shapes stay the same but Eps values in each section are “swept” producing different tables

# New Greens function method

- Largest possible cube is always used
- Volume weighted average Eps is computed for each of the 4 or 8 sections



Speed up of 0.5 to 1000 obtained with average  $\sim 3$

# Restructuring from the “CS” side

- Hindsight is “20:20”
  - Lessons learned from first attempt are very useful
- Unused features add complexity and get in the way
  - Multiple lex/yacc parsers
  - Interactive input layer
  - Resistance analysis
  - Density based thickness variation
  - All geometry info passed through API layer
- Key clue: Deep stack trace in core loops

# Clue #1, Deep stack trace in core loop

```
#22 in rq_query_box_intersect (44 bytes) 8 lines
^^^^^^ Geometry search ^^^^^^^^
#23 in frw_hop_compute_hop_length_with_trap (12 bytes) 33
^^^^^ Hop Loop ^^^^^^^^
#24 in frw_hop_from_gaussian_surface (48 bytes) 146 lines
#25 in frw_walk_set_starting_point (56 bytes) 41 lines
#26 in frw_walkm_perform_walk_from_net (40 bytes) 27 lines
#27 in frw_walkm_perform_random_walk (56 bytes) 38 lines
^^^^^^ Core Walk Loop ^^^^^^^^
#28 in scx_perform_chunks_of_random_walks (40 bytes)
#29 in scx_perform_net_by_net_extraction (32 bytes)
^^^^^^^^ Loop over Nets ^^^^^^^^^
#30 in scx_extract (32 bytes)
#31 in sccli_excmd_extract (cmd, argc, argv)
#32 in cmd_execute ()
#33 in cmd_source_aux ()
#34 in cmd_source ()
#35 in cmd_execute ()
#36 in statcap_run (argc argv)
#37 in main (argc, argv)
```

```
#0 geo_box3_intersect_geo_box3 (12 bytes) 22 lines
#1 leaf_box_intersect_array_x (20 bytes) 56 lines
#2 rq_oct2_node_query_box_intersect (20 bytes) 157 lines
#3 rq_oct2_node_query_box_intersect (20 bytes)
#4 rq_oct2_node_query_box_intersect (20 bytes)
#5 rq_oct2_node_query_box_intersect (20 bytes)
#6 rq_oct2_node_query_box_intersect (20 bytes)
#7 rq_oct2_node_query_box_intersect (20 bytes)
#8 rq_oct2_node_query_box_intersect (20 bytes)
#9 rq_oct2_node_query_box_intersect (20 bytes)
#10 rq_oct2_node_query_box_intersect (20 bytes)
#11 rq_oct2_node_query_box_intersect (20 bytes)
#12 rq_oct2_node_query_box_intersect (20 bytes)
#13 rq_oct2_node_query_box_intersect (20 bytes)
#14 rq_oct2_node_query_box_intersect (20 bytes)
#15 rq_oct2_node_query_box_intersect (20 bytes)
#16 rq_oct2_node_query_box_intersect (20 bytes)
#17 rq_oct2_node_query_box_intersect (20 bytes)
#18 rq_oct2_node_query_box_intersect (20 bytes)
#19 rq_oct2_node_query_box_intersect (20 bytes) 157 lines
#20 rq_oct2_query_box_intersect_finger (8 bytes) 57 lines
#21 rq_oct2_query_box_intersect (48 bytes) 40 lines
```

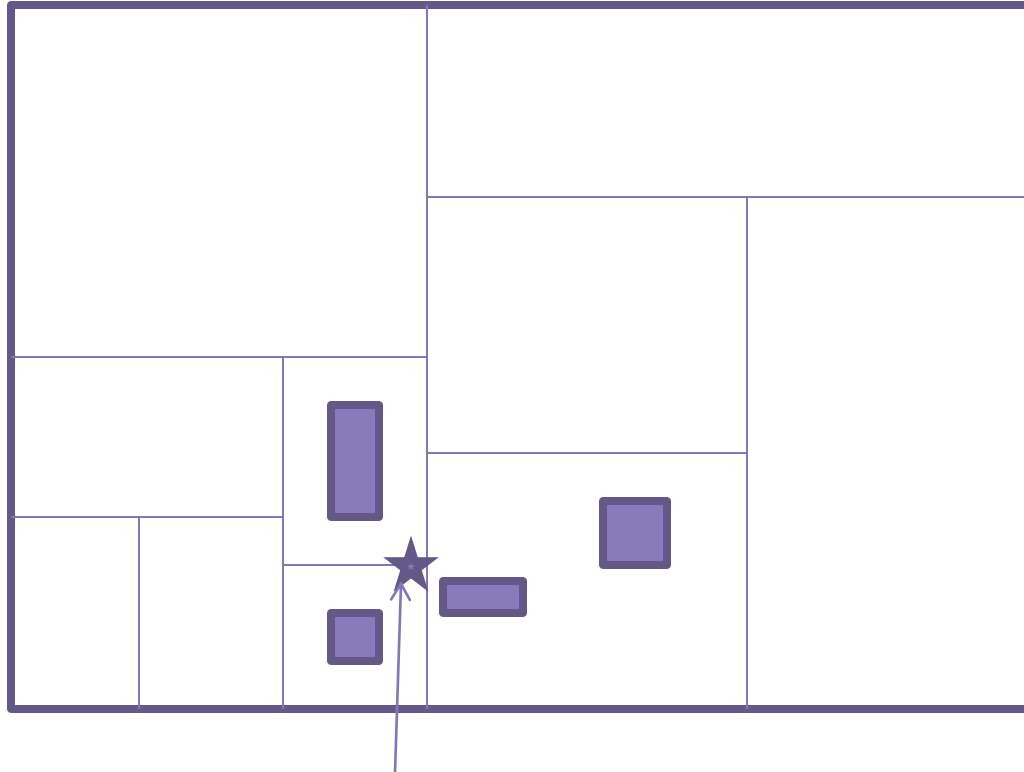
A lot of activity is going on!

Recursive search algorithm is elegant, but function calls are not free!

Number of function arguments grew as functionality was added

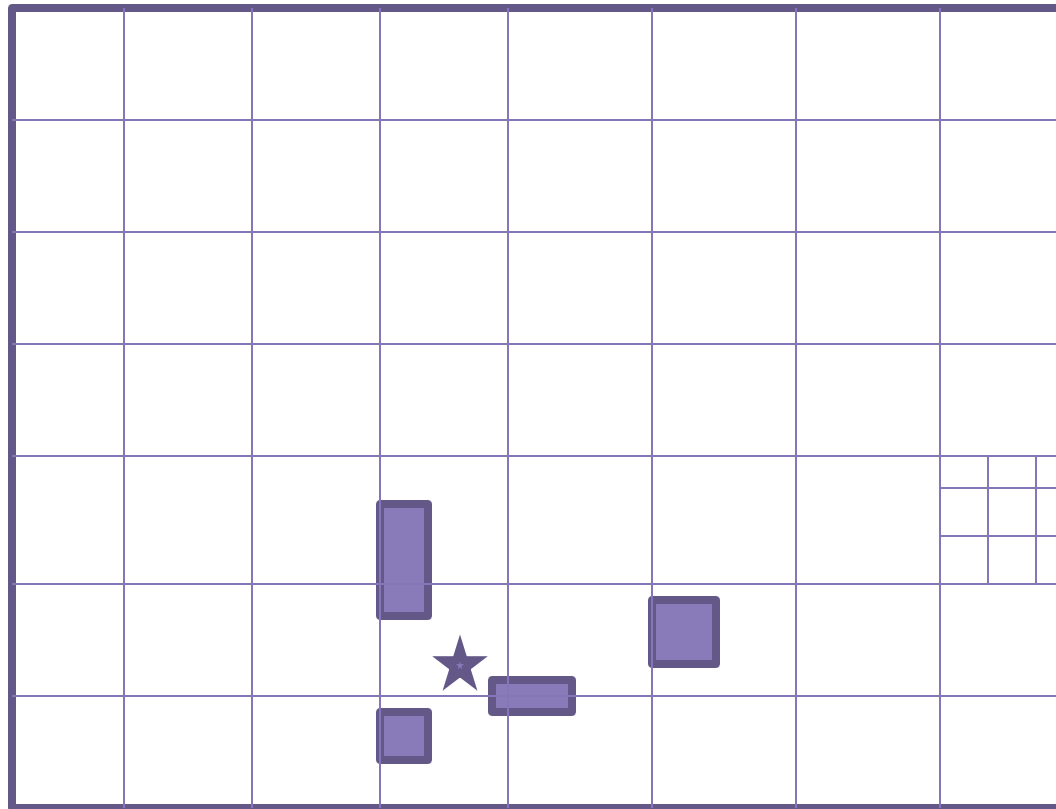
# K-D tree gives efficient storage but is complex

Log(N) effort to locate point with many conditionals



To check this point need to check 3 cells and go up and down tree

# New code uses simple x-y array



Target 8 boxes/cell  
Some boxes hit many cells

For very dense cells a  
2<sup>nd</sup> level can be added  
(happens rarely)

**~4X speed up  
obtained**

Cell index =  $N_y * (x - x_{min}) / dx + (y - y_{min}) / dy$

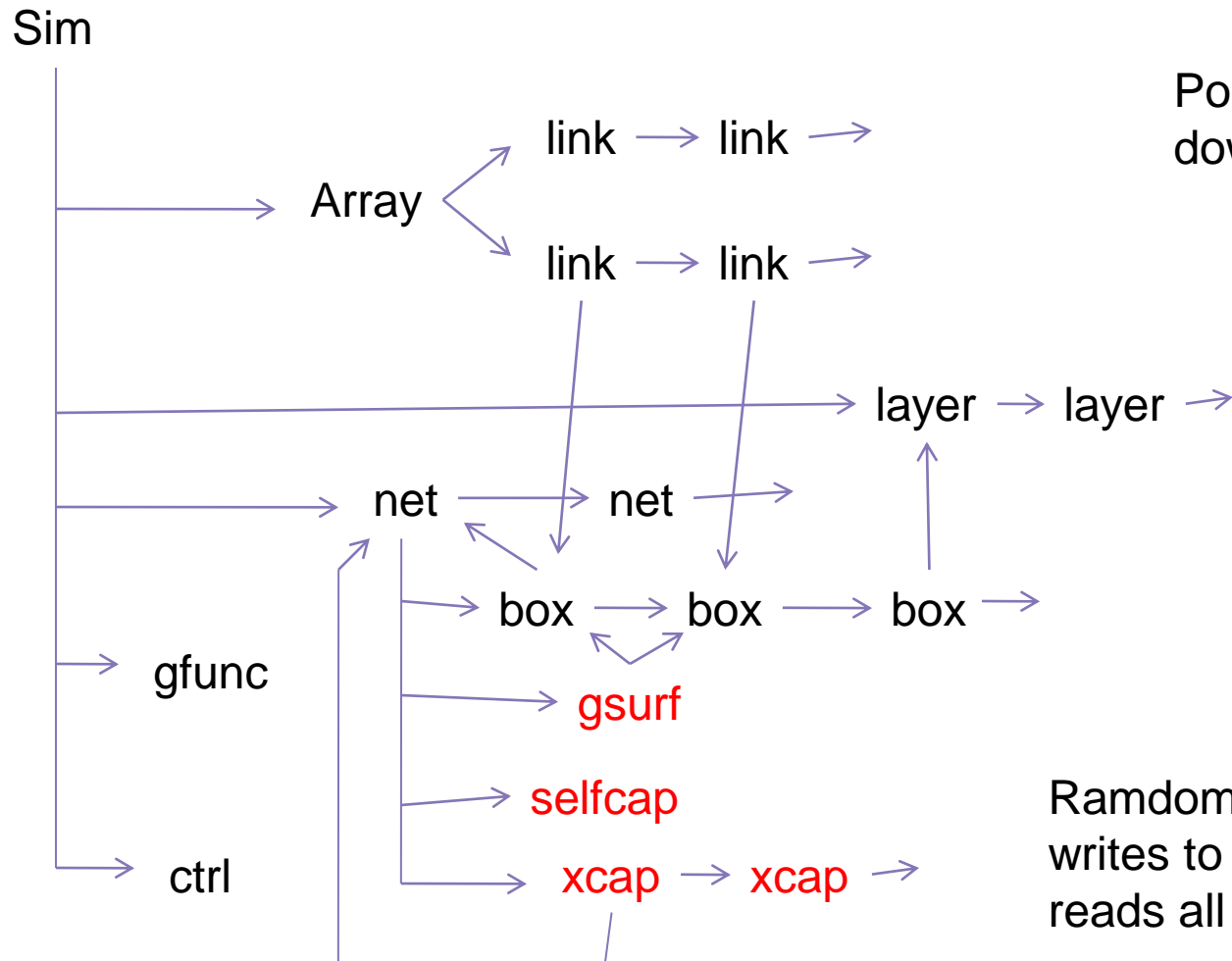
Locate point in constant time with no conditionals

Neighbors are easy to locate too.

Memory for array is typically < 10% of memory for boxes



# Simplified data structures



Pointer to Sim is passed down through call tree

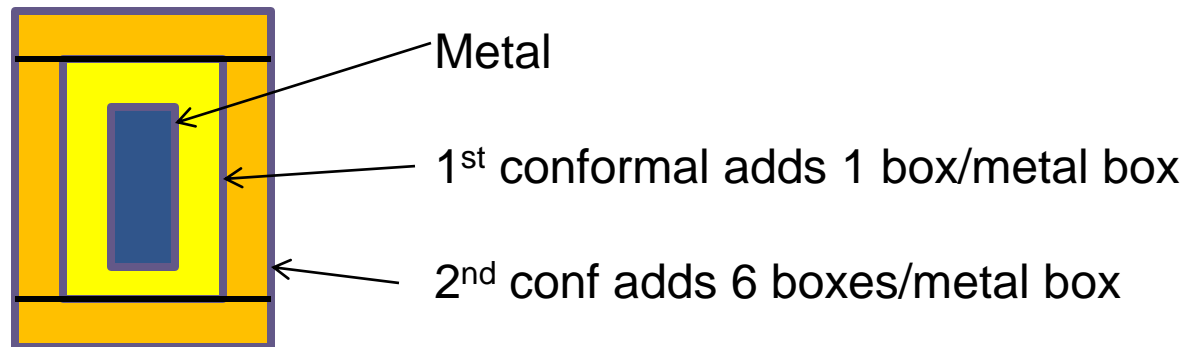
Random walk process writes to **red** items reads all others

# Stack trace for new program

```
#0 box::dist (16 bytes)          20 lines
#1 p_tree::search_c (28 bytes)   31 lines
#2 p_tree::search_ct (24 bytes)  18 lines
#3 p_tree::search_tree (36 bytes) 98 lines
^^^ Geometry search ^^^
#4 solve (16 bytes)              107 lines  <- Walk loop & hop loop
#5 run_gcap (argc, argv)         692 lines  <- Setup & Loop over nets
#6 main (argc, argv)             93 lines
```

# Memory savings

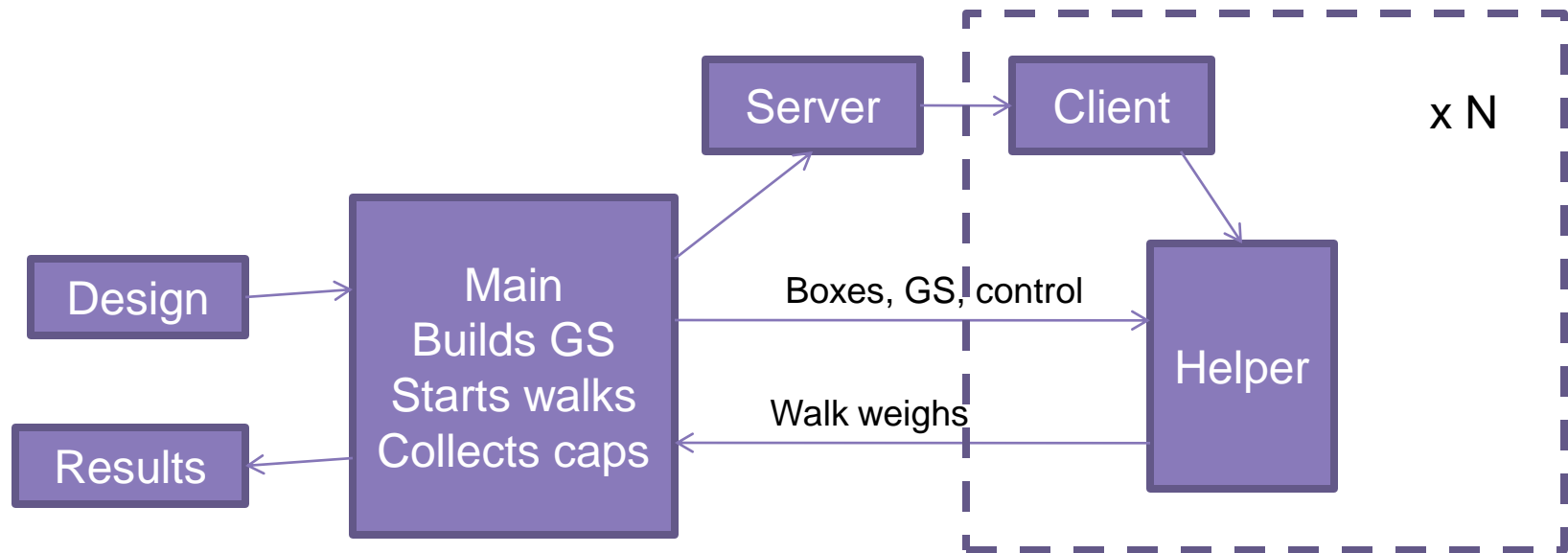
- Conformal layers are generated by inflating underlying metal layer by fixed factors `inf_xy`, `inf_zp`, `inf_zn`
- Old code creates and stores these boxes
- 6 boxes may be needed for 2<sup>nd</sup> level conformal
- New code creates these boxes “on the fly” during walk process.



~3X memory reduction obtained

# DP reworked

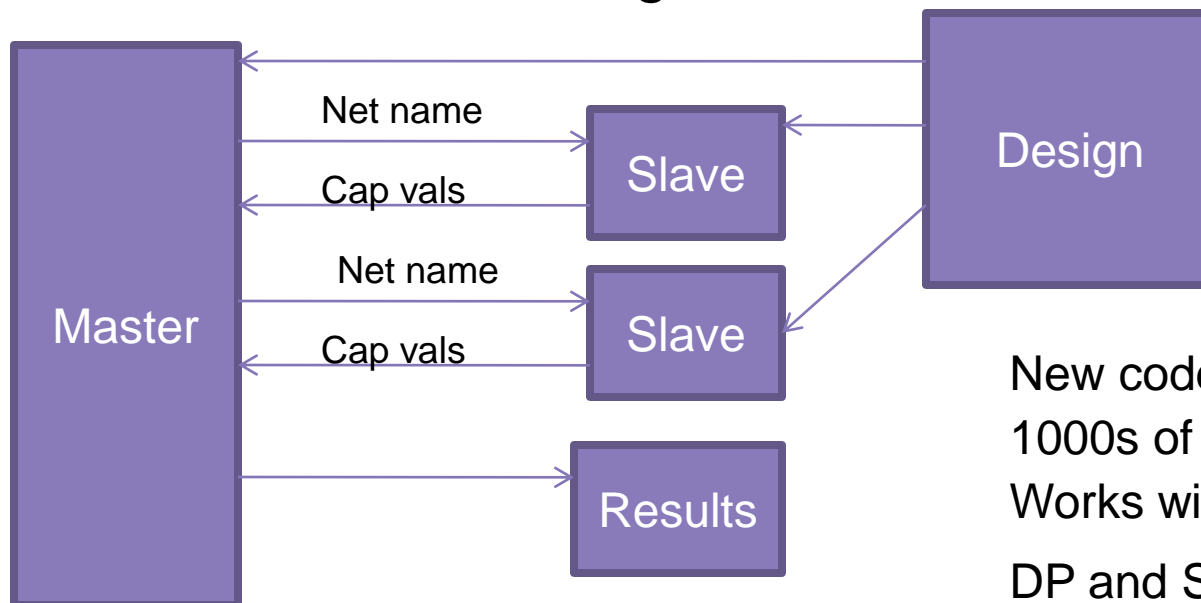
- The random walk method is easy to parallelize.
  - Nets or walks can be done in parallel
- Old architecture
  - Communication through pipes
  - 4 separate executables
  - Very complicated data passing, easy to “miss things” when adding features



Very difficult to debug & maintain, DP and SP give different results

# New DP method

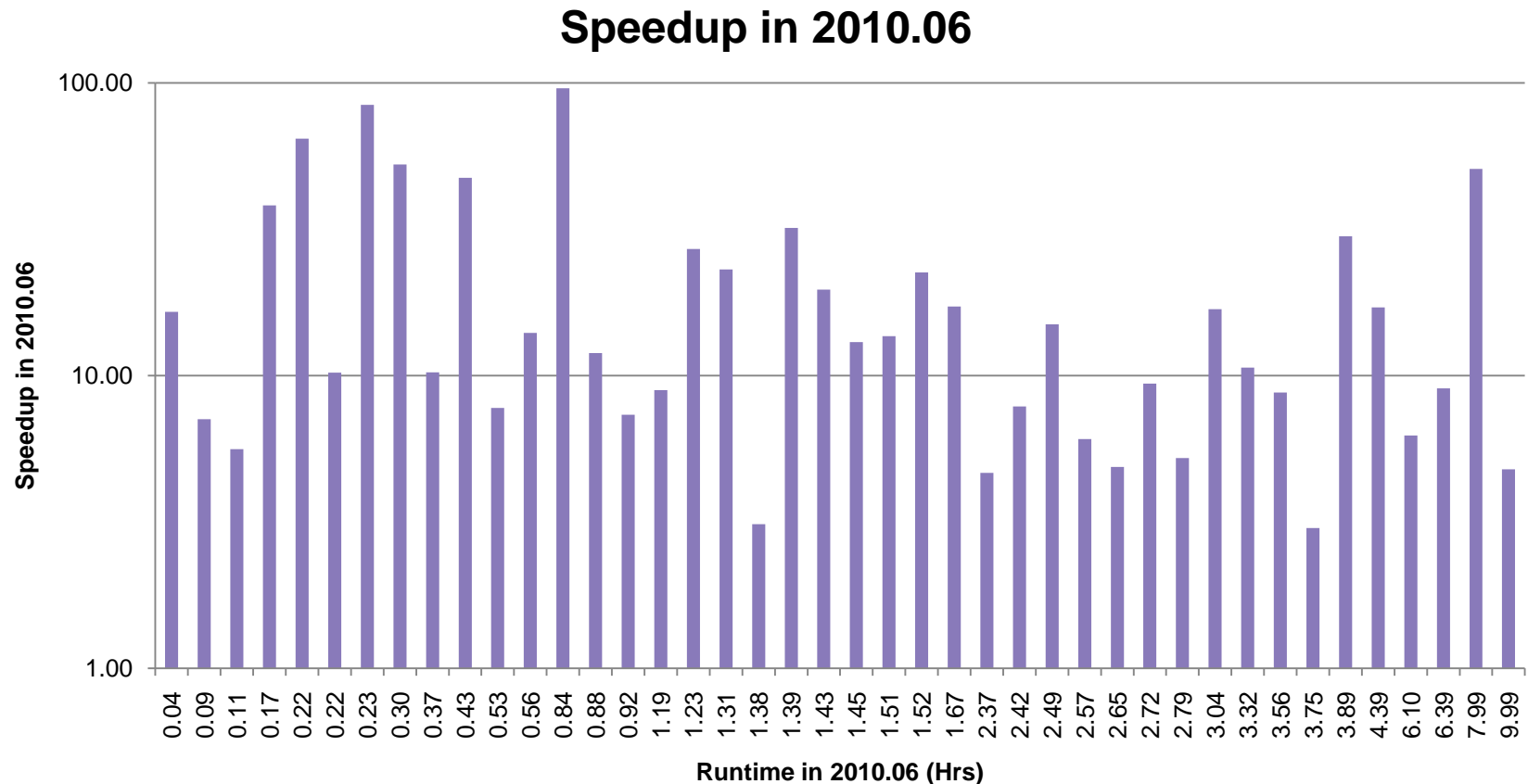
- Single executable (master/slave selected by command line option)
- Master spawns all slaves (LSF, Gridware or machine list)
- Each slave reads design, analyzes nets as instructed, passes back final caps
- Communication through files



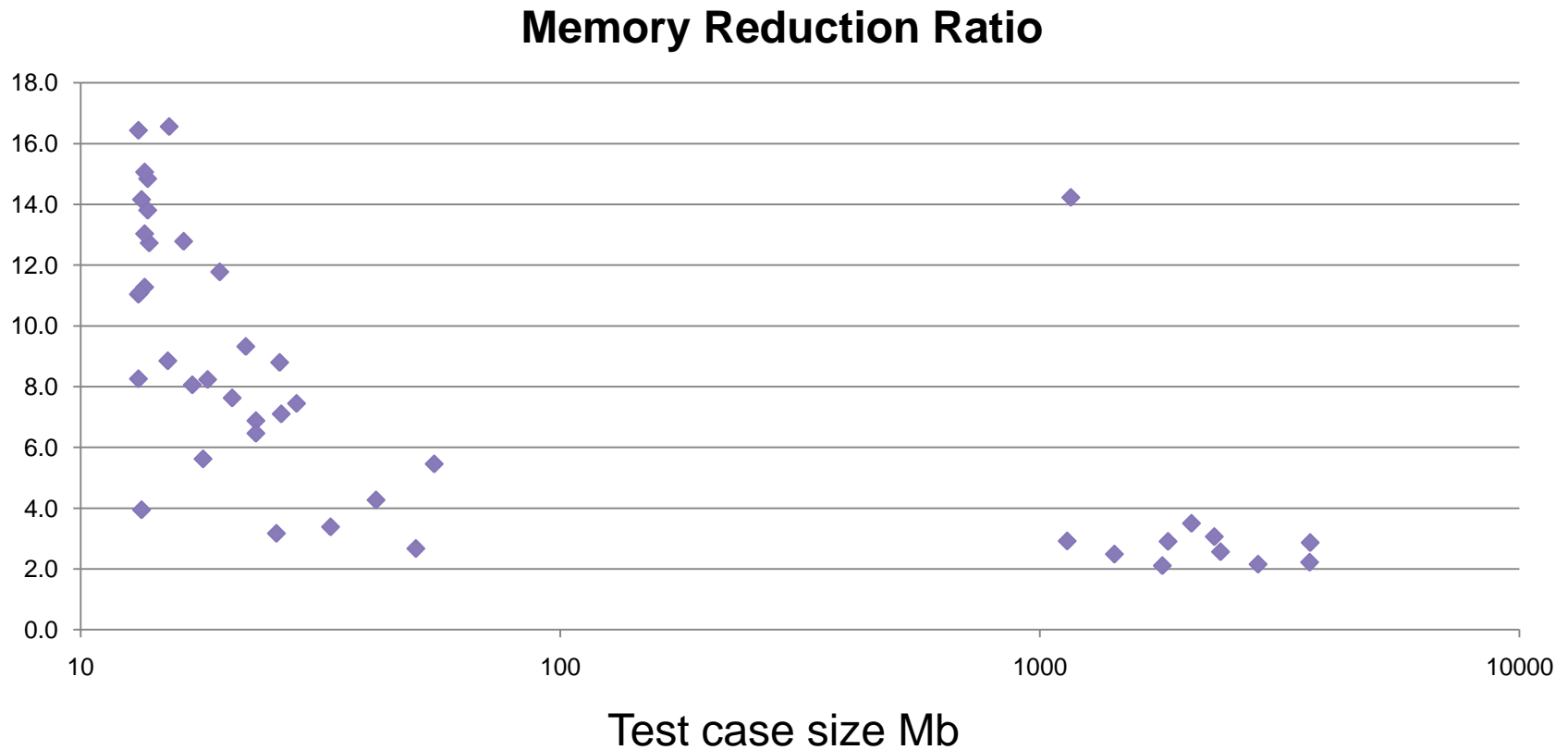
New code slightly more efficient  
1000s of lines of code removed  
Works with multithreading too  
DP and SP give identical results

# Performance results:

- Geometric mean speedup ~15 on 41 diverse test cases



# Results on memory



# Accuracy, 2009.06 release –vs- RC2

Tech	nets	Mean	std	max
ST32	2k	-0.55	2.8	15
STTOP32	2k	0.23	1.03	5.6
STTOP45	2k	-0.17	1.39	5.4
STTOP65	2k	0.23	1.03	5.6
TSMC32	1770	0.05	0.96	5.7
TSMC45	2k	0.31	1.20	6.4
TSMC65	2k	0.06	1.10	8.7
TSMC90	2k	0.12	0.76	5.15

Ave(|mean|+STD) = 1.502%    Av(Max) = 7.2%



# Accuracy, 2010.06 –vs- RC2

Case	Nets	mean	std	max
ST32	2k	0.34	1.27	4.7
STTOP32	2k	-0.19	1.35	6.8
STTOP45	2k	-0.53	1.82	6.7
STTOP65	2k	-0.21	1.36	6.8
TSMC32	2k	0.83	1.66	6.2
TSMC45	2k	0.12	1.24	4
TSMC65	2k	0.52	1.09	4.2
TSMC90	2k	0.29	0.78	2.1

$\text{Av}(|\text{Mean}| + \text{STD}) = 1.704\%$      $\text{Av}(\text{Max}) = 5.18\%$

With no confromal layers  $\text{Av}(|\text{Mean}| + \text{STD}) = 0.95\%$      $\text{Av}(\text{Max}) = 2.8\%$

With Uniform Eps=1     $\text{Av}(|\text{Mean}| + \text{STD}) = 0.64\%$      $\text{Av}(\text{Max}) = 1.5\%$

# How to tell if it is possible to do better

- The competition does it....
  - This is not a good way!
- Find one case that runs “good” and one that runs “bad” and figure out why
  - If you can measure it, you can improve it.
  - “good” may not be “best”
- Try going to extremes
  - Performance, memory and accuracy are usually conflicting goals, try weighting one very heavily and see “how bad things get for the others”

# Profiling?

- Profiling is useful to find bottlenecks in a program
- It is less useful if
  - The problem is numeric or algorithmic
  - The “fat” is distributed throughout the code

# Conclusions

- Getting the best performance requires a high level view of the product
  - Numerics & algorithms
  - Programming techniques
- Some “structured programming” techniques can hurt performance
  - Beware of code and data fragmentation
  - MachTA ran 20% faster after in-lining a 5 line function in the core code.
- Dedicated code gives best performance
  - Medici (2D Fortran) is 4X faster than Taurus (2D/3D C++) on 2D problems.