Solving Hard Instances of Floorplacement

Aaron Ng, Igor Markov *University of Michigan*
Rajat Aggarwal *Xilinx, Inc.*
Venky Ramachandran *Calypto Design Systems, Inc.*

Outline

- Motivation and previous work
  - Design trends and placement tools: RTL placement
  - Floorplacement techniques
- Difficult floorplacement instances
  - Empirical analysis of existing techniques
- Scaling floorplacement up with SCAMPI
  - Techniques to improve floorplacement
  - Empirical results
  - Advantages and drawbacks
- Conclusions

Motivation & previous work

Design trends & placement tools

- Traditional placement is *bit-level*
  - Relatively late in the design flow
  - Relatively slow
- Layout of final implementations
  - IP modules, memory, SoCs

  → hard macro modules
- System-level design & high-level synthesis
  - Fast performance estimations, prototyping
  - Build custom RTL library – pre-characterized area, timing, power

  → soft macro modules

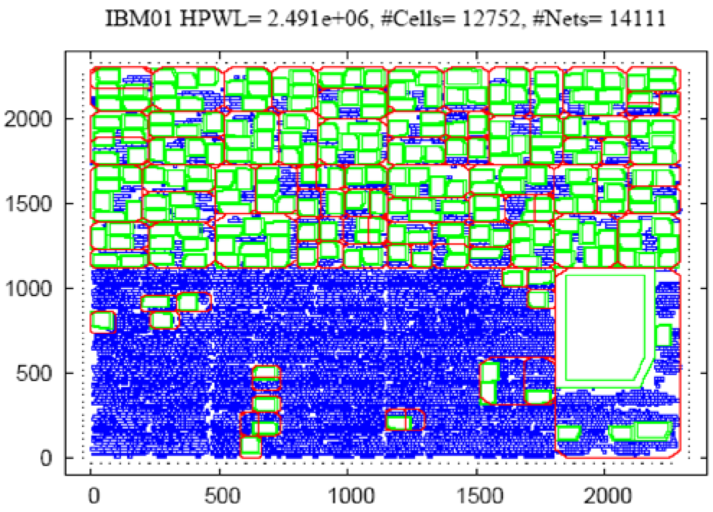
Support for larger scale & greater complexity

- Moving away from bit-level design → more macros
- Floorplanning
  - Std cell placement & floorplanning have similar objectives
    - non-overlapping module locations
    - optimization of interconnect, but
  - More expensive algorithms required for floorplanning
    - std cells fit in rows and are relatively similar in size
    - macro modules can span rows & vary greatly in size

→ Floorplanning algorithms do not scale well

Unification of floorplanning and placement

- Floorplacement [Adya, ICCAD04]
  - Simultaneous placement + floorplanning
  - Various combinatorial + analytic techniques (PATOMA, Capo, APlace)

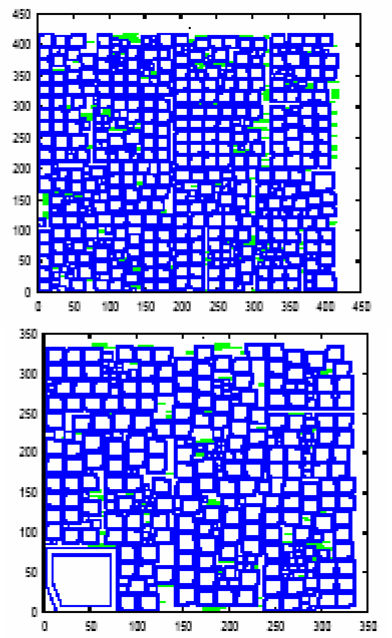


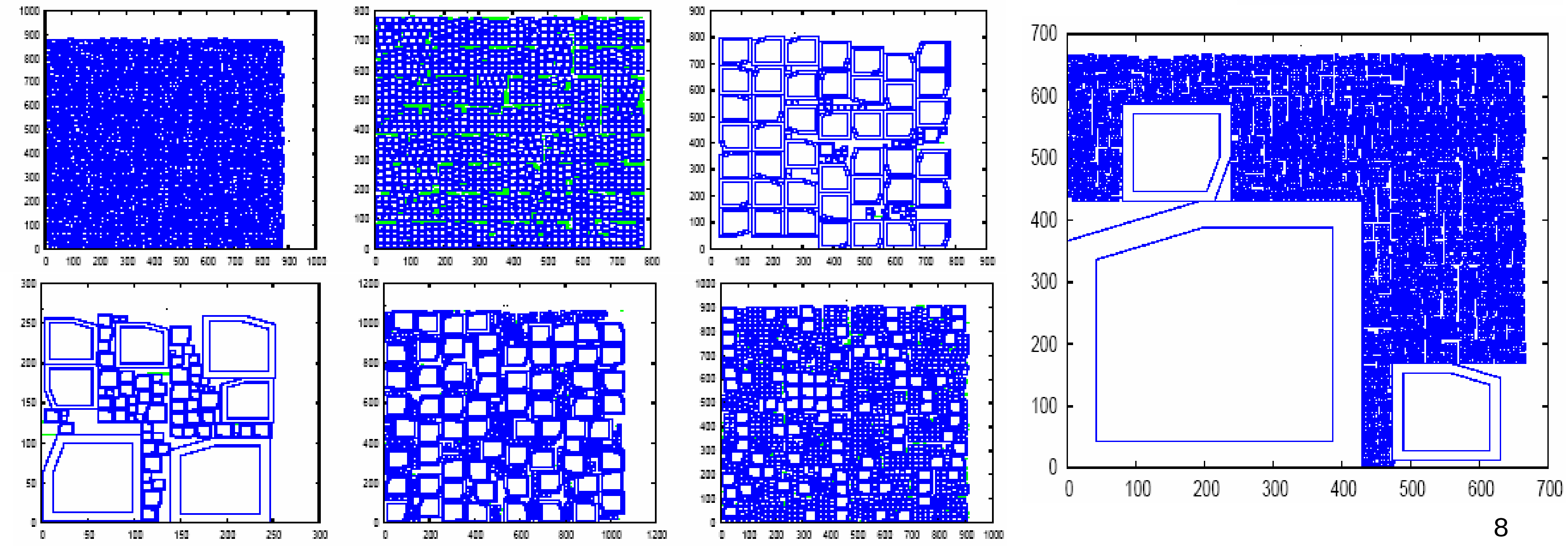
- Shortcomings of unified frameworks
  - Placement + floorplanning integration is not seamless
  - Tradeoff between scalability & accuracy (e.g., sacrificed strength of floorplanning algorithms)
  - **To illustrate these effects, we introduce a suite of hard floorplacement benchmarks**

Difficult floorplacement instances

Difficult instances

- 81 to 8827 RTL modules
- Hard & soft modules, some std cells
- $\text{Area}_{\text{largest}}$ up to 50% of total cell area
- $\text{Area}_{\text{largest}}$ / $\text{Area}_{\text{smallest}}$ 650 to 185330
- http://vlsicad.eecs.umich.edu/BK/ISPD06bench



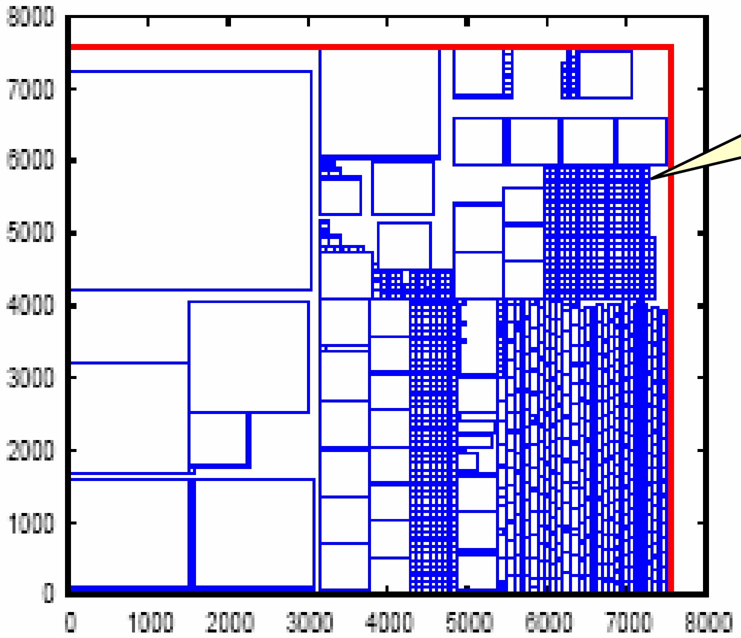


Empirical analysis of existing techniques

Partitioning & fast block-packing

- PATOMA [J.Cong et. al, ASPDAC 2005]
- Hierarchical min-cut partitioning
    - Bears the burden of minimizing interconnect
- Fast block-packing on resulting partitions
    - Check area feasibility
    - Weak wirelength optimization
- Contingency plan
    - Best legal packing is saved at every level
    - If partitioning cannot continue, best legal packing is used

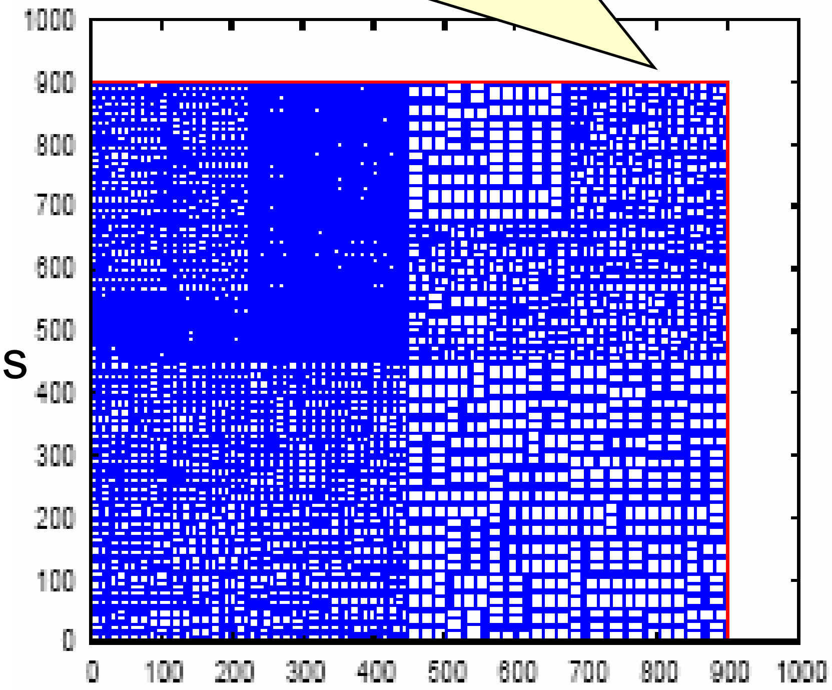
Partitioning & fast block-packing (cont'd)



Fast block-packing solutions used too early

Fast lookahead block packers check area feasibility of floorplanning instances
- produce false negatives
- bail out too early

Bad wirelength in some cases
(9.7x worse in this case)



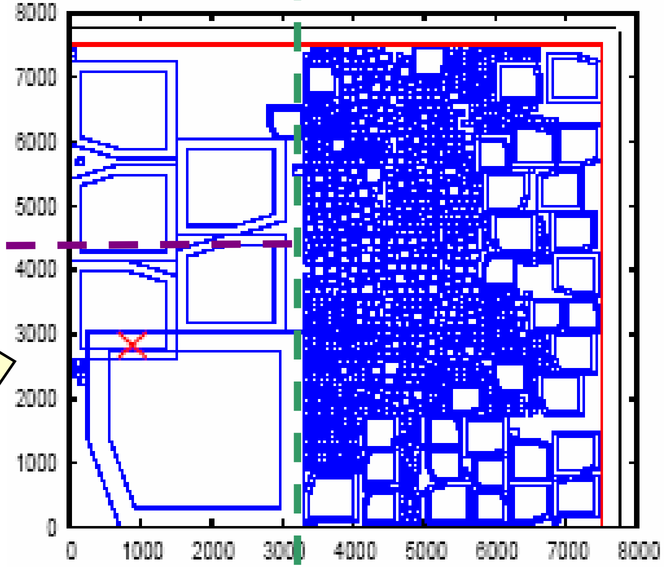
Partitioning & strong block-packing

- Capo (with Parquet) [J. A. Roy et. al, TCAD 2006]
- Top-down min-cut placement framework
  - Dynamically invoke floorplanner using heuristics (e.g., when a block is too large to fit in child partitions)
  - Can undo partitioning decisions and perform FP instead
- Floorplanning by simulated annealing
  - Floorplan representations capture large solution space (e.g., SeqPair, B*-tree)
  - Multi-objective optimization (area & wirelength)
  - Hard & soft blocks with any aspect ratios
  - Limited effective operating range (up to ~100 modules)

Partitioning & strong block-packing (cont'd)

1st cut

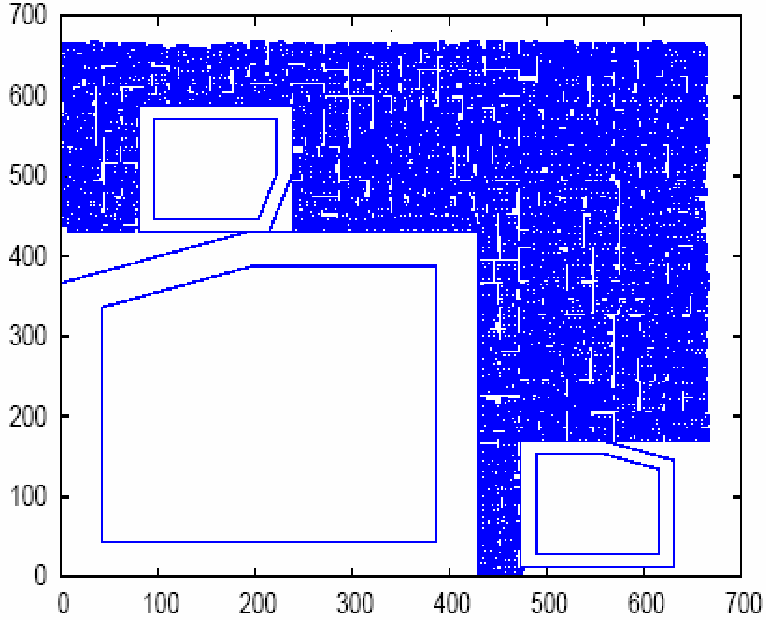
2nd cut



Capo invokes floorplanning on bottom-left bin, but discovers that it cannot find a legal solution

The bottom-left bin is merged with the top-left bin, and floorplanning is retried. Capo still fails to floorplan and cannot proceed because only one level of backtracking is allowed. This is an example of area misallocation discovered too late.

At the very top level, the largest macro cannot fit in either subpartition. Capo invokes the floorplanner on 8827 (too many) modules

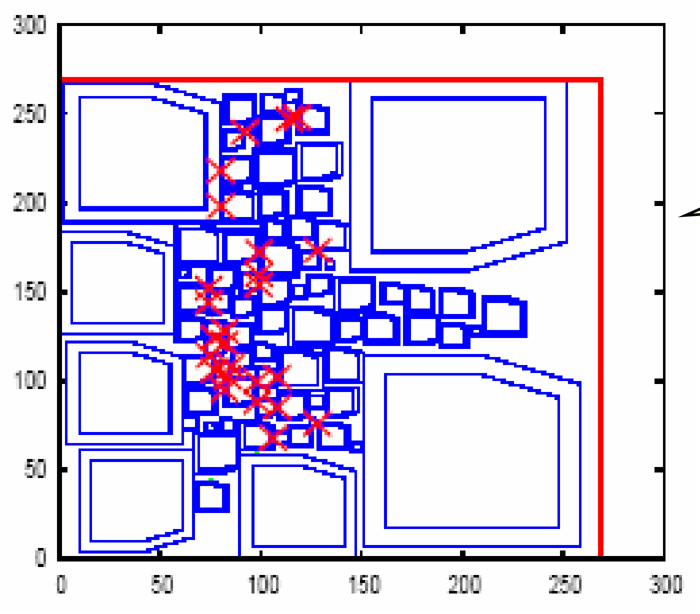


Analytical placement, cell spreading

- APlace [A. B. Kahng et. al, ICCAD 2005]
- Non-linear optimization
  - DensityWeight*DensityPenalty + WLweight*TotalWL
  - DensityPenalty = $\sum_g (\sum_c \text{Potential}(c,g) - \text{ExpPotential}(g))^2$

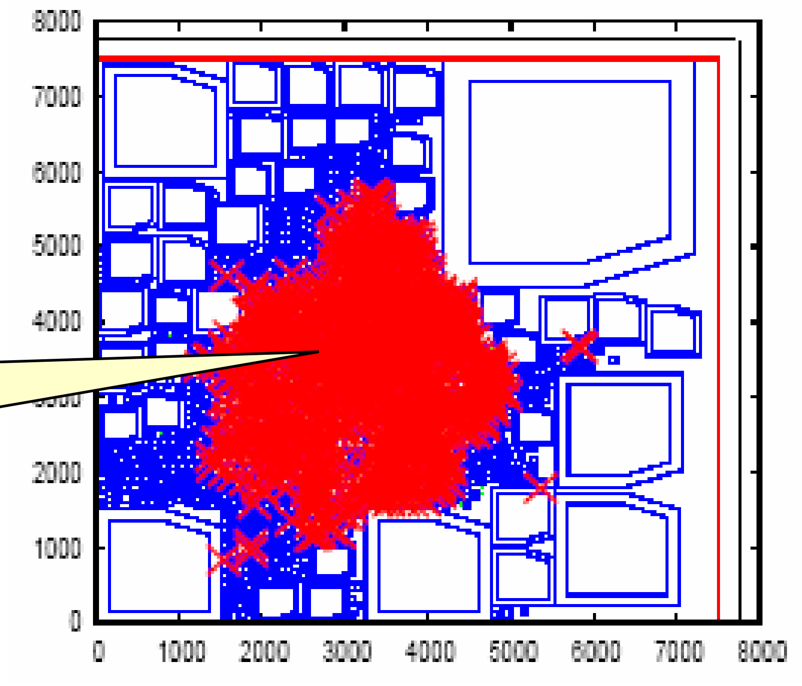
    (Potential is a bell-shaped function of:
    module dims, a radius of influence & module's distance from grid cells)
  - $WL(t) = \alpha(\ln(\sum e^{x_i/\alpha}) + \ln(\sum e^{-x_i/\alpha})) + \alpha(\ln(\sum e^{y_i/\alpha}) + \ln(\sum e^{-y_i/\alpha}))$ (for a net *t*)
- Simultaneous handling of macros and std cells
  - Clustering for scalability and better solution quality
- Legalization usually required after cell-spreading

Analytical plcmnt, cell spreading (cont’d)



Cell-spreading != legalization

When multiple modules are clustered, the shape and area of clusters is hard to predict. This results in overlaps.



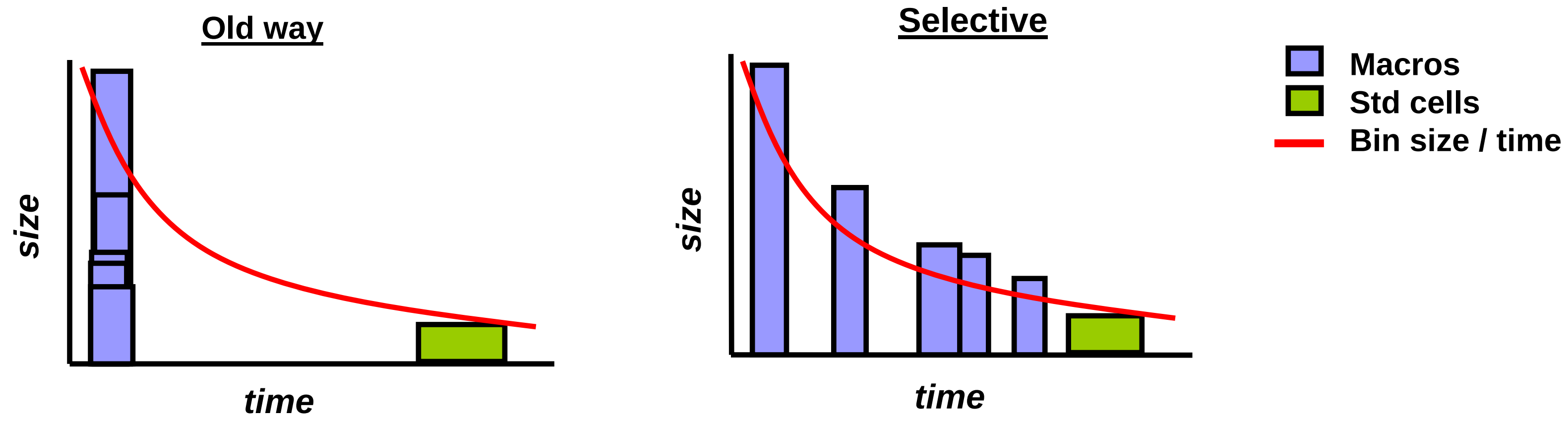
Scaling floorplacement up

Scaling floorplacement up

- Hierarchical framework: coarse view → fine view
  - Approximations more tolerable at the coarse level
  - Accurate/detailed algorithms required at the fine level
  - Our work bridges the gap between coarse & detail levels
- SCAMPI
  - Scalable Advanced Macro Placement Improvements
  - Selective macro placement and clustering
  - Obstacle handling
  - Look-ahead floorplanning
  - Whitespace allocation by block densities

Selective macro placement & clustering

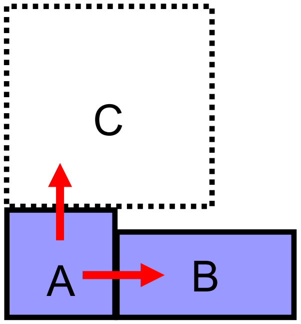
- Place *large* modules early
  - ☐ A module is placed & fixed when it becomes *large* relative to its bin (partition)
  - ☐ Cluster smaller modules & std cells into soft blocks



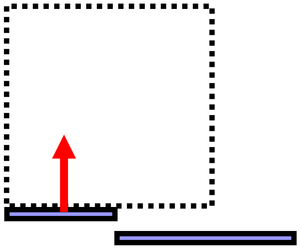
- Specific locations are determined at the right level of spatial hierarchy

Obstacle handling

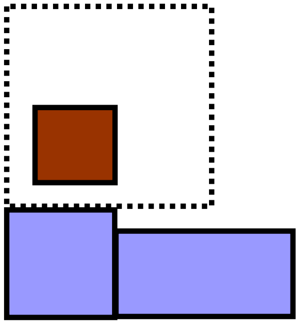
- Necessity
  - Macros placed early become obstacles
  - Obstacles can also appear in input
- Our approach
  - Modify well-known B*-tree evaluation procedure



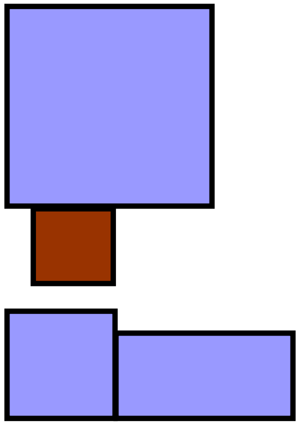
DFS B*-tree to evaluate packing from an ordering



Contour data structure for fast evaluation



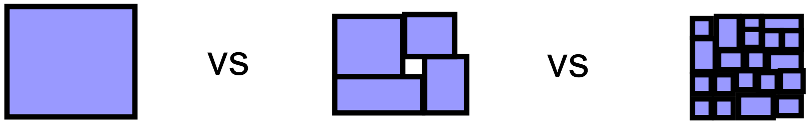
Block C wants to go above A, but obstacle present



Shift C to closest position past obstacle

Other improvements

- Ad hoc look-ahead floorplanning
  - Quick area feasibility check for a bin
  - Fast block-packing of *large* blocks
  - Aggressive clustering to reduce the problem size
- Whitespace allocation by block *densities*
  - Sum of area underestimates area of packed blocks (assumes zero deadspace)
  - Estimate deadspace by using sum of module perimeters (e.g., surface area)



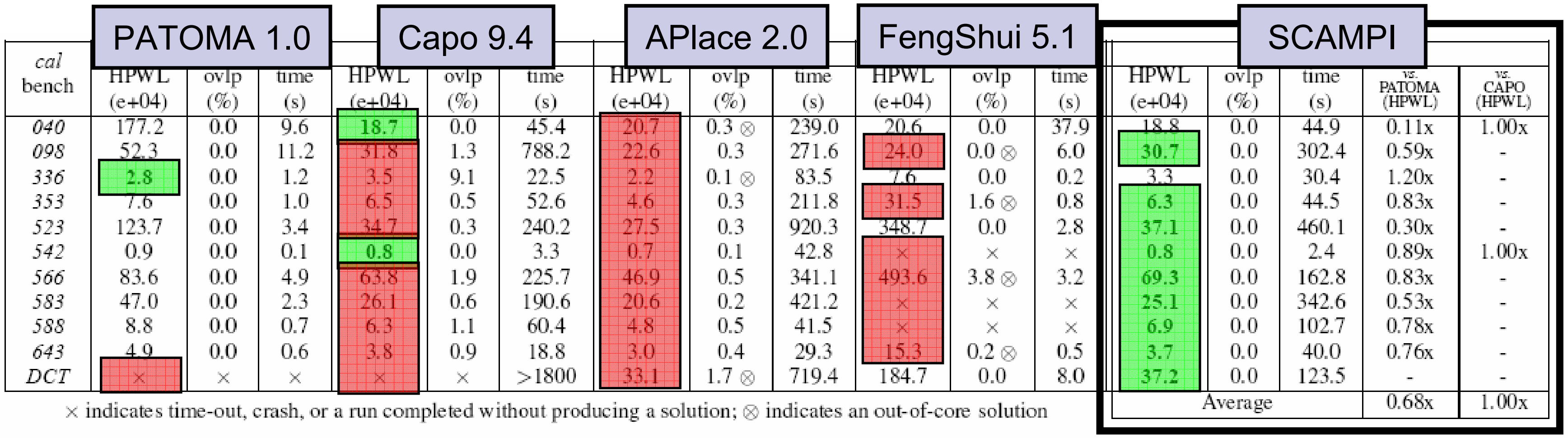
no deadspace **some deadspace** **no deadspace**

  - Compare bins and adjust cutlines after partitioning

Empirical results

🟩 Best legal solutions

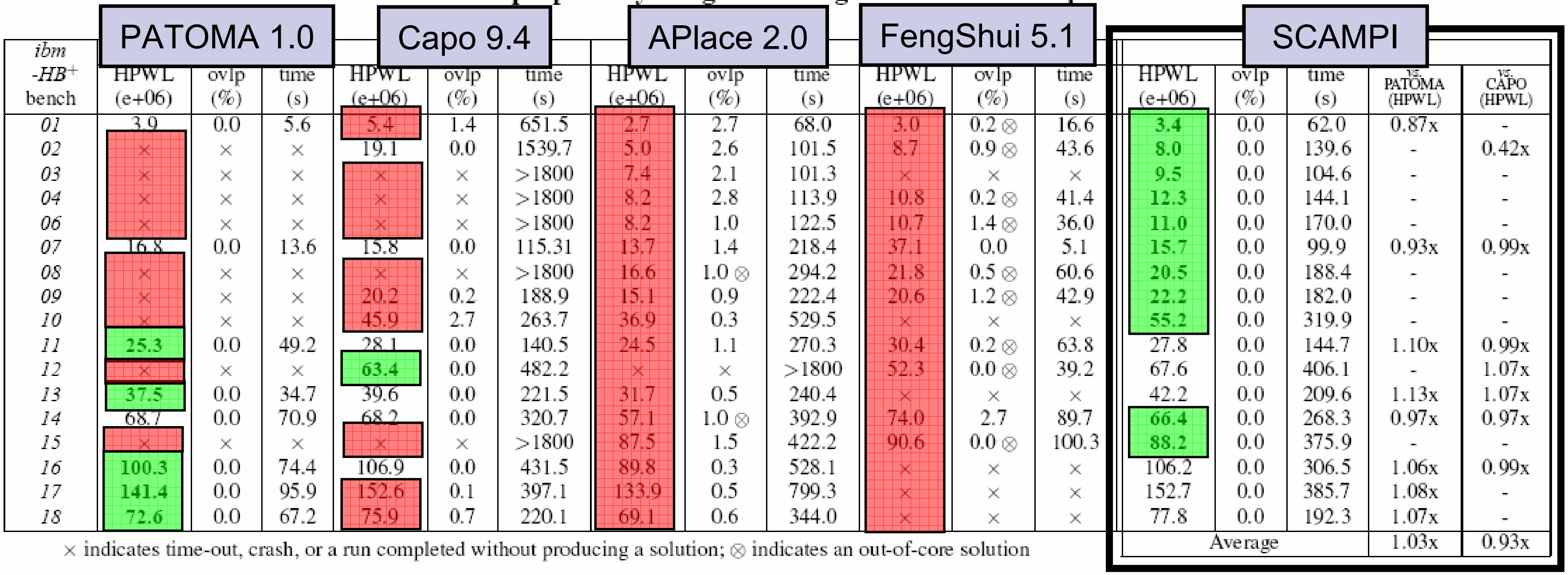
🟥 Illegal or no solution



cal bench	PATOMA 1.0			Capo 9.4			APlace 2.0			FengShui 5.1			SCAMPI				
	HPWL (e+04)	ovlp (%)	time (s)	HPWL (e+04)	ovlp (%)	time (s)	HPWL (e+04)	ovlp (%)	time (s)	HPWL (e+04)	ovlp (%)	time (s)	HPWL (e+04)	ovlp (%)	time (s)	vs. PATOMA (HPWL)	vs. CAPO (HPWL)
040	177.2	0.0	9.6	🟩 **18.7**	0.0	45.4	🟥 20.7	0.3 ⊗	239.0	20.6	0.0	37.9	18.8	0.0	44.9	0.11x	1.00x
098	52.3	0.0	11.2	🟥 31.8	1.3	788.2	🟥 22.6	0.3	271.6	🟥 24.0	0.0 ⊗	6.0	🟩 **30.7**	0.0	302.4	0.59x	-
336	🟩 **2.8**	0.0	1.2	🟥 3.5	9.1	22.5	🟥 2.2	0.1 ⊗	83.5	7.6	0.0	0.2	3.3	0.0	30.4	1.20x	-
353	7.6	0.0	1.0	🟥 6.5	0.5	52.6	🟥 4.6	0.3	211.8	🟥 31.5	1.6 ⊗	0.8	🟩 **6.3**	0.0	44.5	0.83x	-
523	123.7	0.0	3.4	🟥 34.7	0.3	240.2	🟥 27.5	0.3	920.3	348.7	0.0	2.8	🟩 **37.1**	0.0	460.1	0.30x	-
542	0.9	0.0	0.1	🟩 **0.8**	0.0	3.3	🟥 0.7	0.1	42.8	🟥 ×	×	×	🟩 **0.8**	0.0	2.4	0.89x	1.00x
566	83.6	0.0	4.9	🟥 63.8	1.9	225.7	🟥 46.9	0.5	341.1	🟥 493.6	3.8 ⊗	3.2	🟩 **69.3**	0.0	162.8	0.83x	-
583	47.0	0.0	2.3	🟥 26.1	0.6	190.6	🟥 20.6	0.2	421.2	🟥 ×	×	×	🟩 **25.1**	0.0	342.6	0.53x	-
588	8.8	0.0	0.7	🟥 6.3	1.1	60.4	🟥 4.8	0.5	41.5	🟥 ×	×	×	🟩 **6.9**	0.0	102.7	0.78x	-
643	4.9	0.0	0.6	🟥 3.8	0.9	18.8	🟥 3.0	0.4	29.3	🟥 15.3	0.2 ⊗	0.5	🟩 **3.7**	0.0	40.0	0.76x	-
DCT	🟥 ×	×	×	🟥 ×	×	>1800	🟥 33.1	1.7 ⊗	719.4	184.7	0.0	8.0	🟩 **37.2**	0.0	123.5	-	-
													Average			0.68x	1.00x

× indicates time-out, crash, or a run completed without producing a solution; ⊗ indicates an out-of-core solution

Table 4: Runs on proprietary designs. Best legal solutions are emphasized in bold.



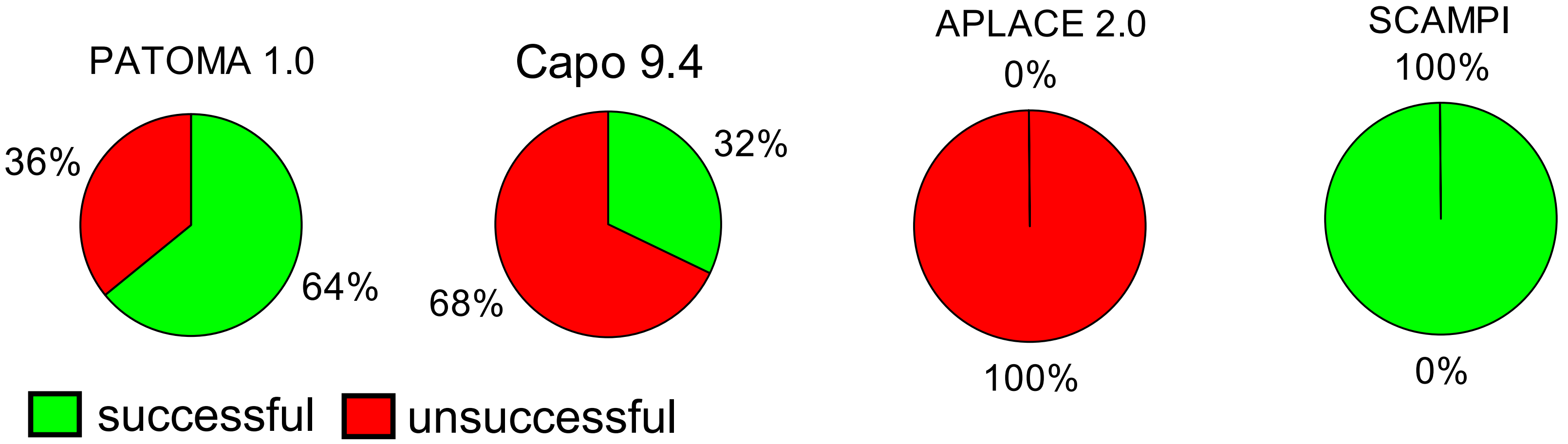
ibm -HB$^+$ bench	PATOMA 1.0			Capo 9.4			APlace 2.0			FengShui 5.1			SCAMPI				
	HPWL (e+06)	ovlp (%)	time (s)	HPWL (e+06)	ovlp (%)	time (s)	HPWL (e+06)	ovlp (%)	time (s)	HPWL (e+06)	ovlp (%)	time (s)	HPWL (e+06)	ovlp (%)	time (s)	vs. PATOMA (HPWL)	vs. CAPO (HPWL)
01	3.9	0.0	5.6	🟥 5.4	1.4	651.5	🟥 2.7	2.7	68.0	🟥 3.0	0.2 ⊗	16.6	🟩 **3.4**	0.0	62.0	0.87x	-
02	🟥 ×	×	×	19.1	0.0	1539.7	🟥 5.0	2.6	101.5	🟥 8.7	0.9 ⊗	43.6	🟩 **8.0**	0.0	139.6	-	0.42x
03	🟥 ×	×	×	🟥 ×	×	>1800	🟥 7.4	2.1	101.3	🟥 ×	×	×	🟩 **9.5**	0.0	104.6	-	-
04	🟥 ×	×	×	🟥 ×	×	>1800	🟥 8.2	2.8	113.9	🟥 10.8	0.2 ⊗	41.4	🟩 **12.3**	0.0	144.1	-	-
06	🟥 ×	×	×	🟥 ×	×	>1800	🟥 8.2	1.0	122.5	🟥 10.7	1.4 ⊗	36.0	🟩 **11.0**	0.0	170.0	-	-
07	16.8	0.0	13.6	15.8	0.0	115.31	🟥 13.7	1.4	218.4	🟥 37.1	0.0	5.1	🟩 **15.7**	0.0	99.9	0.93x	0.99x
08	🟥 ×	×	×	🟥 ×	×	>1800	🟥 16.6	1.0 ⊗	294.2	🟥 21.8	0.5 ⊗	60.6	🟩 **20.5**	0.0	188.4	-	-
09	🟥 ×	×	×	🟥 20.2	0.2	188.9	🟥 15.1	0.9	222.4	🟥 20.6	1.2 ⊗	42.9	🟩 **22.2**	0.0	182.0	-	-
10	🟥 ×	×	×	🟥 45.9	2.7	263.7	🟥 36.9	0.3	529.5	🟥 ×	×	×	🟩 **55.2**	0.0	319.9	-	-
11	🟩 **25.3**	0.0	49.2	28.1	0.0	140.5	🟥 24.5	1.1	270.3	🟥 30.4	0.2 ⊗	63.8	27.8	0.0	144.7	1.10x	0.99x
12	🟥 ×	×	×	🟩 **63.4**	0.0	482.2	🟥 ×	×	>1800	🟥 52.3	0.0 ⊗	39.2	67.6	0.0	406.1	-	1.07x
13	🟩 **37.5**	0.0	34.7	39.6	0.0	221.5	🟥 31.7	0.5	240.4	🟥 ×	×	×	42.2	0.0	209.6	1.13x	1.07x
14	68.7	0.0	70.9	68.2	0.0	320.7	🟥 57.1	1.0 ⊗	392.9	🟥 74.0	2.7	89.7	🟩 **66.4**	0.0	268.3	0.97x	0.97x
15	🟥 ×	×	×	🟥 ×	×	>1800	🟥 87.5	1.5	422.2	🟥 90.6	0.0 ⊗	100.3	🟩 **88.2**	0.0	375.9	-	-
16	🟩 **100.3**	0.0	74.4	106.9	0.0	431.5	🟥 89.8	0.3	528.1	🟥 ×	×	×	106.2	0.0	306.5	1.06x	0.99x
17	🟩 **141.4**	0.0	95.9	🟥 152.6	0.1	397.1	🟥 133.9	0.5	799.3	🟥 ×	×	×	152.7	0.0	385.7	1.08x	-
18	🟩 **72.6**	0.0	67.2	🟥 75.9	0.7	220.1	🟥 69.1	0.6	344.0	🟥 ×	×	×	77.8	0.0	192.3	1.07x	-
													Average			1.03x	0.93x

× indicates time-out, crash, or a run completed without producing a solution; ⊗ indicates an out-of-core solution

Table 5: Runs on IBM-HB$^+$. Best legal solutions are emphasized in bold.

Empirical results (cont'd)

- Success rates



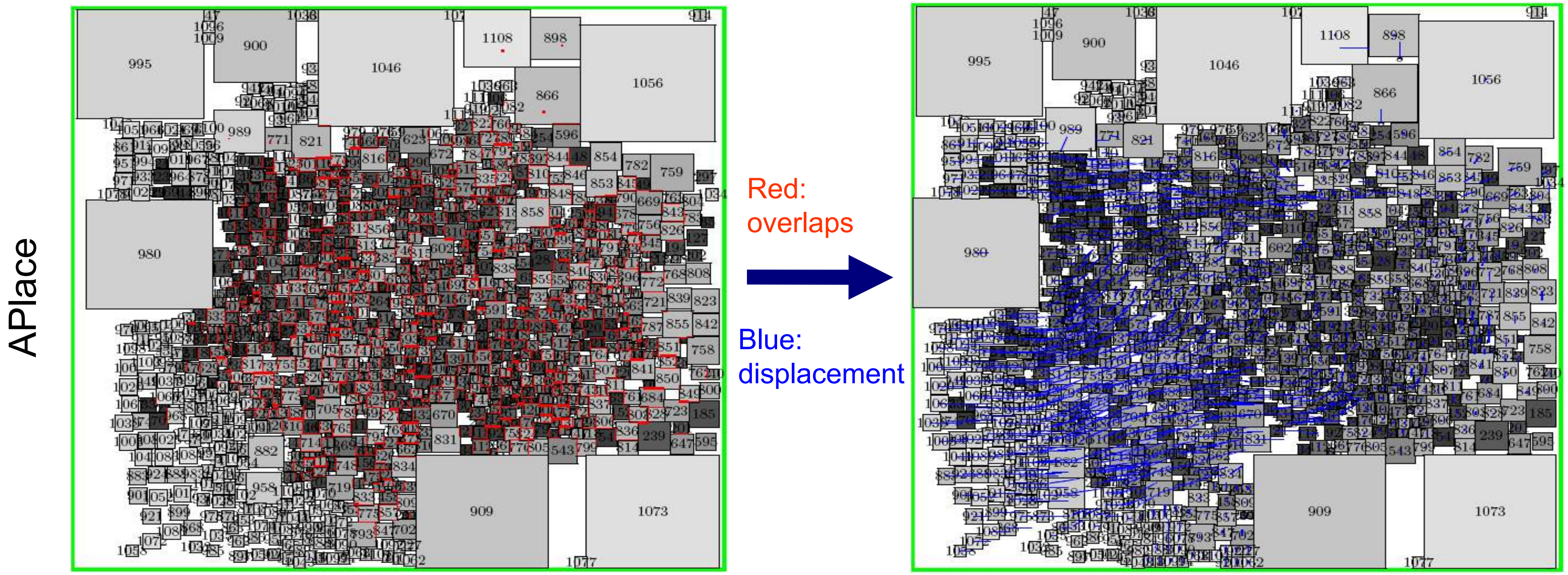
- Wirelength comparison
  - Averaged over successful runs of Capo 9.4 & PATOMA
  - SCAMPI achieves 3.5% and 14.5% better HPWL, resp.

Advantages & drawbacks of SCAMPI

- Advantages
  - Robust (68% and 36% better success rates than Capo9.4 and PATOMA)
  - Handles soft & hard macros, and std. cells
  - Handles obstacles & wide ranges of block dimensions
  - Good routability [J. A. Roy et. al, ISPD 2006]
- Potential drawbacks
  - Worse wirelength than some tools (e.g., APlace)
  - But APlace currently produces illegal floorplans
  - Stronger legalization can make APlace more competitive (see next slide)

Ongoing work: floorplan assistant

- AI-based floorplan legalizer
- Preliminary results:
  - ☐ Removes overlaps quickly, e.g., from APlace placements
  - ☐ Preserves placement
  - ☐ Some increase in wirelength seems inevitable



Red: overlaps

Blue: displacement

Conclusions

- RTL placement includes
  - Numerous hard & soft blocks, and standard cells
  - Macros, IP blocks, memories of very different sizes
  - Fixed obstacles
- SCAMPI solves hard instances using

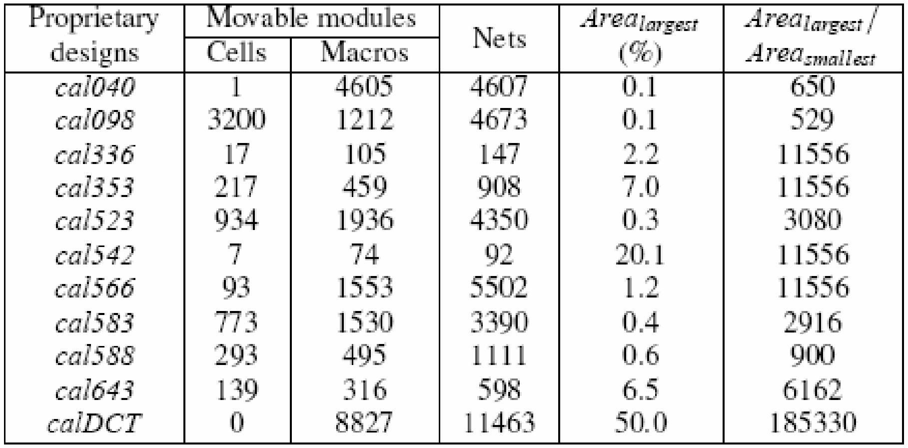
  - Selective floorplanning & macro clustering
  - Support for obstacles in the B*-tree representation
  - Ad hoc look-ahead floorplanning
  - Whitespace allocation by block densities
- Suite of hard floorplacement instances
  - http://vlsicad.eecs.umich.edu/BK/ISPD06bench
- SCAMPI is available in source code

Questions?

Reproducing difficult instances

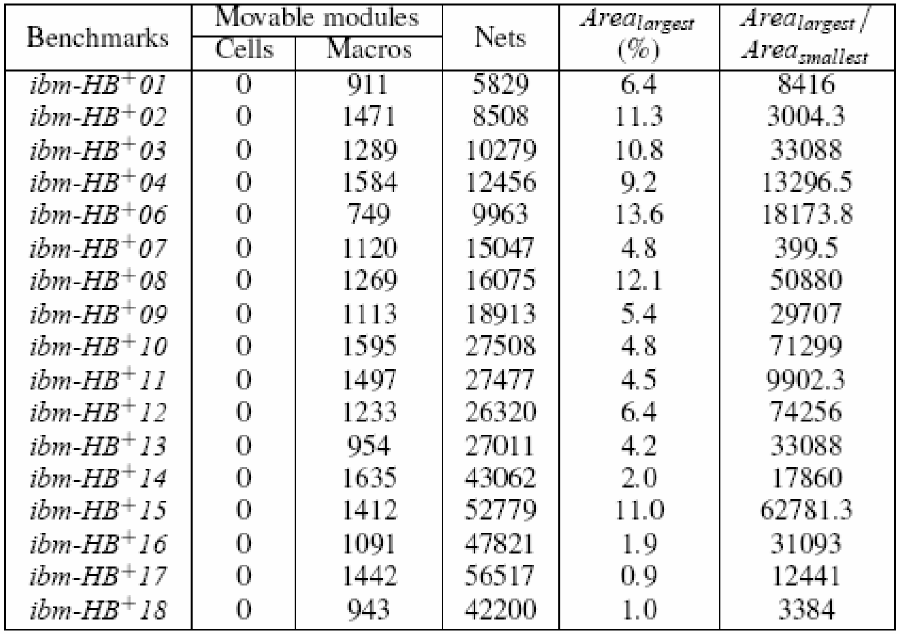
- In general, difficulties are from scale and/or large variations in module sizes
- We take IBM-HB (which were from IBM/ISPD‘98)
  - Std cells → macros
- We introduce IBM-HB+ (derived from IBM-HB)
  - An example of how to re-create difficult instances
  - Largest macro inflated 100%
  - Smaller macros shrunk to preserve total cell area

Benchmark characteristics



Proprietary designs	Movable modules		Nets	$Area_{largest}$ (%)	$Area_{largest}$ / $Area_{smallest}$
	Cells	Macros			
cal040	1	4605	4607	0.1	650
cal098	3200	1212	4673	0.1	529
cal336	17	105	147	2.2	11556
cal353	217	459	908	7.0	11556
cal523	934	1936	4350	0.3	3080
cal542	7	74	92	20.1	11556
cal566	93	1553	5502	1.2	11556
cal583	773	1530	3390	0.4	2916
cal588	293	495	1111	0.6	900
cal643	139	316	598	6.5	6162
calDCT	0	8827	11463	50.0	185330

Table 2: Characteristics of the proprietary designs.

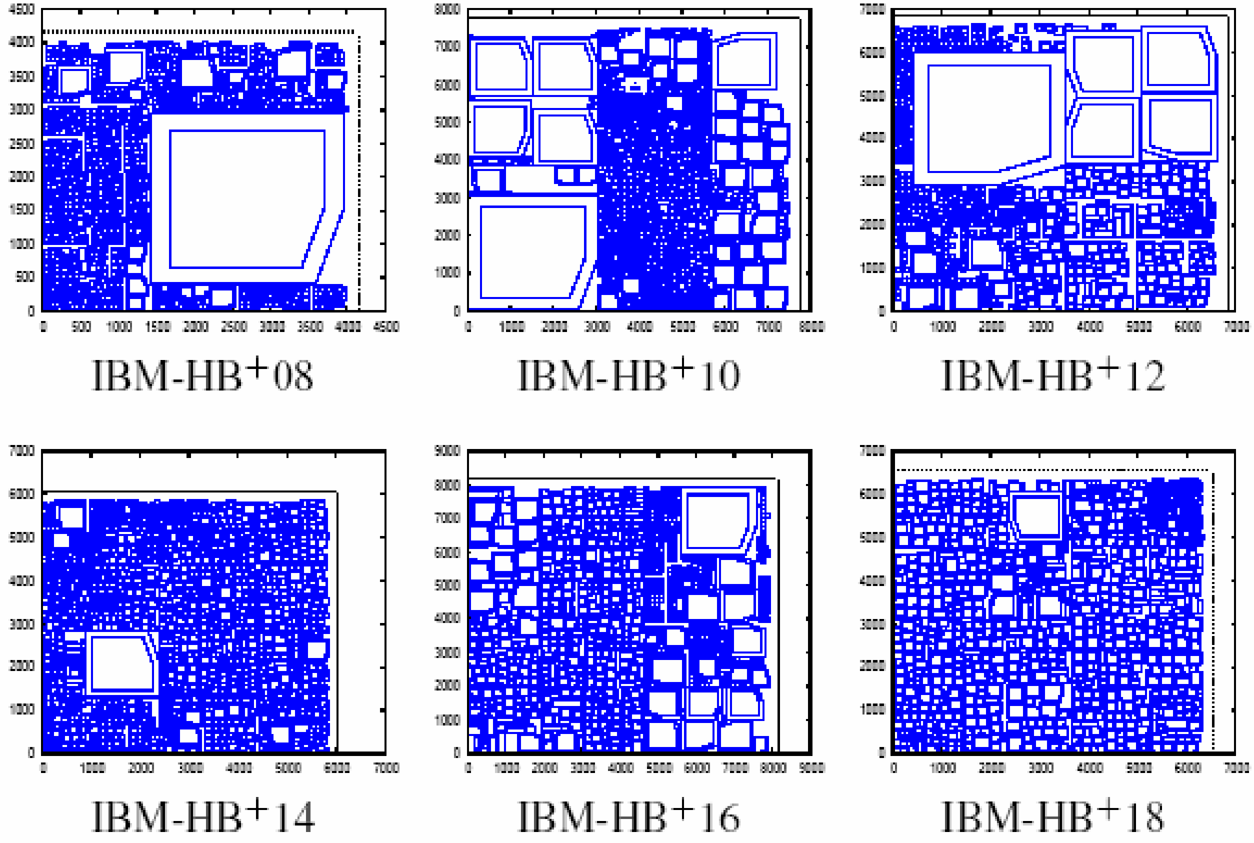


Benchmarks	Movable modules		Nets	$Area_{largest}$ (%)	$Area_{largest}$ / $Area_{smallest}$
	Cells	Macros			
ibm-HB$^+$01	0	911	5829	6.4	8416
ibm-HB$^+$02	0	1471	8508	11.3	3004.3
ibm-HB$^+$03	0	1289	10279	10.8	33088
ibm-HB$^+$04	0	1584	12456	9.2	13296.5
ibm-HB$^+$06	0	749	9963	13.6	18173.8
ibm-HB$^+$07	0	1120	15047	4.8	399.5
ibm-HB$^+$08	0	1269	16075	12.1	50880
ibm-HB$^+$09	0	1113	18913	5.4	29707
ibm-HB$^+$10	0	1595	27508	4.8	71299
ibm-HB$^+$11	0	1497	27477	4.5	9902.3
ibm-HB$^+$12	0	1233	26320	6.4	74256
ibm-HB$^+$13	0	954	27011	4.2	33088
ibm-HB$^+$14	0	1635	43062	2.0	17860
ibm-HB$^+$15	0	1412	52779	11.0	62781.3
ibm-HB$^+$16	0	1091	47821	1.9	31093
ibm-HB$^+$17	0	1442	56517	0.9	12441
ibm-HB$^+$18	0	943	42200	1.0	3384

Table 3: Characteristics of the IBM-HB$^+$ benchmarks.

IBM-HB+

- http://vlsicad.eecs.umich.edu/BK/ISPD06bench



IBM-HB^{+}08 IBM-HB^{+}10 IBM-HB^{+}12

IBM-HB^{+}14 IBM-HB^{+}16 IBM-HB^{+}18

Figure 2: Six of the seventeen IBM-HB^{+} benchmarks.

Floorist

- Constraint-driven floorplan repair*
- Build constraint graphs from placement ordering
  - Represent pair-wise relationships between modules
- Perform conflict-directed iterative repair on graphs
  - Overlapping pairs are initially constrained
  - Induce constraints to resolve overlaps, or
  - Identify blocks on critical paths,
    modify their relationships with other modules
- Translate constraint graphs back
- APlace + Floorist = best-seen results for IBM-HB

* M. Moffitt, A. N. Ng, “Constraint-driven floorplan repair”, DAC 2006

FengShui placements