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

Graeme Smecher, Steve Wilton, Guy Lemieux

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

- $32 \times 32 = 1024$ 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 $\{\text{temperatures}\}$
   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

PEs <x,y>

bpm

pbm

Nets

bnm

nbm

Blocks

(programs)

DYNAMIC

STATIC
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

PEs exchange BlockIDs

PEs exchange nets for their BlockIDs

PEs exchange BlockIDs for their nets

(already updated)
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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Blocks</th>
<th>Nets</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>me</td>
<td>1024</td>
<td>998</td>
<td>1,242</td>
</tr>
<tr>
<td>dir</td>
<td>1024</td>
<td>760</td>
<td>1,785</td>
</tr>
<tr>
<td>chem</td>
<td>1024</td>
<td>749</td>
<td>1,250</td>
</tr>
<tr>
<td>mcm</td>
<td>256</td>
<td>244</td>
<td>404</td>
</tr>
<tr>
<td>honda</td>
<td>256</td>
<td>240</td>
<td>379</td>
</tr>
<tr>
<td>pr</td>
<td>256</td>
<td>128</td>
<td>181</td>
</tr>
</tbody>
</table>

*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

Swaps per Temperature Step (1000s; Traditional Placer)

Normalized Placement Cost

Swap Rounds per Temperature Step (1000s; Distributed Placer)

dir
me
pr
mcm
honda
chem

(a) 5-PE
8-Neighbour Swaps

Swaps per Temperature Step (1000s; Traditional Placer)

Normalized Placement Cost

Swap Rounds per Temperature Step (1000s; Distributed Placer)

- dir
- me
- pr
- mcm
- honda
- chem
12-Neighbour Swaps
Update-chain Frequency

![Graph showing normalized placement cost over updates]

- me
- pr
- mcm
- honda
- chem
- dir

Normalized Placement Cost

Updates
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 $1/256^{\text{th}}$ 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 1/256\textsuperscript{th} 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