SimPL: An Effective Placement Algorithm

Myung-Chul Kim, Dong-Jin Lee and Igor L. Markov
Dept. of EECS, University of Michigan

Designing New Chips Quickly

- Idea 1: describe ICs in HW descr. languages (HDL)
  - Verilog, SystemVerilog, C, SystemC
- Idea 2: use software tools to compile HDL into silicon
  - Many intermediate representations, many passes
- Idea 3: use software tools to optimize chips
  - Functional (logic) optimization A*B+C
    (Boolean & polynomial algebras)
  - Physical optimization: gate locations
    (computational geometry, non-convex optimization)
- Idea 4: use software tools to verify chips
- Idea 5: use software tools to optimize test sequences
- Idea 6: use software to debug chips

Global Placement: Motivation

- Interconnect lagging in performance while transistors continue scaling
  - Circuit delay, power dissipation and area dominated by interconnect
  - Routing quality highly controlled by placement
- Circuit size and complexity rapidly increasing
  - Scalable placement algorithm is critical
  - Simplicity, integration with other optimizations
**Goals in Placement**

- Find good relative ordering of cells
  - Minimize wire length and congestion
  - Maximize timing slack
- Find good spacing of cells
  - Eliminate wiring congestion problems
  - Provide space for post placement stages
    - clock trees
    - buffer insertion
    - timing correction
- Find good global position

---

**Optimize Relative Order**

To spread ...

.. or not to spread
Place to the left

A B C

... or to the right

A B C

Optimize Relative Order

Without whitespace, placement is dominated by ordering

Example of Global Placement (APlace 2.04 from UCSD)
Example of Global Placement (mFar from UCSB)

Placement Formulation

- **Objective:** Minimize estimated wirelength
  - Half-perimeter wirelength (HPWL)
  - $\sum (\max X - \min X) + (\max Y - \min Y)$

- **Subject to constraints:**
  - **Legality:** Row-based placement with no overlaps
  - **Routability:** Limiting local interconnect congestion for successful routing
  - **Timing:** Meeting performance target of a design

Quadratic Placement

- Consider a graph first, not a hypergraph
- Minimize $\Sigma (x_i-x_j)^2+(y_i-y_j)^2$ (the sum is over $e_{ij}$)
  - Seems unrelated to $\Sigma |x_i-x_j|+|y_i-y_j|$ but can still be separated into $x$- and $y$-components
- **Physical analogy:** Hooke’s law
  - Consider an elastic spring, spread by $x$
  - Force $F=-kx$ ($k$ is the spring constant)
  - Energy $E=\frac{1}{2}kx^2$
  - Our goal: minimize the energy of the system

Iterative Optimization

- A system of springs will only settle in a minimum
Prior Work

- Ideal placer
  - Fast runtime without sacrificing solution quality
  - Simplicity, integration with other optimizations

Key features of SimPL

- Flat quadratic placement
- Primal dual optimization
  - Closing the gap between upper and lower bounds

Common Analytical Placement Flow

SimPL Flow

- B2B graph building
- Linear system solver
- Look-ahead legalization (Upper-Bound)
- Global placement
- Legalization and detailed placement
- Converge


We delegate final legalization and detailed placement to FastPlace-DP [M. Pan, et al, “An Efficient and Effective Detailed Placement Algorithm”, ICCAD2005]
SimPL: Look-ahead Legalization

- **Purpose**: Produces almost-legal placement (Upper-Bound) while preserving the relative cell ordering given by linear system solver (Lower-Bound)

- **Identify target region**
  - Find overflow bin $b$
  - Create a minimal wide enough bin cluster $B$ around $b$

- **Perform geometric top-down partitioning**
  - Find cell area median ($C_c$) and whitespace median ($C_B$)
  - Assign cells ($C_c$) to corresponding partitions ($C_B$)

- **Non-linear scaling**
  - Form stripe regions
  - Move cells across stripe regions in-order based on whitespace
SimPL: Look-ahead Legalization (3)

Example (adaptec1)

Look-ahead legalization stops when target regions become small enough.

SimPL: Using legal locations as anchors

- **Purpose:** Gradually perturb the linear system to generate lower-bound solutions with less overlap.

- **Anchors and Pseudonets**
  - Look-ahead locations used as fixed, zero-area anchors.
  - Anchors and original cells connected with 2-pin pseudonets.
  - Pseudonet weights grow linearly with iterations.

Next illustration: Tug-of-war between low-wirelength and legalized placements.

SimPL Iterations on Adaptec1 (1)

- **Iteration=0 (Init WL Opt.)**
- **Iteration=1 (Upper Bound)**
- **Iteration=2 (Lower Bound)**
- **Iteration=3 (Upper Bound)**
Convergence of SimPL

- Legal solution is formed between two bounds

The convergence of SimPL is illustrated through a graph showing the wirelength upper and lower bounds as a function of the iteration number. The graph demonstrates how the solution converges within the bounds.

Empirical Results: ISPD05 Benchmarks

- Experimental setup
  - Single threaded runs on a 3.2GHz Intel core i7 Quad CPU Q660 Linux workstation
  - HPWL is computed by GSRC Bookshelf Evaluator
  - < 5000 lines of code in C++, including CG solver for sparse linear systems (w Jacobi preconditioner)

The table below summarizes the empirical results for various benchmarks:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>APLACE2.0</th>
<th>HPWL</th>
<th>Runtime</th>
<th>FASTPLACE3.0</th>
<th>HPWL</th>
<th>Runtime</th>
<th>MPL6</th>
<th>HPWL</th>
<th>Runtime</th>
<th>SimPL</th>
<th>HPWL</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAPTE1</td>
<td>211K</td>
<td>78.35</td>
<td>35.02</td>
<td>78.16</td>
<td>2.50</td>
<td>77.91</td>
<td>18.36</td>
<td></td>
<td></td>
<td>76.87</td>
<td>2.47</td>
<td></td>
</tr>
<tr>
<td>ADAPTE2</td>
<td>255K</td>
<td>95.70</td>
<td>50.57</td>
<td>93.56</td>
<td>3.66</td>
<td>92.04</td>
<td>19.91</td>
<td></td>
<td></td>
<td>90.37</td>
<td>3.40</td>
<td></td>
</tr>
<tr>
<td>ADAPTE3</td>
<td>452K</td>
<td>218.52</td>
<td>119.53</td>
<td>213.85</td>
<td>8.48</td>
<td>214.16</td>
<td>58.92</td>
<td></td>
<td></td>
<td>206.38</td>
<td>6.68</td>
<td></td>
</tr>
<tr>
<td>BIGBLUE1</td>
<td>275K</td>
<td>100.02</td>
<td>44.91</td>
<td>96.32</td>
<td>3.77</td>
<td>96.89</td>
<td>22.82</td>
<td></td>
<td></td>
<td>95.83</td>
<td>2.47</td>
<td></td>
</tr>
<tr>
<td>BIGBLUE2</td>
<td>558K</td>
<td>153.75</td>
<td>100.96</td>
<td>154.91</td>
<td>9.62</td>
<td>152.34</td>
<td>61.55</td>
<td></td>
<td></td>
<td>143.56</td>
<td>7.58</td>
<td></td>
</tr>
<tr>
<td>BIGBLUE3</td>
<td>1.10M</td>
<td>411.59</td>
<td>209.24</td>
<td>365.59</td>
<td>21.59</td>
<td>344.10</td>
<td>85.23</td>
<td></td>
<td></td>
<td>336.19</td>
<td>13.02</td>
<td></td>
</tr>
<tr>
<td>BIGBLUE4</td>
<td>2.18M</td>
<td>871.29</td>
<td>489.05</td>
<td>834.19</td>
<td>40.93</td>
<td>829.44</td>
<td>189.83</td>
<td></td>
<td></td>
<td>796.78</td>
<td>37.37</td>
<td></td>
</tr>
</tbody>
</table>

The average runtime performance is also indicated for each benchmark.
Empirical Results: Scalability Study

- Take an existing design (ISPD 2005) and split each movable cell into two cells of smaller size
  - Each connection to the original cell is inherited by one of two split cells, which are connected by a 2-pin net

![Diagram showing the splitting of a cell into two smaller cells.]

Not in ICCAD paper

Speeding Up Placement Using Parallelism

- Parallelization by simplification
  - CAD optimizations often require redesign and simplification of entire algorithms to use fewer components
  - Each component of SimPL algorithm is amenable to parallelism.

- Runtime bottlenecks in the sequential variant of the SimPL algorithm
  1. Conjugate gradient-based linear system solver
  2. B2B net model updates
  3. Top-down partitioning and non-linear scaling

Runtime Break-up

![Pie chart showing the distribution of runtime for various tasks in the placement process.]

Parallelism in Conjugate Gradient Solver

- Main runtime bottleneck in SimPL: Linear system solver (conjugate gradient)

- Coarse-grain row partitioning
  - Implemented using OpenMP3.0 compiler intrinsic

- SSE2 (Streaming SIMD Extensions) instructions
  - Process 4 multiple data with a single instruction
  - Marginal runtime improvement in SpMxV

- Reducing memory bandwidth demand of SpMxV
  - CSR (Compressed Sparse Row) format

Parallelism in CG Solver - Example

```c
// inner product of two float vectors x and y
float inner_product(vector<float>& x, vector<float>& y)
{
    float p_acc[4], inner_product=(float)0.;
    __m128 X, Y, acc = __mm_set_zero_ps();
    unsigned i;
    #pragma omp parallel for schedule(static)
    private(X,Y) lastprivate(i) reduction(+:acc)
    num_threads(NUM_CORES)
    for (i=0 ; i<x.size()-4 ; i+=4)
    {
        X = __mm_load_ps(&x[i]);
        Y = __mm_load_ps(&y[i]);
        acc = __mm_add_ps(acc, __mm_mul_ps(X, Y));
    }
    __mm_store_ps(p_acc, acc);
    for (; i<x.size() ; i++)
        inner_product+=x[i]+y[i];
    return inner_product;
}
```

B2B net model update
- B2B model is separable
- Can process the x and y cases in parallel

If more than 2 cores available:
- Process the x and y cases in parallel
- Additionally, split the nets of the netlist into equal groups that can be processed by multiple threads.

Runtime Break-ups with SSE support

Parallelism in Look-ahead Legalization (1)
- Look-ahead legalization started consuming a significant fraction of overall runtime
- Top-down geometric partitioning and non-linear scaling (T&N) are amenable to parallelization
  - Top-down partitioning generates an increasing number of subtasks of similar sizes which can be solved in parallel
  - After each level of T&N on bin cluster, each thread generates two sub-clusters with similar numbers of cells.
Parallelism in Look-ahead Legalization (2)

- LAL keeps the global queue of bin clusters $Q$ and each thread has a private queue of bin clusters $Q_i$
- Static partitioning
  - Assign initial bin clusters to available threads such that each thread has similar number of bin clusters to start
- Subtask updates
  - Thread $t_i$ adds only one of two sub-clusters to its own cluster queue $Q_i$ for the next level of T&N, while the remainder is added to the global cluster queue $Q$
- Dynamic task retrieving
  - Whenever $Q_i$ of a thread $t_i$ becomes empty, the thread $t_i$ dynamically retrieves clusters from the global cluster queue $Q$. The number of clusters to be retrieved $N = \max(Q.size()/N\_threads, 1)$

Empirical Results – Overall Speed-ups

- Experimental setup
  - Multi threaded runs on an 8-core AMD-based system with four dual-core CPUs and 16GByte RAM
  - Each CPU was Opteron 880 processor running at 2.4GHz with 1024KB cache

Empirical Results – Component Speed-ups

<table>
<thead>
<tr>
<th>Ckts</th>
<th>1 CORE</th>
<th>2 THREADS</th>
<th>4 THREADS</th>
<th>8 THREADS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CG</td>
<td>CG+SSE</td>
<td>CG</td>
<td>CG+SSE</td>
</tr>
<tr>
<td>AD1</td>
<td>1.37</td>
<td>1.88</td>
<td>2.21</td>
<td>1.45</td>
</tr>
<tr>
<td>AD2</td>
<td>1.61</td>
<td>1.77</td>
<td>2.09</td>
<td>1.50</td>
</tr>
<tr>
<td>AD3</td>
<td>1.57</td>
<td>1.76</td>
<td>2.17</td>
<td>1.48</td>
</tr>
<tr>
<td>AD4</td>
<td>1.50</td>
<td>1.65</td>
<td>2.07</td>
<td>1.51</td>
</tr>
<tr>
<td>BB1</td>
<td>1.57</td>
<td>1.77</td>
<td>2.17</td>
<td>1.27</td>
</tr>
<tr>
<td>BB2</td>
<td>1.62</td>
<td>1.87</td>
<td>2.24</td>
<td>1.48</td>
</tr>
<tr>
<td>BB3</td>
<td>1.54</td>
<td>1.73</td>
<td>2.25</td>
<td>1.60</td>
</tr>
<tr>
<td>BB4</td>
<td>1.94</td>
<td>2.12</td>
<td>2.01</td>
<td>2.01</td>
</tr>
<tr>
<td>Avg</td>
<td>1.60×</td>
<td>1.92×</td>
<td>2.28×</td>
<td>1.54×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ckts</th>
<th>4 THREADS</th>
<th>8 THREADS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CG</td>
<td>CG+SSE</td>
</tr>
<tr>
<td>AD1</td>
<td>2.03</td>
<td>2.37</td>
</tr>
<tr>
<td>AD2</td>
<td>2.12</td>
<td>3.01</td>
</tr>
<tr>
<td>AD3</td>
<td>1.88</td>
<td>3.17</td>
</tr>
<tr>
<td>AD4</td>
<td>1.81</td>
<td>3.03</td>
</tr>
<tr>
<td>BB1</td>
<td>2.02</td>
<td>3.14</td>
</tr>
<tr>
<td>BB2</td>
<td>1.72</td>
<td>2.97</td>
</tr>
<tr>
<td>BB3</td>
<td>1.68</td>
<td>3.04</td>
</tr>
<tr>
<td>BB4</td>
<td>2.71</td>
<td>4.48</td>
</tr>
<tr>
<td>Avg</td>
<td>2.00×</td>
<td>3.21×</td>
</tr>
</tbody>
</table>
On-going Research

- Integration with physical synthesis
  - Look-ahead placement offers opportunity for early estimation of circuit parameters
    - Routability / Congestion look-ahead (*SimPLR*)
    - Timing look-ahead
    - Power-density look-ahead
  - Improving the speed and quality of physical synthesis

- Macro placement
- Structured signal-aware placement

Conclusions

- New flat quadratic placement algorithm: *SimPL*
  - Novel primal-dual based approach
  - Amenable to integration with physical synthesis

- Self-contained, compact implementation
  - Fastest among available academic placers
  - Highly competitive solution quality
  - Amenable to parallelism

Questions and Answers

Thank you!
Time for Questions