UBC System on a Chip research lab

Self-Hosted Placement for Massively Parallel Processor Arrays (MPPAs)

Graeme Smecher,
Steve Wilton, **<u>Guy Lemieux</u>**

Thursday, December 10, 2009
FPT 2009

Landscape

- Massively Parallel Processor Arrays
  - 2D array of processors
    - Ambric: 336, PicoChip: 273, AsAP: 167, Tilera: 100
  - Processor-to-processor communication
- Placement (locality) matters
  - Tools/algorithms immature

Opportunity

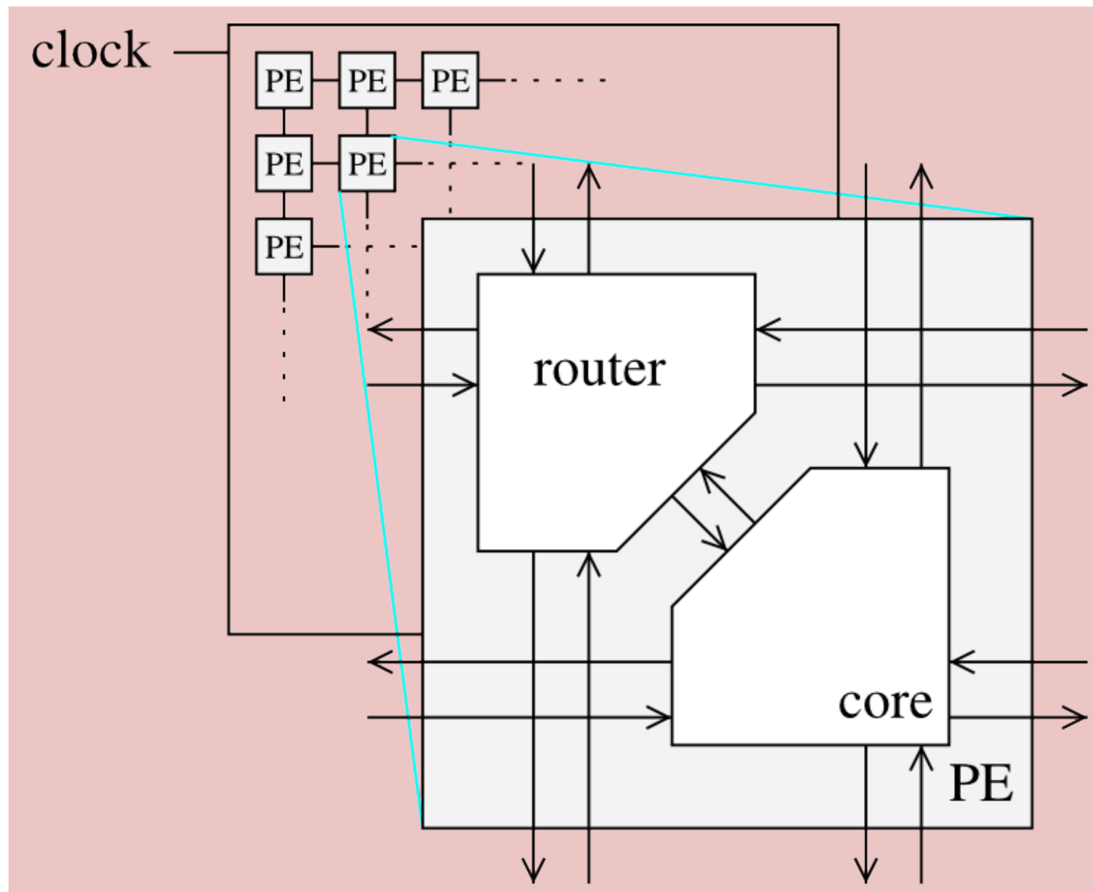
- MPPAs track Moore's Law
  - Array size grows
    - E.g. Ambric:336, Fermi:512
- Opportunity for FPGA-like CAD?
  - Compiler-esque speed needed
  - Self-hosted *parallel* placement
    - M x N array of CPUs computes placement for M x N programs
    - Inherently scalable

Overview

- **Architecture**
- Placement Problem
- Self-Hosted Placement Algorithm
- Experimental Results
- Conclusions

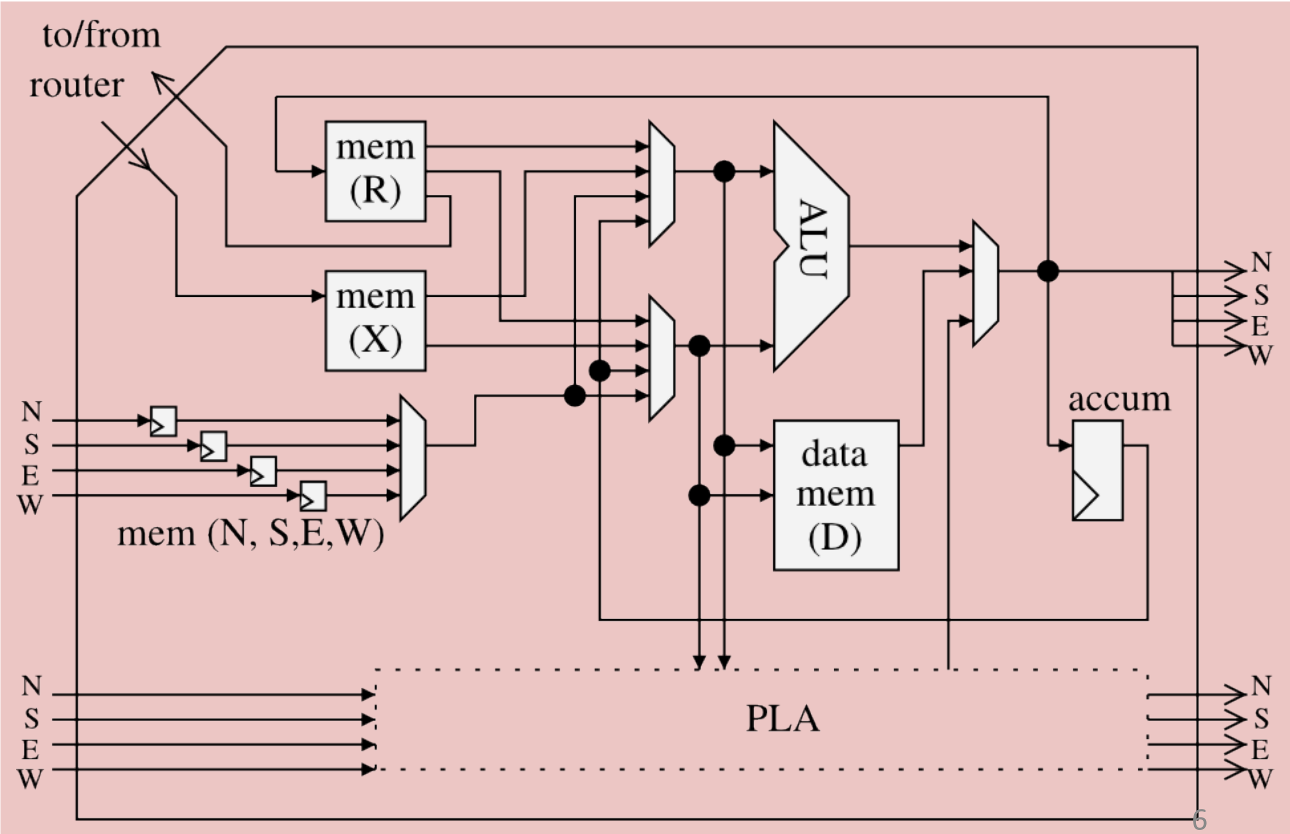
MPPA Architecture

- 32 x 32 = 1024 PEs
- PE = RISC + Router
- RISC core
  - In-order pipeline
  - More powerful PE than prev talk
- Router
  - 1-cycle per hop



MPPA Architecture (cont' d)

- Simple RISC core
  - More capable than RVEArch
- Small local RAM



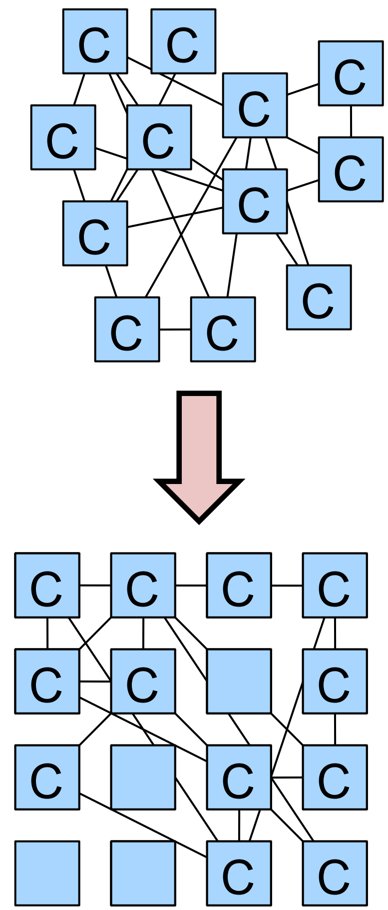
Overview

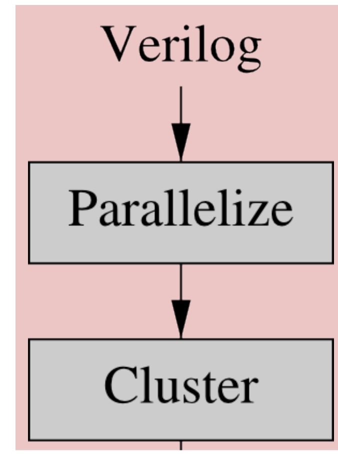
- Architecture
- **Placement Problem**
- Self-Hosted Placement Algorithm
- Experimental Results
- Conclusions

Placement Problem

- Given: netlist graph
  - Set of "cluster" programs
    - One per PE
  - Communication paths

- Find: good 2D placement
  - Use simulated annealing
  - E.g., minimum total Manhattan wirelength





Overview

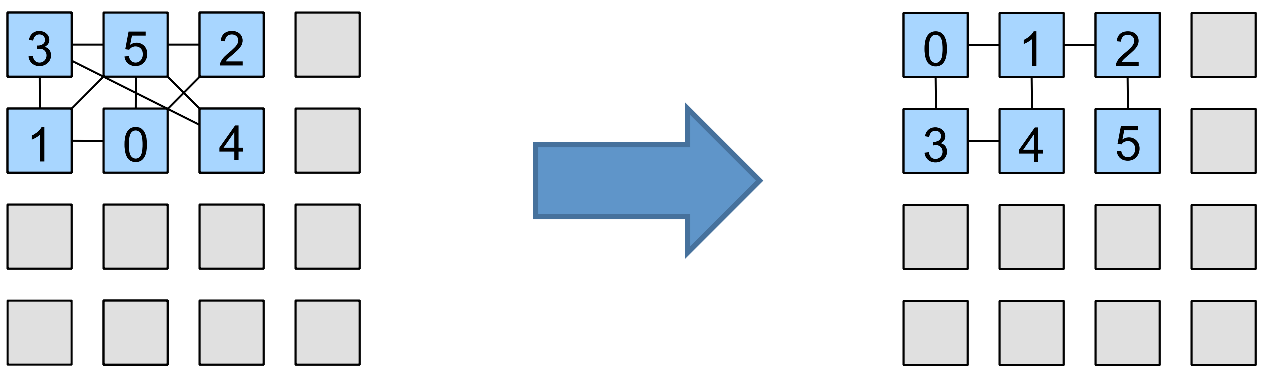
- Architecture
- Placement Problem
- **Self-Hosted Placement Algorithm**
- Experimental Results
- Conclusions

Self-Hosted Placement

- Idea from Wrighton and DeHon, FPGA03
  - Use FPGA to place itself
  - Imbalanced: tiny problem size needs **HUGE** FPGA
  - N-FPGAs needed to place 1-FPGA design

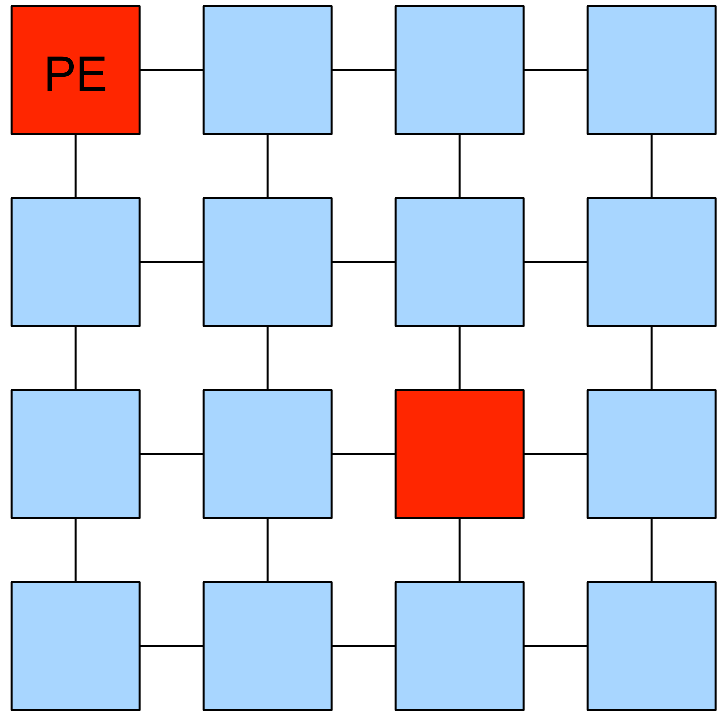
Self-Hosted Placement

- Use MPPA to place itself
  - PE powerful enough to place itself
  - Removes imbalance
  - 2 x 3 PEs to place 6 “clusters” into 2 x 3 array



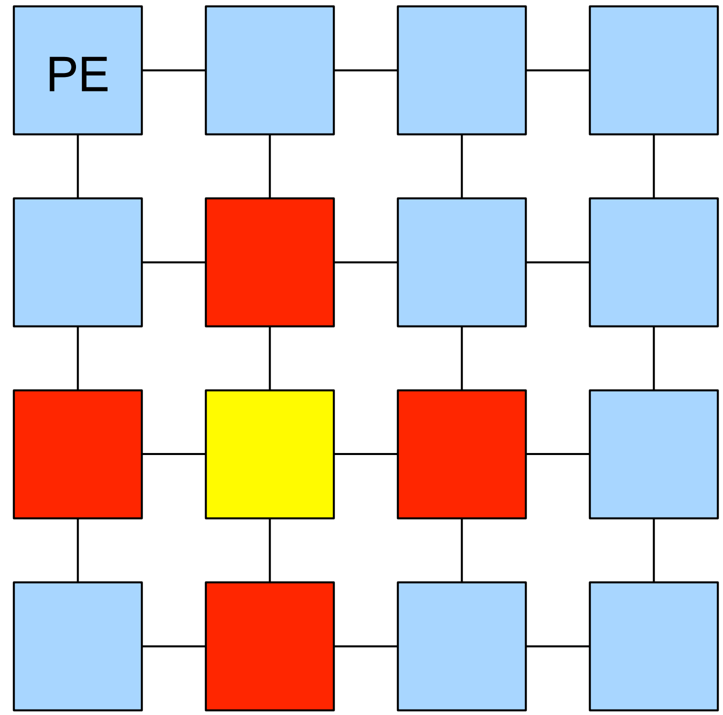
Regular Simulated Annealing

1. initial: random placement
2. for T in {temperatures}
    1. for n in 1..N clusters
        1. Randomly select 2 blocks
        2. Compute swap cost
        3. Accept swap if
           i) cost decreases, or
           ii) random trial succeeds



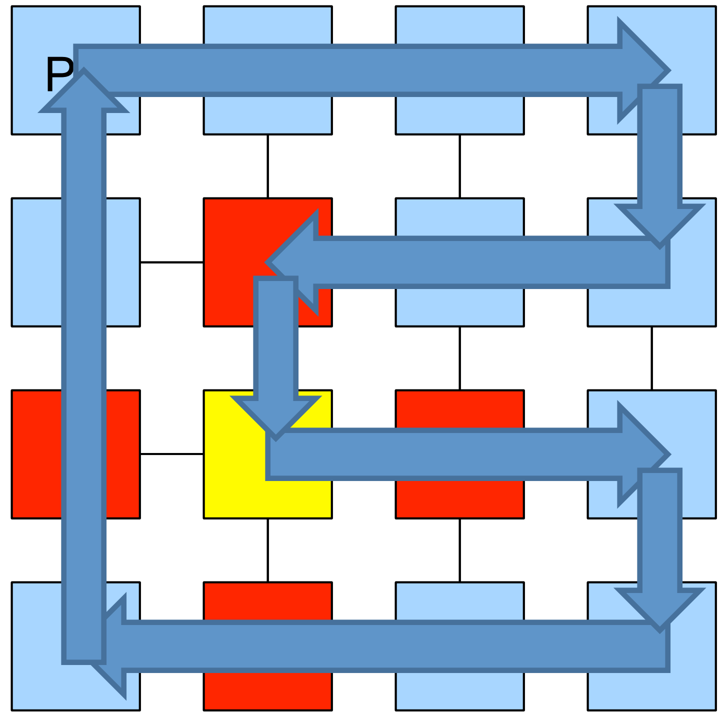
Modified Simulated Annealing

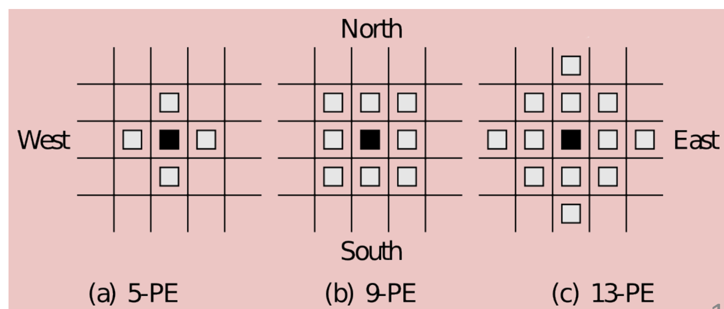
1. initial: random placement
2. for T in {temperatures}
    1. for n in 1..N clusters
        1. **Consider all pairs in neighbourhood of n**
        2. Compute swap cost
        3. Accept swap if
           i) cost decreases, or
           ii) random trial succeeds



Self-Hosted Simulated Annealing

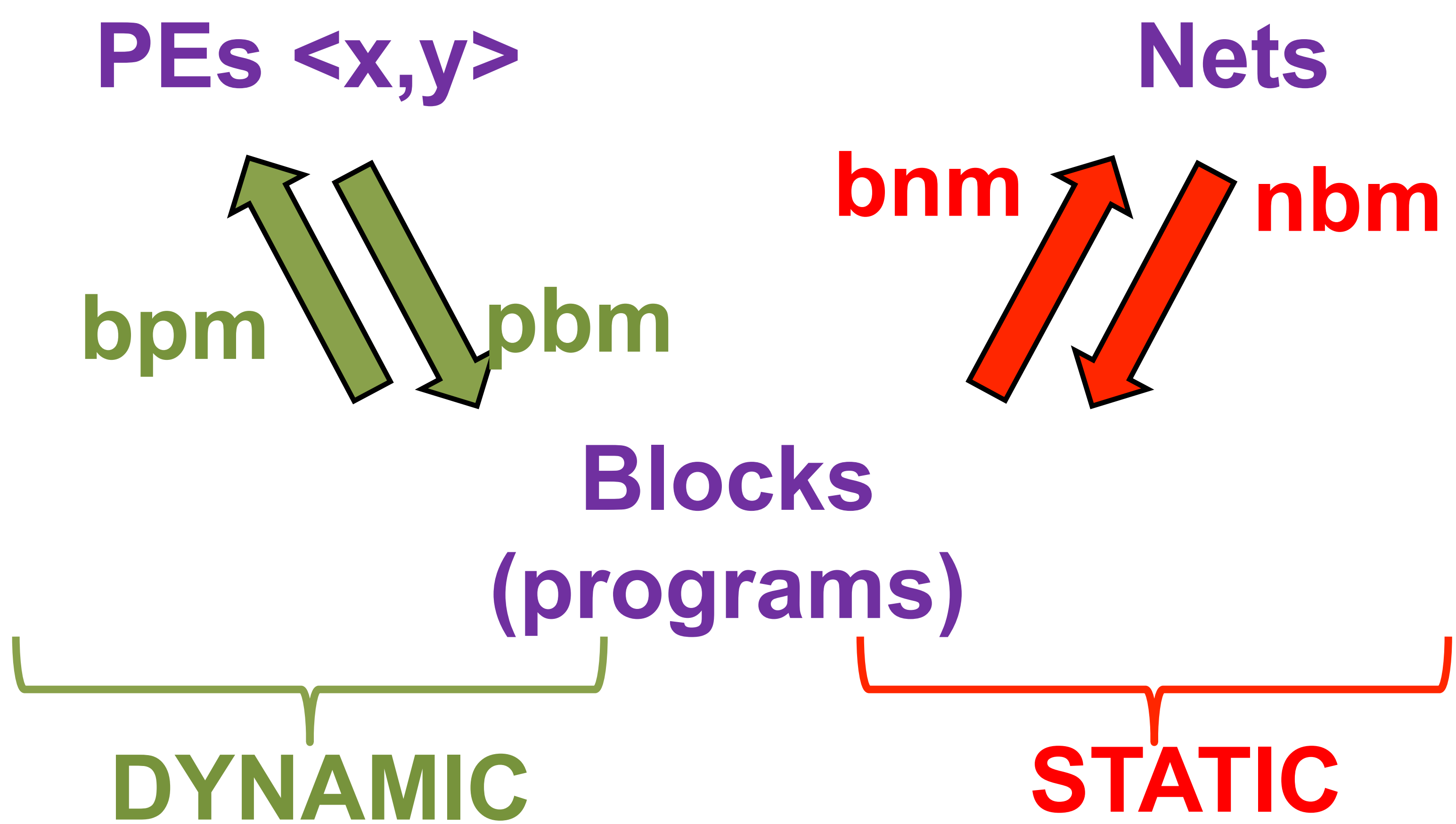
1. initial: random placement
2. for T in {temperatures}
    1. **for n in 1..N clusters**
        1. **Update position chain**
        2. Consider all pairs in neighbourhood of n
        3. Compute swap cost
        4. Accept swap if
           i) cost decreases, or
           ii) random trial succeeds





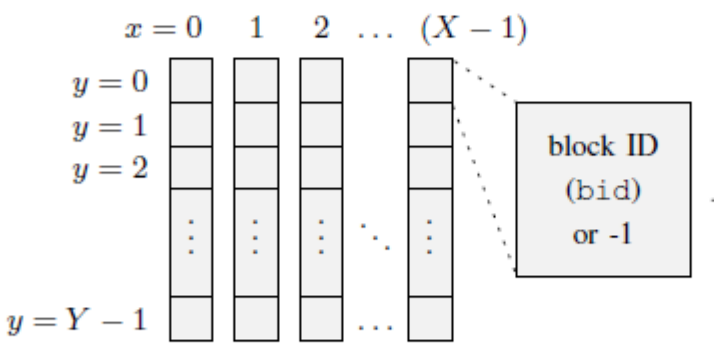
Algorithm Data Structures

- Place-to-block maps
- Net-to-block maps



Algorithm Data Structures

pbm



Full map in each PE

Partial map in each PE

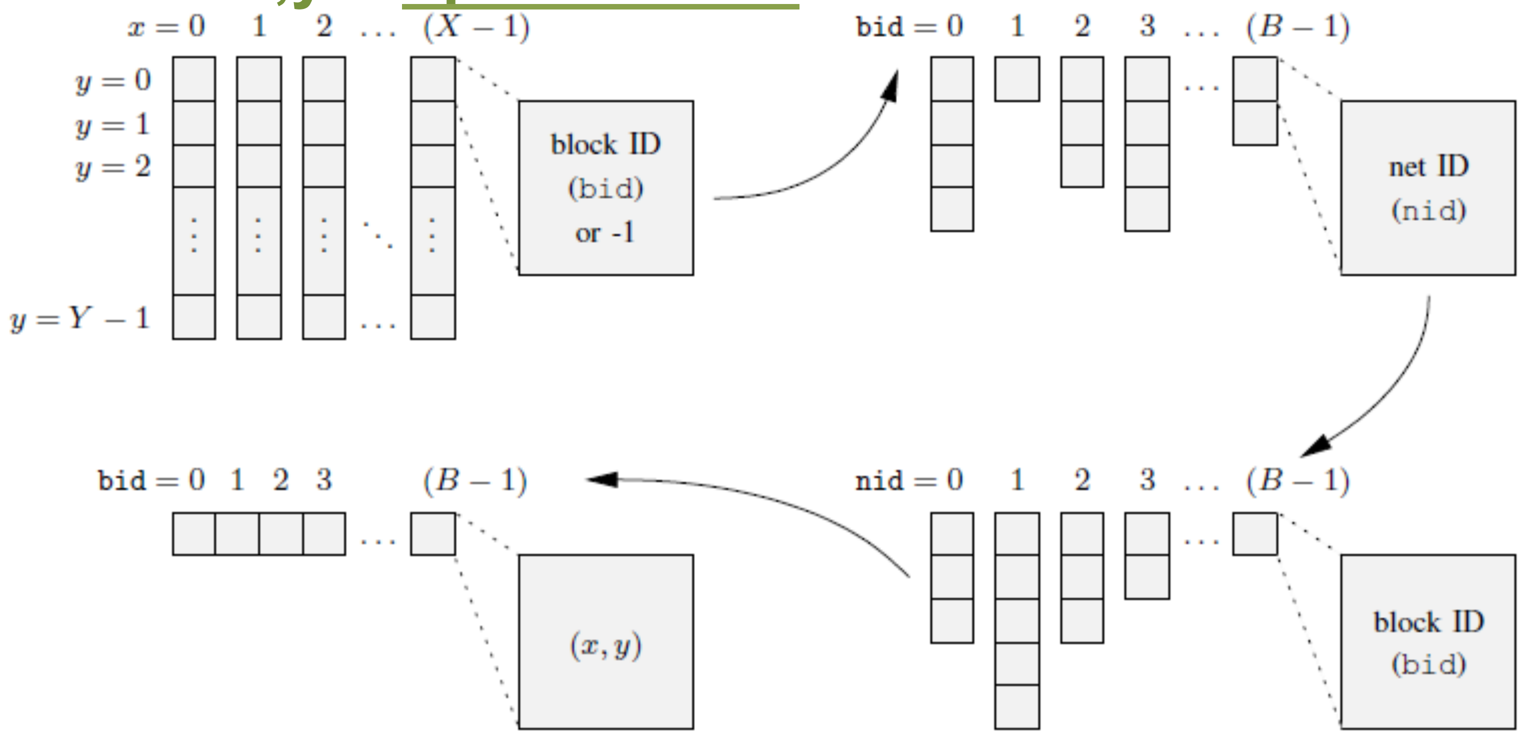
Swap Transaction

- PEs pair up
  - Deterministic order, hardcoded in algorithm
- Each PE computes cost for own BlockID
  - Current placement cost
  - After cost if BlockID was swapped
- PE 1 sends cost of swap to PE 2
  - PE 2 adds costs, determines if swap accepted
  - PE 2 sends decision back to PE 1
  - PE 1 and PE2 exchange data structures if swap

Data Structure Updates

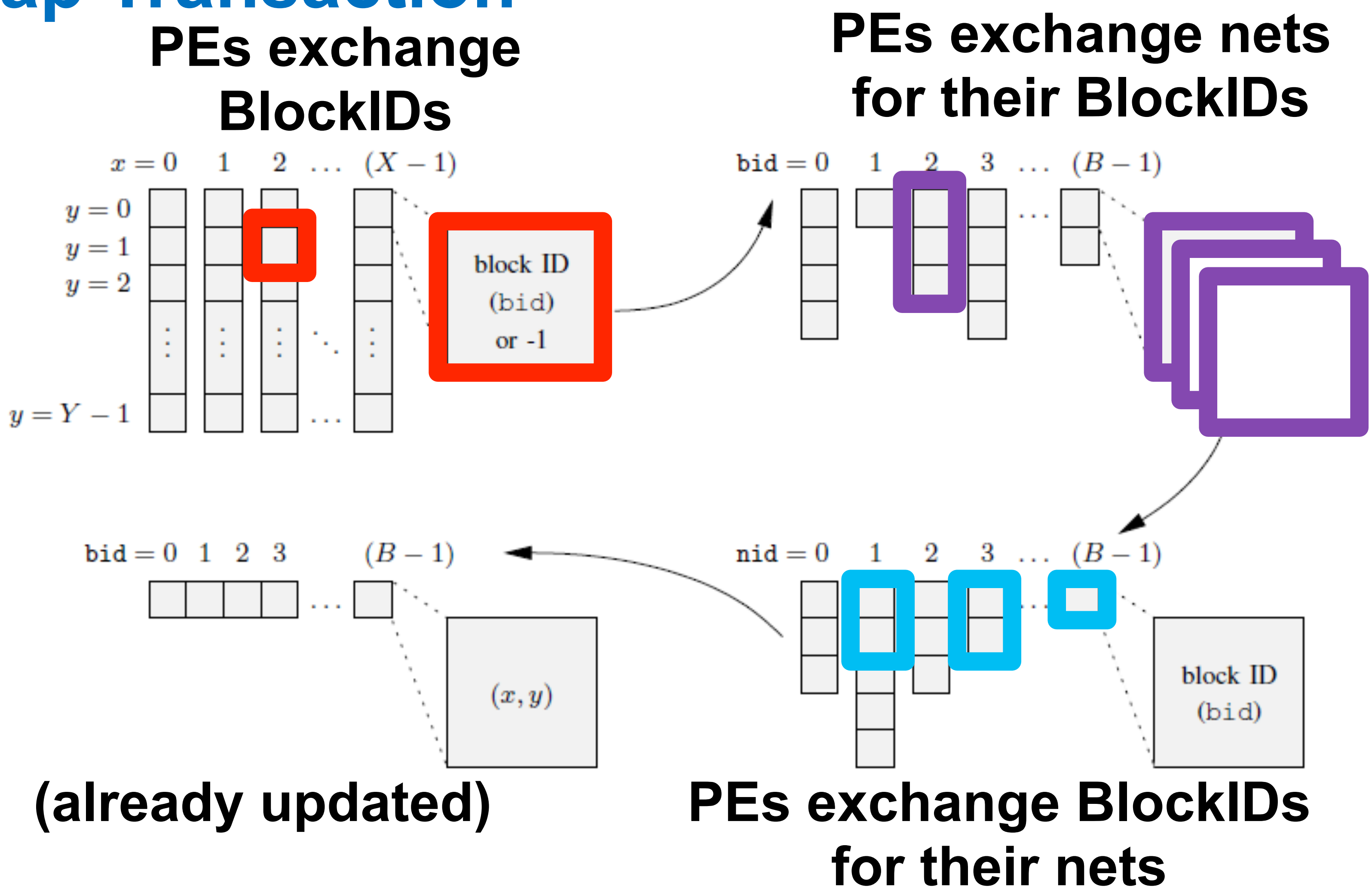
Dynamic structures
Local <x,y>: update on swap
Other <x,y>: update chain

Static structures
Exchanged with swap



Data Communication

Swap Transaction



Overview

- Architecture
- Placement Problem
- Self-Hosted Placement Algorithm
- **Experimental Results**
- Conclusions

Methodology

- Three versions of Simulated Annealing (SA)
  - Slow sequential SA
    - Baseline, generates "ideal" placement
    - Very slow schedule (200k swaps per T drop)
    - Impractical, but nearly optimal
  - Fast Sequential SA
    - Vary parameters across practical range
  - Fast Self-Hosted SA

Benchmark "Programs"

- Behavioral Verilog dataflow circuits
  - Courtesy Deming Chen, UIUC
  - Compiled using RVETool into parallel programs

- Hand-coded Motion Estimation kernel
  - Handcrafted in RVEArch
  - Not exactly a circuit

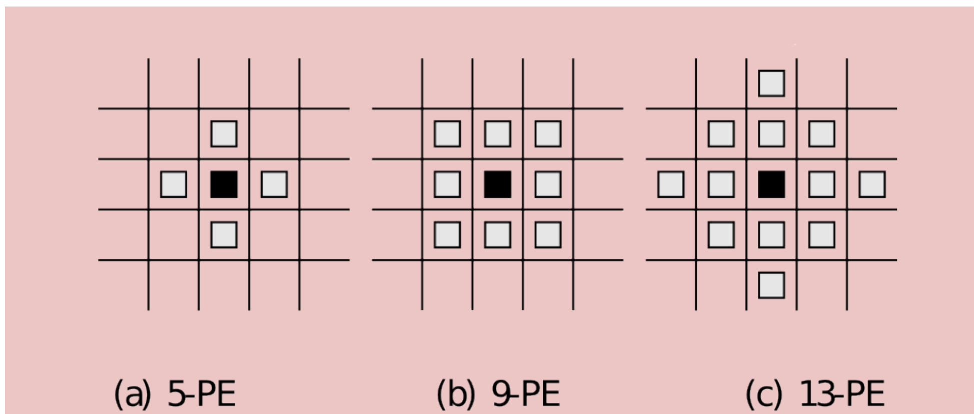
Benchmark Characteristics

Benchmark	Blocks	Nets	Cost
me	1024	998	1,242
dir	1024	760	1,785
chem	1024	749	1,250
mcm	256	244	404
honda	256	240	379
pr	256	128	181

Up to 32 x 32 array size

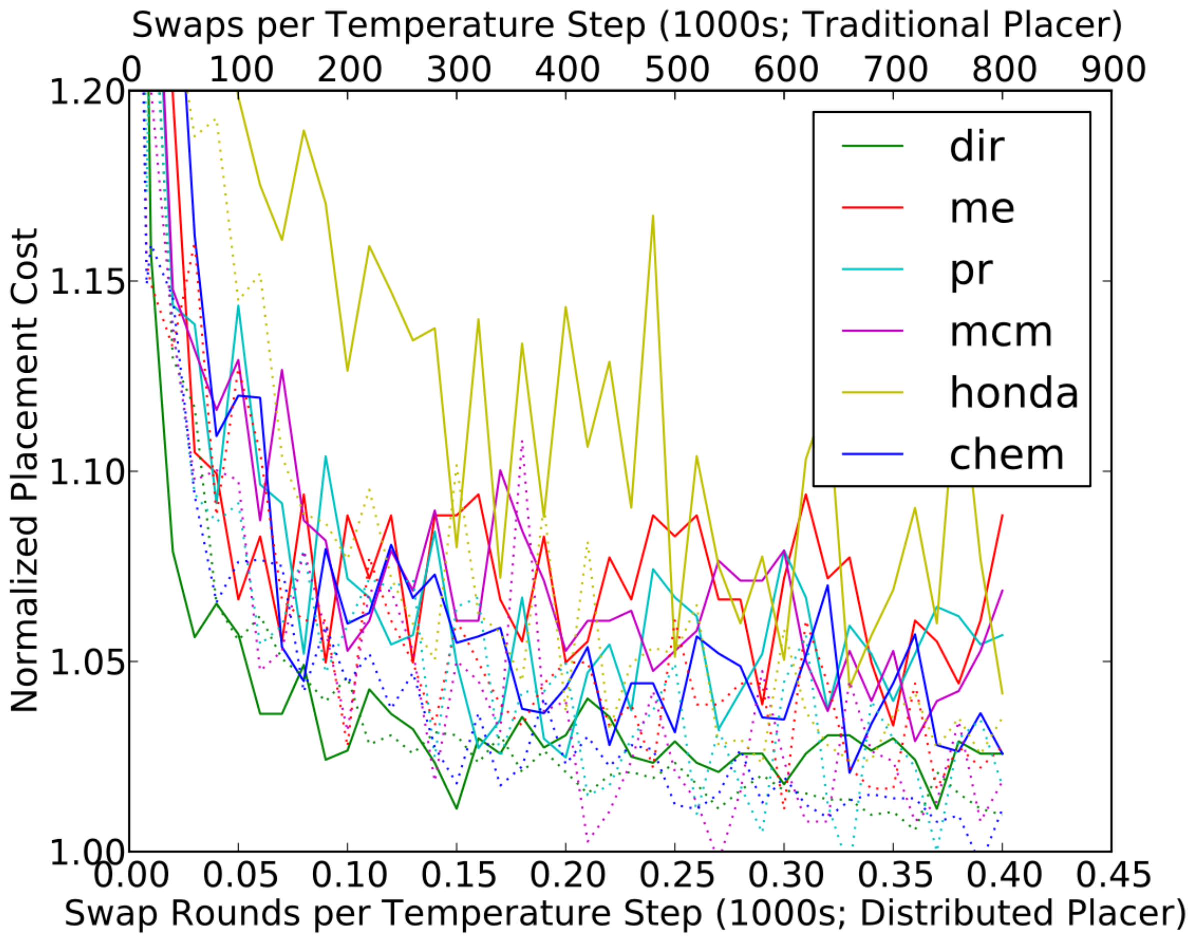
Result Comparisons

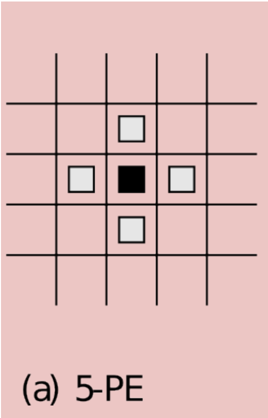
- Investigate options
  - Best neighbourhood size: 4 8 12



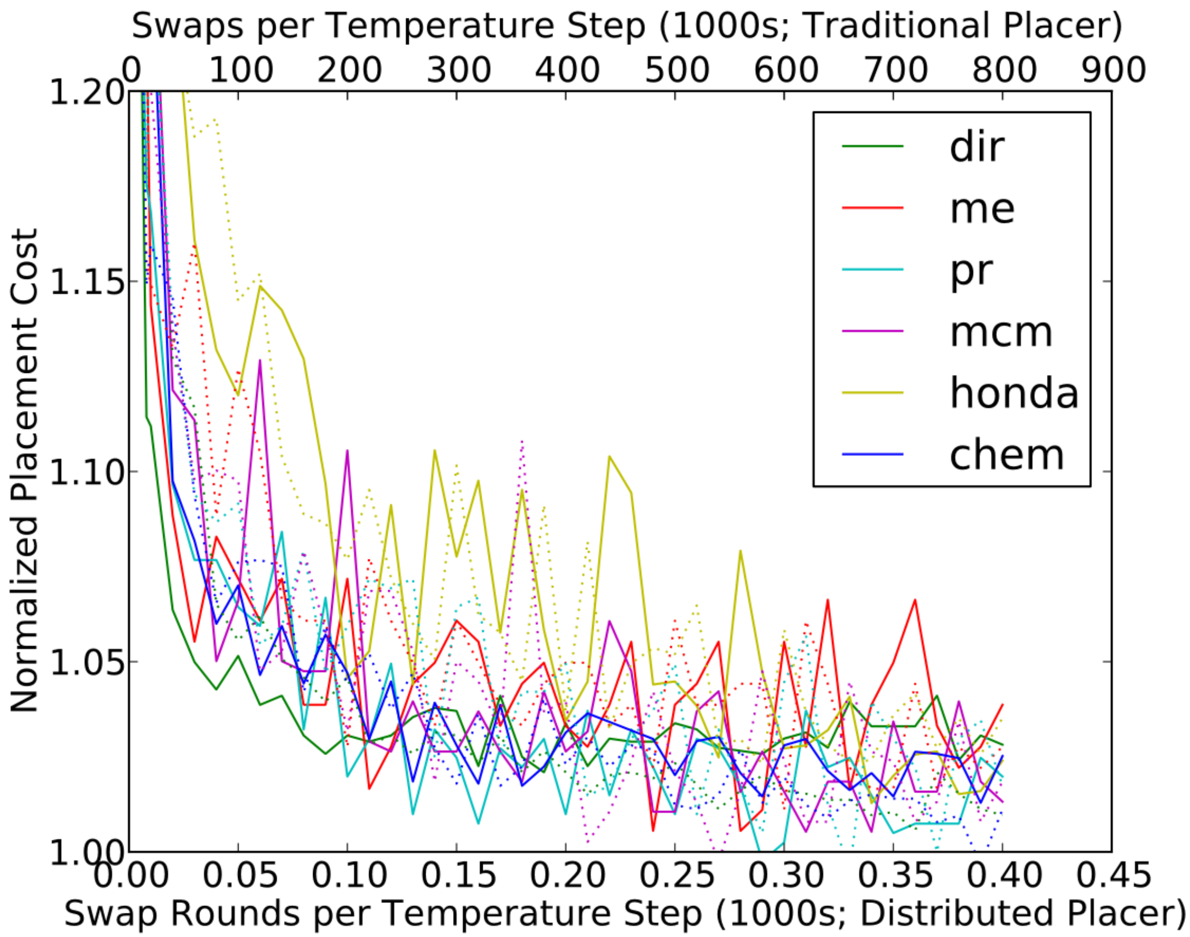
  - Update chain frequency
  - Stopping temperature

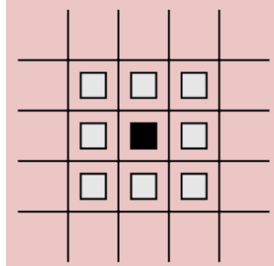
4-Neighbour Swaps





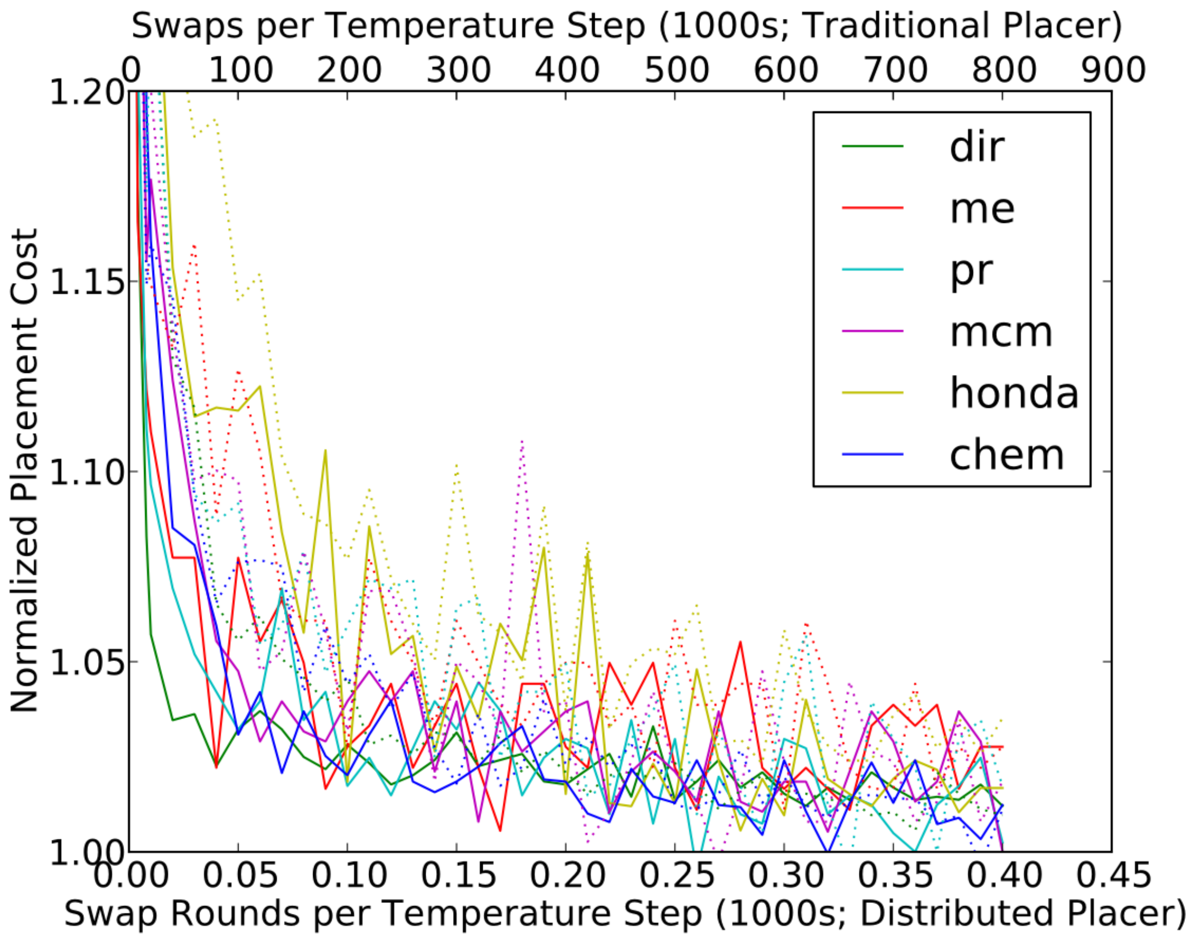
8-Neighbour Swaps

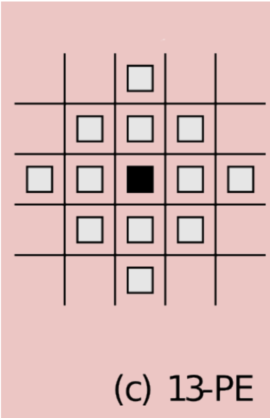




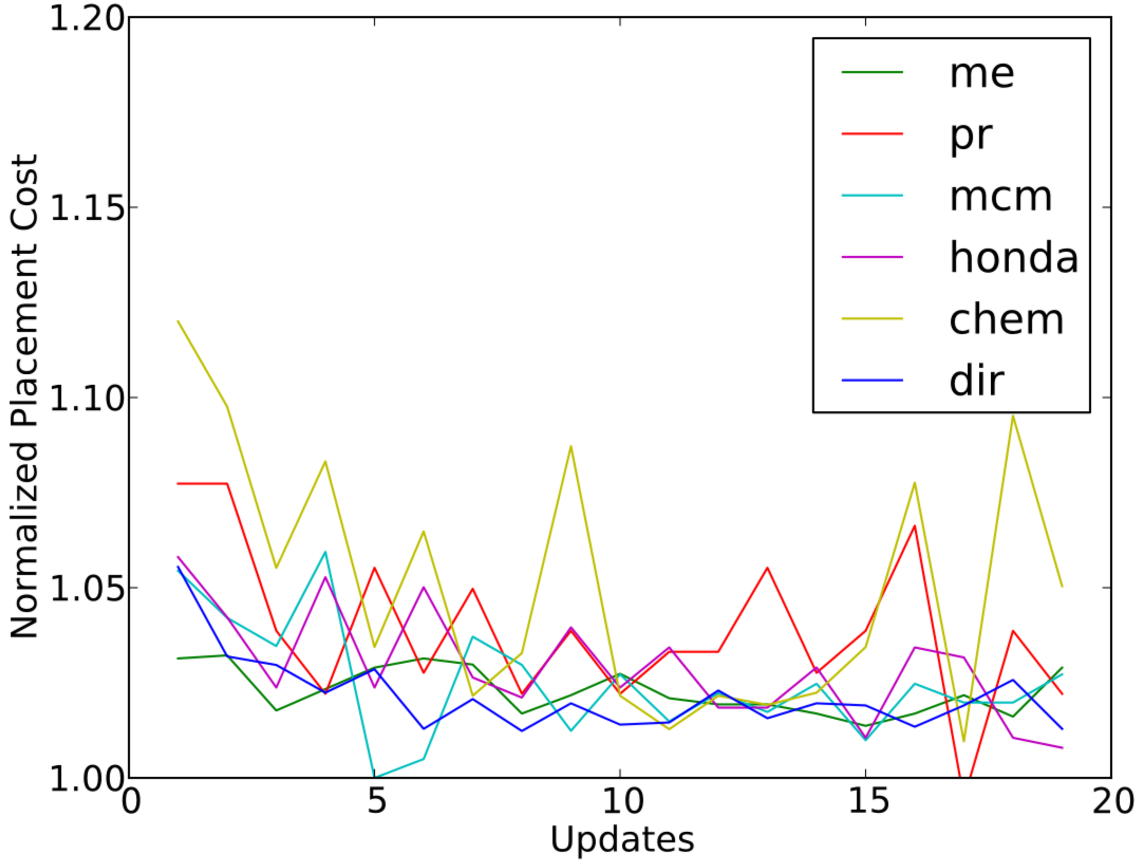
(b) 9-PE

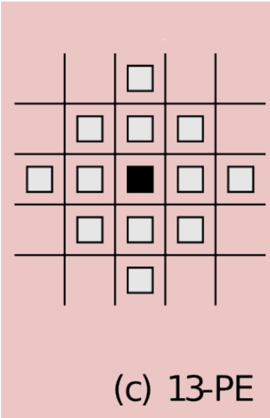
12-Neighbour Swaps



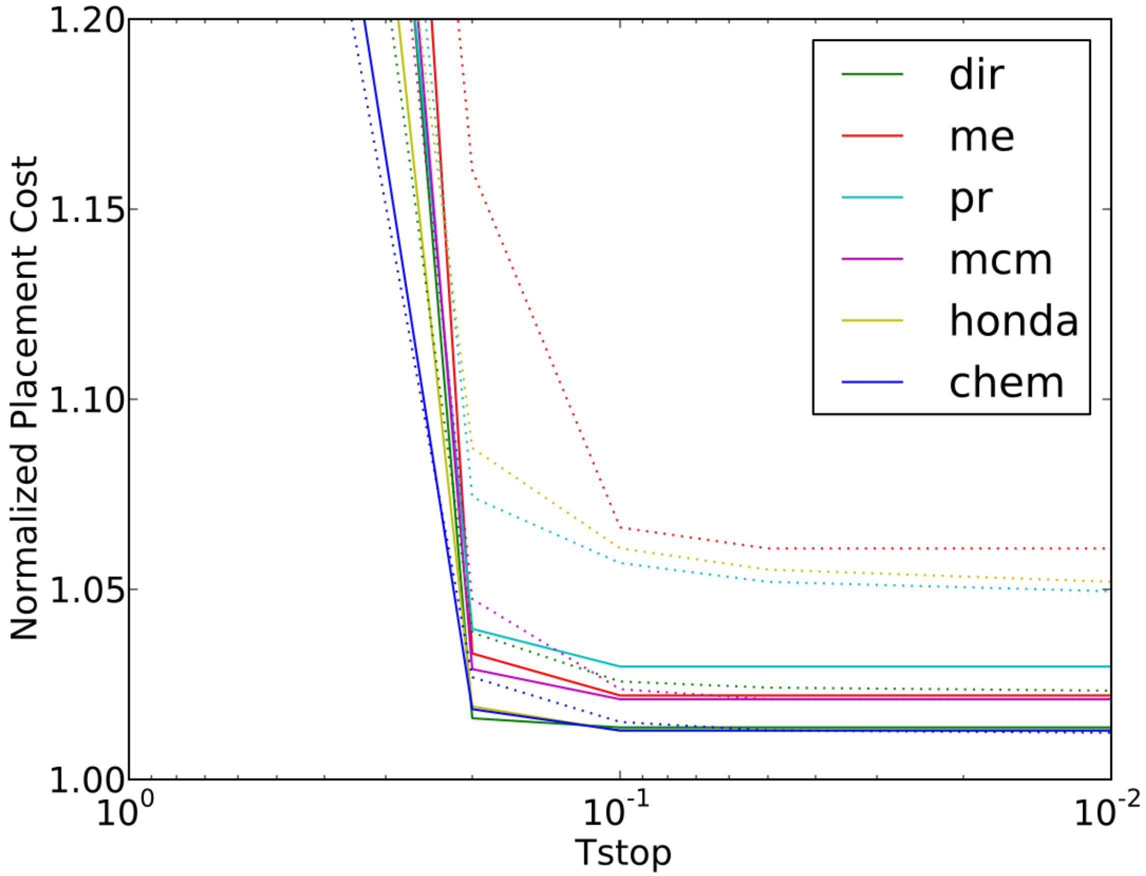


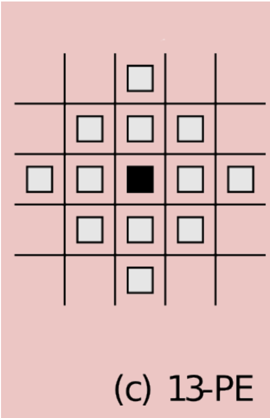
Update-chain Frequency





Stopping Temperature





Limitations and Future Work

- These results were simulated on a PC
  - Need to target real MPPA
  - Performance in <# swaps> vs <amount of communication> vs <runtime>

- Need to model limited RAM per PE
  - We assume complete netlist, placement state can be divided among all PEs
  - Incomplete state if memory is limited?
    - e.g., discard some nets?

Conclusions

- Self-Hosted Simulated Annealing
  - High-quality placements (within 5%)
  - Excellent parallelism and speed
    - Only 1/256th number of swaps needed
  - Runs on target architecture itself
    - Eat you own dog food
    - Computationally scalable
    - Memory footprint may not scale to uber-large arrays

Conclusions

- Self-Hosted Simulated Annealing
  - High-quality placements (within 5%)
  - Excellent parallelism and speed
    - Only 1/256th number of swaps needed
  - Runs on target architecture itself
    - Eat you own dog food
    - Computationally scalable
    - Memory footprint may not scale to uber-large arrays

- Thank you!

EOF