Defect-Tolerant Logic Implementation on Nanorossbars by Exploiting Logic Mapping & Morphing Simultaneously

Yehua Su, Wenjing Rao
Outline

- Nanocrossbar & background
- Defect Tolerant Logic Implementation
  - Mapping
  - Morphing
  - Integration of Mapping and Morphing
- Simulation Results
- Conclusions
Nanoscale Fabrication

- Bottom-up
  - Small structures
  - Self-assembly process
- Why?
  - Possibly the only viable way to construct nanoelectronic systems economically

▲ Implications:
- Lead to large # of defects
- Result in regular structures
- Require reconfigurability
On the Rise: NanoCrossbar

- Advantage: Compatibility to
  - Bottom-up fabrication
  - PLA-like logic
  - Multiple nano device candidates
Change in the Flow

PLA manufacturing → testing → Good PLAs → Logic Synthesis & Optimization → 2-level Logic Function → Behavioral Description

Nano Crossbar manufacturing → testing → nano-crossbar With Defect Map

D-T Logic Implementation
Defect Tolerance

- **Challenging**
  - like testing, but harder?

- **Reconfigurability + regularity**
  - like BISR, but harder?

---

Device configurability

Other flexibilities?
Modeling

- Matrix representation for logic function & crossbar

\[ f = ab + bc' \]

\[
\begin{array}{ccc}
  a & b & c' \\
  ab & 1 & 1 & 0 \\
  bc' & 0 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{ccc}
  X & 0 & X \\
  0 & 1 & X \\
  1 & X & 1 \\
\end{array}
\]
Mapping is then...

- **Row & Column correspondence**

  \[
  \begin{array}{ccc}
  a & b & c' \\
  1 & 1 & 0 \\
  0 & 1 & 1 \\
  \end{array}
  \]

- **Cell compatibility**

  - compatible
  
  \[
  \begin{array}{ccc}
  0 & 1 & x \\
  0 & 1 & 1 \\
  \end{array}
  \]

  - mismatch
  
  \[
  \begin{array}{ccc}
  0 & 1 & 0 \\
  \end{array}
  \]
Mapping Example

- Invalid with 3 mismatches

\[ f = ab + bc' \]

\[ f' = b + ab \]
Try another way...

- Perfect mapping without mismatches

\[ f = ab + bc' \]
Backtracking Framework

Logic function

\[ \text{crossbar} \]

\[ v_1, v_2, \ldots \]

\[ p_1, p_2, \ldots \]

\[ c_1, c_2, \ldots \]

\[ r_1, r_2, \ldots \]

\[ M_1, M_2, \ldots, M_k, \ldots, M_v! x p! \]
Heuristics

- BT tree structure:
  - Col / Row interleaving

- Mapping trial selection
  - Preserving X’s
  - Efficient pruning
Challenges

• Solution space is huge
• NP-complete
  - Runtime - mapping is per chip!
  - Yield
Some mismatches are ok...

- \(f\) & \(g\) equivalent in this case

\[
f = ab + ac + bc' \\
g = abc + ac + bc'
\]
Equivalent Morphing Forms

c’d’+a’c’d+abcd+ab’c’

ac’d’+a’c’+abcd+ab’c

c’d’+a’c’+abcd+ab’c
How would Morphing help?

- Perfect
- MM -> equivalent
- MM -> changed function

Promising, but...
- Infinite # of forms
- Equivalent checking overhead
- Integration with mapping
Logic Equivalence Checking

- General Logic Eq Checking is hard
- But it’s much easier when...
  - The two functions are similar

- Method - divide and conquer
  - Shannon Expansion

\[
\begin{align*}
  f(x_1, x_2, \ldots, x_n) &= x_i' f(x_i=0) + x_i f(x_i=1) \\
  g(x_1, x_2, \ldots, x_n) &= x_i' g(x_i=0) + x_i g(x_i=1)
\end{align*}
\]

- Mismatch -> splitting var \( x_i \)
Splitting on MM var makes it easy

\[ f = c'd' + a'c'd + acd + ab'c \]

\[ g = c'd' + a'c'd + acd + ab'c \]

\[ d = 0 \]
\[ d = 1 \]
During mapping process, instead of BT, turn to LEC
If tolerable, move on to new f’
Otherwise, BT
Amortizing Runtime

Trading runtime w/ storage: Hash-table

1) Pre-profiling (off-line)
   - A mismatch analysis for function
   - limited to 1 or 2 mismatches

2) Dynamically buildup (online)
   - Invoke LEC during mapping process

<table>
<thead>
<tr>
<th>key (val, product)</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3,5), (4,12)</td>
<td>Y</td>
</tr>
<tr>
<td>(14, 8)</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>(3,7), (24,87)</td>
<td>Y</td>
</tr>
<tr>
<td>(3,7), (24,87), (10, 5)</td>
<td>Y</td>
</tr>
</tbody>
</table>

mm location: (3,7), (24,87), (10, 5)
Experimental Results: Yield

Benchmark sqrt8(size 40x16)  Benchmark sao2(size 58x20)

Mapping + Morphing

Success due to Morphing
Best improvement

Yield Comparison

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Yield</th>
</tr>
</thead>
<tbody>
<tr>
<td>con1 (9x14x2)</td>
<td>30% mapping</td>
</tr>
<tr>
<td>rd53 (32x10x3)</td>
<td>20% mapping</td>
</tr>
<tr>
<td>misex1 (32x16x7)</td>
<td>20% mapping</td>
</tr>
<tr>
<td>sqrt8 (40x16x4)</td>
<td>40% mapping</td>
</tr>
<tr>
<td>sao2 (58x20x4)</td>
<td>40% mapping</td>
</tr>
<tr>
<td>5xp1 (75x14x10)</td>
<td>20% mapping</td>
</tr>
<tr>
<td>bw (87x10x28)</td>
<td>100% mapping</td>
</tr>
<tr>
<td>9sym (87x18x1)</td>
<td>10% mapping</td>
</tr>
</tbody>
</table>
Runtime cost

Average runtime cost for finding a valid mapping (not counting the ones hitting runtime upperbound)

Benchmark con1(9x14)

Benchmark sa02(size 58x20)
Conclusions

- Defect-tolerant logic implementation becomes a fundamental issue for nanocrossbars
- Mapping helps – perfect implementation exploiting defects
- Morphing helps – some mismatches are tolerable due to equivalent functionalities
- The two schemes can be carried out in a unified framework
- Improving yield without adding cost